VRIJE UNIVERSITEIT

USING INFORMATION FLOW TRACKING TO PROTECT LEGACY BINARIES

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus prof.dr. L.M. Bouter,
in het openbaar te verdedigen ten overstaan van de promotiecommissie van de Faculteit der Exacte Wetenschappen
op woensdag 30 mei 2012 om 11.45 uur in de aula van de universiteit,
De Boelelaan 1105

door

Joanna Maria Słowińska

egenomen te Warschau, Polen
promotor: prof.dr.ir. H.J. Bos
copromotor: prof.dr.ir. H.E. Bal
Chapter 1

Introduction

Even though memory corruption vulnerabilities are inherent to C, this language is not likely to be replaced by strongly typed languages with memory safety guarantees [76; 200; 30]. Programmers are not willing to relinquish high performance, reuse of code they wrote earlier, and backward compatibility offered by C. However, the lack of safety causes serious security problems. Memory corruption vulnerabilities are reported daily [180; 150; 137], and we regularly witness attacks compromising popular software or critical networks [123; 198].

The research community has long recognised the problem, and has proposed multiple solutions. However, the existing proposals that are practical for production use prove inefficient, while the more comprehensive ones are either inapplicable to legacy software, or incur a high performance overhead.

In this thesis, we address the problem of protecting legacy C binaries against memory corruption attacks. We focus on techniques employing data flow tracking, since they are applicable to existing software, and at the same time offer a mechanism to monitor and accurately reason about a program execution. Because such monitoring is often prohibitively expensive, current systems employing data flow tracking are mainly limited to non production machines, such as malware analysis engines or honeypots. In our work, we seek solutions that would let us benefit from the wealth of information available during a run of the program, but at the same time be efficient and applicable in a timely fashion.

We divide memory corruption attacks into two classes: (1) control-diverting, that divert the flow of execution of a program to code injected or chosen by an attacker, and (2) non-control-diverting, that do not directly divert a program’s control flow, but might modify a value in memory that represents e.g., a user’s privilege level, or a server configuration string.

The research community has widely applied information flow tracking to protect against both types of memory corruptions. A popular branch of the technique, known as Dynamic Taint Analysis [62; 149], has been successfully employed to detect control-diverting attacks. In this dissertation, we further extend this mechanism
to perform attack analysis. We develop Prospector, an emulator capable of tracking which bytes contributed to a buffer overflow attack on the heap or stack. We use this information to generate signatures, which effectively stop polymorphic attacks, and also allow for efficient filtering. Further, we propose Hassle, a honeypot that is capable of generating signatures for attacks over both encrypted and non-encrypted channels.

As far as non-control-diverting attacks are concerned, several projects have attempted to employ an extended version of dynamic taint analysis to handle them. We analyse and evaluate this technique. Since the mechanism appears to have serious problems that limit its applicability, we introduce BodyArmour, a completely new method of protecting legacy binaries against buffer overflow attacks, also the non-control-diverting ones. BodyArmour tracks how pointers are used at runtime, to see when they access memory beyond buffer’s boundaries. As BodyArmour requires knowledge about memory objects used by the binary, we present Howard, a dynamic approach to unearth the necessary information.

1.1 The Problem

It has been already forty years since Anderson identified memory corruptions [10], and fifteen years since Aleph One provided a detailed introduction to stack smashing attacks [84]. The security community has recognised the problem, and has implemented various solutions in real-world systems. Static analysis has improved code quality by identifying many errors during development, but it is imprecise, and might incur both false positives and false negatives [224]. Furthermore, address space layout randomisation (ASLR) [27], data execution prevention (PaX/DEP) [154], and canaries [63] can thwart some of the attacks.

Despite all these solutions, buffer overflows alone rank third in the CWE SANS top 25 most dangerous software errors [70]. The security implications are evident—Table 1.1 lists some major buffer overflow attack outbreaks we have witnessed in recent years.

The problems persist in the real world because the adopted solutions prove insufficient, whereas more powerful protection mechanisms are either too slow for practical usage, they break backward compatibility, or require source code and recompilation. While an extensive overview of major defence mechanisms is presented in Chapter 2, we focus now on the few solutions which are the most relevant to this thesis.

Anti-virus software and network intrusion detection systems (NIDS) monitor executable files or the network traffic, and frequently search for signatures, i.e., patterns distinguishing malicious attacks from benign data. However, polymorphic attacks, zero-day attacks, and data encryption, all render signature-based solutions limited.
1.1. THE PROBLEM

<table>
<thead>
<tr>
<th>Name</th>
<th>Year</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Morris</td>
<td>1988</td>
<td>The Morris worm [83] was the earliest documented hostile exploitation of a buffer overflow. It became also the first worm to spread extensively &quot;in the wild&quot;. It infected about 6,000 UNIX machines.</td>
</tr>
<tr>
<td>Code Red</td>
<td>2001</td>
<td>The Code Red worm [225] exploited a buffer overflow in MS Internet Information Services (IIS). It spread by probing random IP addresses and infecting all hosts vulnerable to the IIS exploit. Over 359,000 unique hosts got infected in a 24-hours period on July 19th.</td>
</tr>
<tr>
<td>Slammer</td>
<td>2003</td>
<td>The SQL Slammer worm [138] exploited a buffer overflow in MS SQL Server and Desktop Engine database products. It spread rapidly, infecting most of its 75,000 victims within ten minutes.</td>
</tr>
<tr>
<td>Zotob</td>
<td>2005</td>
<td>The Zotob worm [202; 54] exploited a stack-based buffer overflow in the Plug and Play service for MS Windows 2000 and Windows XP SP1. Its outbreak was covered &quot;live&quot; on CNN television, as the network’s own computers got infected.</td>
</tr>
<tr>
<td>Conficker</td>
<td>2008</td>
<td>The Conficker worm [134; 136] spread itself primarily through a buffer overflow vulnerability in the MS Server Service. It compromised many critical networks [123; 14], and security experts estimate that it has passed a milestone of having infected more than 7 million computers [58].</td>
</tr>
<tr>
<td>Stuxnet</td>
<td>2010</td>
<td>The Stuxnet worm targeted Siemens industrial software and equipment running MS Windows. It used four zero-day attacks, including a boundary condition error [198]. Different variants of Stuxnet targeted Iranian nuclear facilities with the probable target widely suspected to be uranium enrichment infrastructure in Iran [90; 91].</td>
</tr>
</tbody>
</table>

Table 1.1: Major buffer overflow attack outbreaks.

Runtime host solutions take advantage of the wealth of information present when a vulnerable application is running to protect against attacks. Dynamic Taint Analysis (DTA), proposed by Denning et al. [77] and later implemented in TaintCheck [149], is one of the few techniques that protect legacy binaries against memory corruption attacks on control data. Because of its accuracy, the technique is very popular in the systems and security community. However, it can slow down the protected application by an order of magnitude, and in practice, it is limited to non-production machines like honeypots or malware analysis engines. Furthermore, DTA can usually detect only control-flow diverting attacks, so it does not defend against the non-control-diverting ones.

Another powerful protection mechanism comes in a form of compiler extensions. WIT [6] is an attractive framework that marries immediate detection of memory corruption to excellent performance. To harden an application, WIT requires recompilation. Unfortunately, access to source code or recompilation is often not possible in
practice. Most vendors do not share the source, or even the symbol tables, with their customers. In all probability, many programs in use today will never be recompiled at all. To protect such software, we need a solution that works for binaries.

In this thesis, we do not consider detection mechanisms such as anomaly detection or behaviour based approaches. Although they are related in the sense that they detect attacks also, they differ greatly in approach and issues (for instance, reducing the number of false positives is the core problem for these systems).

1.2 Goals

The goal of this work is to investigate solutions to protect legacy binaries against memory corruption attacks in a timely fashion. Furthermore, we do not limit ourselves to control-diverting attacks, but we also address the non-control-diverting ones. Throughout the thesis, we explore different paths to binary protection, from vulnerability signatures, to host level solutions. The ultimate goal is to detect an attack as soon as possible, and protect a binary before we even know that it is vulnerable.

Intrusion detection systems (IDS) impose a complex and conflicting list of constraints on signatures to be checked. First, signatures should incur a negligible ratio of false positives, and second, the number of false negatives should be also low. Third, we should be able to check signatures at high rates. Fourth, we should cater to polymorphic attacks with polymorphic exploits. Finally, the signatures should be generated automatically in a timely fashion, ideally without the need to replay the attack. Polymorphic attacks demand that signature generators take into account properties other than simple byte patterns. For instance, previous approaches have examined such properties as the structure of executable data, e.g., Kruegel et al. [116], or anomalies in process/network behaviour [87; 118; 130].

Since there will be always a lag between a new threat discovered and a signature being applied in IDS for detecting the threat, solutions preventing infection at the host are also desirable. We require from an attack detection mechanism to provide unsupervised detection of zero-day attacks, incur only few or no false positives, and have an acceptable performance overhead. Furthermore, it needs to be applicable to existing legacy binaries, as we cannot assume that application source code is accessible for recompilation.

Information flow tracking is a mechanism to monitor how a program uses and propagates certain data, e.g., data coming from the network. This technique can be used to accurately reason about the effects of code execution. Multiple systems employ its incarnation, Dynamic Taint Analysis, to prevent an attacker from influencing the control flow of a program. By using a virtualisation layer, information flow tracking systems can be applied to legacy binaries. However, this flexibility does not come without a cost. The technique, when implemented in software, incurs an overhead of up to 15x-20x. In this thesis, we attempt to design and develop so-
The research community has developed a number of solutions which focus on detecting control-diverting attacks. As the consequences of non-control-diverting attacks can be as serious as the control-diverting one’s, multiple projects have tried to apply an incarnation of DTA to detect both. An extended version of this technique has been also employed to track the propagation of keystrokes, and detect keyloggers. In this thesis, we aim to evaluate this method, and explore other solutions to reliably detect non-control-diverting attacks.

The goals of this thesis can be summarised in the following research questions:

Question 1  To what extent can dynamic taint analysis techniques be used to detect attacks which exploit memory corruption vulnerabilities to overwrite non-control data? And as a corollary, can we use these techniques to track the propagation of keystrokes, and detect keyloggers?

Question 2  Can we develop solutions for protecting legacy binaries against buffer overflow attacks, by means of information flow tracking, and without requiring access to source code, that can be applied in a timely fashion?

1.3 Contributions

The contributions of this thesis can be summarised in the following statements:

• We analysed and evaluated pointer tainting, an incarnation of dynamic taint analysis used to detect keyloggers and memory corruption attacks on non-control data. We argue that full pointer tainting is probably not suited for detecting privacy-breaching malware like keyloggers. Moreover, it is unclear whether another incarnation of dynamic taint analysis – limited pointer tainting – can be applied to automatically detect non-control-diverting attacks on the most popular architecture (x86) and the most popular operating system (Windows) (Chapter 3).

• We developed an emulator capable of tracking which bytes contributed to an overflow attack on the heap or stack. Whenever we recognise the protocol governing the malicious network message, we use this information to generate signatures for polymorphic attacks by looking at the length of protocol fields, rather than the actual contents. In practice, the number of false positives is negligible, and the number of false negatives is also low. At the same time, the signatures allow for efficient filtering (Chapter 4).

• Further, we propose a honeypot that is capable of generating signatures for attacks over both encrypted and non-encrypted channels (Chapter 5).
• We implemented an approach to harden binary software without access to source code or even the original symbol tables. Using our approach, we can protect binaries against buffer overflows pro-actively, before we know they are vulnerable. Besides attacks that divert the control flow of a program, we also detect and stop attacks against non-control data. Further, we demonstrated that our solution stops a variety of real exploits, and works in a timely fashion (Chapter 7).

This mechanism protects binaries by instrumenting array accesses to make sure that they are safe from buffer overflows. Since we assume absence of debugging information and symbol tables, we developed a framework which extracts arrays and array accesses from a stripped binary. It does so by tracking and observing memory access patterns (Chapter 6).

1.4 Organisation of the Thesis

The rest of the thesis is organised as follows. In Chapter 2, we offer some background information on memory corruption attacks, and current detection mechanisms. The remainder of the thesis is divided into two parts: Part I is dedicated to dynamic taint analysis, its usefulness and limitations, while Part II goes beyond dynamic taint analysis, and seeks for a more comprehensive solution to detect memory corruption attacks in legacy binaries. We start Part I by discussing dynamic taint analysis in detail. In Chapter 3, we first review basic tainting, the most popular branch of the method, used to detect control-diverting attacks (Section 3.2). Next, we evaluate pointer tainting, an incarnation of dynamic taint analysis, which has been employed to handle non-control-diverting attacks and detect privacy breaching malware (Section 3.3). In Chapters 4 and 5, we revisit basic tainting. In Chapter 4, we describe Prospector, a system designed to perform buffer overflow analysis and generate signatures, and in Chapter 5, we present Hassle which allows for applying signatures also to communication over encrypted channels. Part II focuses on an approach which protects C binaries against both control-diverting and non-control-diverting attacks by making sure that pointers in a binary do not cross buffer boundaries. In Chapter 6, we introduce a system called Howard, and we explain how we can exploit information flow tracking to analyse the layout of memory buffers in a binary, and figure out which memory regions require protection. In Chapter 7, we present BodyArmour, which instruments a binary to make sure that the buffers unearthed by Howard are safe from buffer overflows. Finally, we conclude in Chapter 8.
Chapter 2

Background - Memory Corruptions

Memory corruption attacks have become more and more sophisticated. Typically, they arrive as data over a regular communication channel, e.g., the network, and trigger pre-existing low-level software vulnerabilities. When they successfully exploit such flaws, they usually gain control over the execution of the program, or modify some critical data. In this thesis, we consider attacks written in C, and compiled for the x86 architecture, and in this chapter, we review the popular memory corruption attack techniques, along with the solutions developed as responses to them. In Section 2.1, we survey various types of buffer overflows, and in Section 2.2, we briefly discuss format string attacks.

2.1 Buffer Overflows

Exploits allow attackers to compromise machines in various ways. One way to exploit a machine is to use techniques like buffer overflows to divert the flow of execution to code injected by the attacker. Alternatively, the same exploit techniques may attack non-control data, as pointed out by Chen et al. [45]. For instance, a buffer overflow that modifies a value in memory that represents a user’s identity, a user’s privilege level, or a server configuration string.

In this section, we first sketch the key idea behind buffer overflows (Section 2.1.1), and we explain the difference between control-diverting and non-control-diverting attacks (Section 2.1.2). Later, in Sections 2.1.3- 2.1.7, we review the popular ways to exploit buffer overflow vulnerabilities, and we discuss various defence mechanisms proposed by the security community. Each of these sections is dedicated to a particular attack technique: we describe the attack mechanism, and defences proposed to defeat it.
Figure 2.1: Example of a traditional stack smashing attack.

2.1.1 Buffer Overflows – the Mechanism

In a classic buffer overflow exploit, a program receives input from an attacker, and stores it in an undersized buffer, e.g., an array. A buffer overflow occurs when the program is able to write beyond the end of the buffer. Runtime environments for languages such as Java, C#, or Python are immune to this kind of attacks. Their interpreters detect a buffer overrun, and generate an exception. C and C++ perform no such checks, though. As a result, attackers often exploit this defect by using a buffer overflow to either change the control flow of a program, or manipulate the value of a critical variable.

At the code level, buffer overflow vulnerabilities often involve a violation of a programmer’s assumptions. As a basic example, consider the function shown in Figure 2.1 that attempts to store an attacker’s provided argument `arg` in the stack buffer `buf`. The function `foo` is vulnerable to the classic stack overflow attack. The programmer assumes that the contents of `arg` fits in the stack buffer, and does not perform any checks. However, imagine that the attacker supplies an argument that is longer than the destination buffer, and it contains both an executable payload and the address at which the payload will be loaded into the program’s memory – that is, the address of `buf`. When the function executes, the `memcpy` function overwrites the stack memory with the contents of `arg`, and the stored return address gets the value of the address of `buf`. Later, once `foo` returns, the payload gets executed: the control is transferred to `buf` instead of returning to the function’s caller.

2.1.2 Control-diverting versus Non-control-diverting Attacks

Depending on the way an attacker aims to take control of the vulnerable program, we distinguish between two types of attacks: (1) control-diverting, and (2) non control-
diverting. We now define what they are.

**Control-diverting attacks**

Control diversion typically means that an instruction pointer in a process is manipulated by an attacker so that when it is dereferenced, the program starts executing instructions different from the ones it would normally execute at that point.

**Control-diverting attacks** exploit buffer overflows or other vulnerabilities resulting from low-level memory errors to manipulate a value in memory that is subsequently loaded in the processor’s program counter (e.g., return addresses or function pointers). This way attackers can execute either code that was injected by them, or an already existing code, such as a particular library function. An example of an attack against control data is shown in Figure 2.2(a): a stylised server reads a request in a struct’s buffer field and subsequently calls the corresponding handler. By overflowing `reqbuf`, an attacker changes the handler’s function pointer and thus causes the program to call a function at a different address.

**Non-control-diverting Attacks**

**Non-control-diverting attacks**, on the other hand, include memory corruption attacks against non-control data. Memory corruption attacks against non-control data manipulate data values that are not directly related to the flow of control; for instance, a value that represents a user’s privilege level, or the length in bytes of a reply buffer. The attack itself does not lead to unusual code execution. Rather, it leads to elevated privileges, or unusual replies.

A non-control-diverting attack might, for instance, exploit a buffer overflow vulnerability on a server, such that a pointer to (part of) the reply message gets overwritten. As a result, an attacker controls the memory area used for the reply, possibly causing the server to leak confidential information. This example is shown...
in stylised form in Figure 2.2(b), which shows a trivial greeting server. The server stores a pointer to its own name (which is defined as global string) in the variable `name` and then reads the name of the client from a socket. These two names are combined in a greeting message which is echoed to the client. If a malicious client overflows the `cl_name` buffer, it may overwrite the server’s `name` pointer, which means that the reply string is composed of the client’s string and a memory region specified by the attacker. The result is that information leaks out of the system.

As the instructions that are executed are exactly the same as for a non-malicious attack, this is an example of a non-control-diverting attack. For brevity, we will refer to them as non-control data attacks in the remainder of this thesis.

### 2.1.3 Stack Smashing Attacks

Stack smashing is the traditional approach to exploit a buffer overflow vulnerability. Attackers modify a function return address stored on the stack to point it to code supplied by them, and residing in a stack buffer at a known location. When the function returns, the control flow is redirected to the attacker provided data. Refer also to Figure 2.1.

This attack is probably the best known exploit of a low-level security vulnerability. The threat was identified by Anderson in the 70’s [10], while the first documented hostile exploitation dates back to 1988, when the Morris worm [83] infected 6000 UNIX machines. The exploitation technique became widely known in 1996, when Elias Levy (known as Aleph One) published an influential article [84] with a detailed explanation of stack smashing.

### Defence: Data Execution Prevention

The first attempt to mitigate stack smashing attacks came out in 1997 in the form of a non-executable stack patch for Linux [190]. The idea of the Data Execution Prevention (DEP) [133] is conceptually simple: the mechanism prevents attacker’s code smuggled into the program as stack or heap data from executing. It provides an efficient defence against direct code injection attacks, such as stack smashing attacks.

However, DEP offers no barrier against attacks executing code already present in the program, such as return-to-libc attacks or Return Oriented Programming (Section 2.1.4). Nor can it handle non-control-diverting attacks (Section 2.1.7). Despite these counterattacks to DEP, non-executable data plays an important role in the overall defence strategy. For instance, when combined with ASLR (Section 2.1.6), it significantly raises the bar for attackers – it becomes difficult to predict the location of executable memory that could be useful to a return-to-libc attack.

DEP is also limited when applied to software which uses self-modifying code and writes code to memory. For example, this behaviour can be seen in some JIT compilers, and they are normally exempt from data execution prevention.
Hardware-enforced DEP uses the NX page-table-entry bit (present in PAE page tables) to flag memory as non-executable. In the case of x86 processors with unmodified page tables, it is possible to mark data pages non-executable using a technique proposed by the PaX project [154]. This feature relies on the fact that Pentium and newer processors have separate data and instruction translation lookaside buffers (DTLBs and ITLBs). The NX bit is emulated by ensuring that data memory is never present in the ITLBs, and therefore data memory is never executed. Although innovative and effective, this implementation does not work on all x86 processors.

**Defence: Stack canary values**

Several tools protect the return address stored on the stack by confirming the integrity of the top part of a function frame on function returns. Thus they combat all attacks that modify the return address — both the traditional stack smashing attacks, and the more advanced ones discussed in Section 2.1.4 (return-to-libc attacks and Return Oriented Programming). StackGuard [63] places a canary value next to the return address, and compares it with the expected value before the function returns. The value of the canary is supposed to be secret, so that a buffer overflow never leaves the canary intact. To drop this assumption, and also to thwart the attacker from skipping over the canary, and modifying the return address only, StackShield [205] copies the return address to a safe area, and uses this copy for comparisons. ProPolice [86] additionally protects all registers saved in a function’s prologue, e.g., the frame pointer. Stack canaries are widely used in production systems. For example, Microsoft’s Visual Studio provides the /GS compiler option, and ProPolice [86] was implemented as the -fstack-protector option in the GNU C compiler.

These techniques have low overhead, since extra checks are only required in functions with local variables that could be overflowed, e.g., arrays. However, there are several attacks they cannot detect [36; 170]. For example, by using a stack overflow, an attacker might be able to modify the value of a function pointer stored on the stack (Section 2.1.5). Finally, the protection is limited to the stack, leaving heap and static buffers at the mercy of attackers.

### 2.1.4 Other Attacks Corrupting Return Addresses

As an alternative to smuggling executable code, attackers might also supply data that - when used by a program - gives them a control over the vulnerable process by executing code already present in the binary or libraries. They might either reuse entire functions, or chain together sequences of instructions scattered over the whole program. These attacks are called return-to-libc attacks and Return Oriented Programming (ROP), respectively. We discuss them in turn.
Return-to-libc  In a typical return-to-libc attack, attackers supply a command line which, when executed by the program, spawns another process. A straightforward way to achieve this is to use a stack buffer overflow to modify the return address stored on the stack, and make it point to the location of the `system` function in the C standard library. If only an attacker manages to point the `system`'s argument to the string they have provided, then they only have to wait for the vulnerable function to return to get their command executed.

In a return-to-libc attack, the return address on the stack is replaced with the address of a “useful” function, and a portion of the stack is overwritten to provide the necessary arguments. This allows attackers to call existing functions without the need to inject malicious code into a program. Although the attacker could make the code return anywhere, `libc` is the most likely target: it is always linked to the program, and it provides useful functions such as the `system` call to execute an arbitrary command. Hence the name of this class of attacks.

Return Oriented Programming  ROP is a generalisation of the return-to-libc attack. Instead of reusing entire functions from `libc`, attackers assemble their own functions by chaining sequences of instructions present in memory. ROP allows attackers to implement any functionality, so it does not limit them to functions present in binaries or libraries used by a program.

At a high level, ROP combines a large number of short instruction sequences that end with the `ret` instruction to build gadgets. Gadgets perform well-defined simple operations such as load, xor, or jump. A gadget is invoked when the processor executes a `ret` instruction with the stack pointer, ESP, pointing to the bottom word of the gadget. The execution order is controlled through a sequence of gadget addresses, which is a part of the attack payload.

Shacham shows that gadgets allow arbitrary computations by building a Turing-complete set of gadgets from instructions present in `libc`. Since then, researchers have automated the procedure of finding them. Hund et al. present a system that automatically finds gadgets in kernel code, while Dullien et al. build algorithms capable of locating Turing-complete gadget sets for many different architectures. Finally, Schwartz et al. show that it is possible to automatically find a sufficient set of gadgets in binaries as small as 20KB. Their system produces ROP payloads for 80% of Linux `/usr/bin` programs larger than 20KB.

2.1.5 Pointer Subterfuge

In response to the mechanisms we have discussed in previous sections, such as DEP, or stack canaries, attackers developed techniques to exploit buffer overflows to control values of pointers stored on the stack, the heap, or in the global data. Pointer subterfuge is a general term for exploits that involve modifying a pointer’s value, for example function-pointer clobbering or data-pointer modification. Refer to Figure 2.2.
2.1. BUFFER OVERFLOWS

Figure 2.3: Example of a traditional heap smashing attack. When B is taken out of the list of freed chunks, an attacker provided value is written to an attacker’s chosen location.

Defence: Moving local variables below stack buffers

To help thwarting pointer corruption attacks, ProPolice [86] offers some additional protection by placing arrays above all other function local variables on the stack. Also, it copies function arguments so that they sit below any buffers that could be possibly overflown. As a result, stack variables cannot be corrupted through an overflow of a local array.

This mechanism incurs no or very little overhead, and is effective. Unfortunately, it is also limited to exactly these attacks for which it has been designed. Thus it offers no protection against stack buffer underflows, overflows of heap or static buffers, or some more complex stack attacks [36; 170].

2.1.6 Heap Smashing

Pointer subterfuge led to the development of heap-specific attacks, which exploit the implementation of dynamic memory allocators [59; 12; 108; 5]. Many allocators keep heap buffers chained in doubly linked lists of allocated and freed blocks. As presented in Figure 2.3, the lists’ chaining pointers are stored in headers adjacent to their corresponding chunks, and are updated during operations such as freeing a heap object.

A buffer overflow results in corrupting the header neighbouring it in memory. Later, when the chunk is freed, and the list is manipulated by the memory allocator, an attacker provided value is stored in an attacker chosen location. Refer to Figure 2.3 for a schematic overview of the attack.
Defence: Assuring headers integrity

Similarly to stack canaries discussed in Section 2.1.4, heap canaries proposed by Robertson et al. [171] verify the integrity of header blocks associated with memory chunks on the heap. The canary contains a checksum of the chunk header seeded with a random value, and is checked when the chunk is returned to the heap management. Alternatively, memory allocators can thwart heap smashing attacks by storing metadata separately from the heap regions used by vulnerable programs.

Both solutions are efficient, and have been used in production. However, they are again limited to the attacks they were designed for, and they cannot protect any data other than heap metadata, including variables in heap-allocated objects.

Defence: Bounds checkers

Bounds checkers detect bounds violations in a program execution, and hence guard against all buffer overflow attacks. They are built upon an observation that once a pointer $p$ is assigned to an object, then all pointers derived from $p$ should stay within the bounds of the same object. The problem of bounds checking in C is a well researched topic, and the research community has proposed various approaches to realise it. In the current section, we focus on the earlier tools. The more recent ones are presented in Section 2.1.7.

Historically, there are two main approaches to bounds checking: (1) object-based, and (2) pointer-based. Systems implementing both of them have no problems detecting all types of buffer overflows we have discussed so far: stack smashing, return-to-libc, pointer subterfuge, and heap smashing, both control-diverting and non-control-diverting. The difference lies in their fundamental capabilities to protect individual fields within structures (refer to Figure 2.2), backwards-compatibility, and performance overhead. All tools presented in this section are either implemented as compiler extensions or source to source transformations, so they all require recompilation. We discuss both approaches in turn.

Object-based approaches

Object-based approaches [106; 173; 78] track object allocations, and they use a separate data structure, e.g., a splay tree, to map memory to its metadata. All pointer operations, including arithmetic and dereference, are instrumented to check that they remain within the bounds of the same object. An important feature of this approach is its backwards-compatibility. As the original C pointers are not modified in any way, it is possible to use an instrumented program with an uninstrumented library. However, there are also shortcomings. First, each pointer operation is coupled with a look up operation, yielding to a significant performance overhead. Second, this approach does not track pointers, but it only checks whether they remain in bounds of the same object. As a result, it cannot protect individual fields within structures, like in Figure 2.2.
The mechanism proposed by Jones and Kelly [106] is perhaps the first object-based solution. They use a splay tree to map memory to its metadata, letting them retrieve the base and the limit of memory objects at runtime. However, the approach cannot handle the case where an out-of-bounds pointer to an object is computed, stored and later modified to form an in-bounds address again. This weakness causes false positives. Furthermore, the system incurs significant overhead: the average slowdown is 5-6 times.

CRED (C Range Error Detector) [173] extends the approach proposed by Jones and Kelly [106] to allow for manipulations of out-of-bounds pointers, as long as they are not used for memory accesses. To retain the value of such an out-of-bound pointer, and at the same time keep track of its origin, CRED creates an auxiliary object used to store metadata of the pointer. It is further consulted to check whether the value of the pointer is in-bounds again. This rules out false positives. However, the program breaks if an out-of-bounds pointer is passed to an uninstrumented library. To improve performance, the authors observe that buffer overflows are mainly caused by user-supplied string data, and they limit checks to string operations. As a result, CRED has less than 26% average overhead in applications without heavy string usage, and 60-130% otherwise.

Dhurjati et al. [78] present a method that reduces the overhead of the above approaches. They propose to apply Automatic Pool Allocation, a compile time transformation that splits memory into “pools” corresponding to a partitioning of objects computed by a points-to analysis [9; 195]. They maintain a separate splay tree for each pool, reducing the size of the tree at each query, and also the expected cost of the lookup. The technique has the average overhead of 12%, with the maximum of 69%.

**Pointer-based approaches** Pointer-based approaches [152; 105; 57; 144; 217] track base and limit information for every pointer. This is typically implemented using “fat pointers” that augment the pointer representation in memory with the bounds of pointer’s target. The protection is enforced by instrumenting pointer dereference instructions to check whether the pointer is within the bounds associated with it.

Observe that, since this approach tracks the boundaries associated with pointers, it has the means to protect individual fields within structures (refer to Figure 2.2). As we have seen already, the object-based approach records the bounds of all memory objects, and just checks whether pointers never cross these boundaries. As a result, the protection needs to be limited to objects. To illustrate the reason for this difference, consider a scenario where two distinct pointers point to the same object and have different boundaries associated with them. Then it is only the pointer-based approach which is capable of distinguishing between them. However, the field-level protection offered by pointer-based approaches might be overly expensive. For example, the system by Xu et al. [217] does not detect sub-object overflows in the optimised mode.
The pointer-based approach suffers from a number of other problems though. First, since it changes the memory layout of pointers, it breaks the backwards-compatibility with existing code that has not been compiled with the support for fat pointers, such as libraries. In some instances it also requires source code modifications [57]. To overcome these issues, [144; 217] propose to split the metadata from the pointer. This approach partially mitigates the problem, but it also increases performance overhead. Another problem is that memory accesses to pointer variables are no longer atomic, which either requires source code modifications or might break existing multi threaded programs. The overhead is up to 150% for CCured [57], and up to almost 70% for the system by Xu et al. [217].

Defence: Control-flow integrity

Control-flow integrity (CFI) [113; 3] is designed to thwart control-diverting attacks. It ensures that the execution of a program does not stray from a restricted set of possibilities – it dictates that the execution must follow a path of a precomputed control-flow graph (CFG). The graph needs to contain all possible valid executions of the program lest false positives be possible. To determine the CFG, one could employ either static or dynamic analysis. However, none is simple in practice—static analysis has the inherent difficulty to resolve indirect branch targets, while dynamic analysis often covers only a part of a program’s execution paths. There have been several proposals to approach the problem, for example a combination of static and dynamic analysis by Xu et al. [215], value set analysis presented by Balakrishnan et al. [20], and a framework proposed by Kinder et al. [112], which combines control and data flow analysis by means of abstract interpretation.

The CFI policy is enforced at runtime, and a possible implementation may compare the target address of each control-flow transfer, i.e., each jump, call, or return, to a set of allowed destinations.

CFI does not detect non-control-diverting attacks, but it is a useful and cheap-to-enforce policy, which effectively stops the control-diverting ones. The mechanism realised by Abadi et al. [3] employs binary rewriting, and requires neither recompilation nor source-code access. The average performance overhead is 15%, with a maximum of 45%.

Defence: Dynamic taint analysis

Dynamic Taint Analysis (DTA) [66; 149; 62; 161; 100; 31], is frequently used in systems security to detect control-diverting attacks. It is very attractive since it can be applied to existing software or operating systems. The technique is implemented by transparently modifying the runtime environment. In a nutshell, untrusted data from the network is tagged as tainted, and its propagation is tracked throughout a program execution. An alert is generated (only) if an exploit takes place, e.g., when the address of a function to be invoked is tainted. The technique proves to
be reliable and generate few, if any, false positives. However, it can slow down the protected application by an order of magnitude, and in practice, it is limited to non-production machines like honeypots or malware analysis engines. Furthermore, DTA can usually detect only control-flow diverting attacks, so it does not defend against the non-control-diverting ones. We discuss DTA in great detail in Chapter 3.

E. Probabilistic defences

Randomisation and obfuscation approaches obscure the execution environment to provide a probabilistic memory safety. It means that they increase the likelihood that attacks will fail. Such solutions can be cheaper to enforce than deterministic ones. Moreover, as observed by Forrest et al. [93], bringing diversity into computer systems helps thwarting large scale attacks. Indeed, the lack of diversity facilitates large scale attacks due to the identical weakness present in all instances of a vulnerable application. In this section, we discuss two approaches to randomise the vulnerable application: ASLR focusing on the address space layout of a process, and ISR - on its code.

Address space layout randomisation Address space layout randomisation (ASLR) [153] defends against control-diverting attacks by randomly changing the positions of various regions in the address space of a process, e.g., the base of the executable and the location of libraries, heap, stack, or global variables. The idea behind this approach is that it becomes difficult for an attacker to launch a successful control-diverting attack, because he/she would have to learn or guess the values of “interesting” addresses.

Simple implementations of ASLR incur modest performance overhead, and are compatible even with legacy software. For these reasons, ASLR has been widely adopted in production use. Microsoft includes ASLR since Windows Vista [101], and Apple since Mac OS X Leopard (2007) [13; 164]. Mobile devices, such as iPhones and iPads since iOS 4.3 [60], or Android 4.0 [11], provide ASLR as well.

ASLR cannot thwart attacks that do not depend on concrete addresses, such as non-control-diverting attacks (refer to Section 2.1.7). The more serious limitation of ASLR, however, is the small number of memory layout randomisations possible on 32-bit systems [185]. As a result, an attacker may succeed after multiple retries. Alternatively, attackers may first launch a heap spraying attack to place their smuggled code in multiple locations on the heap. In this case, the likelihood that the code injected by the attacker executes increases. Indeed, with multiple buffers, it becomes less crucial to guess the value of a particular “interesting” address.

Instruction set randomisation Instruction set randomisation (ISR) [111; 24; 23] is a technique that obscures the machine instruction set in order to prevent code injection attacks. The key idea is that attackers do not know the key to the randomisation algorithm, so they would inject code that is invalid for that randomised processor,
causing a runtime exception. Observe that by randomising the underlying system’s instructions, code smuggled by an attacker would fail to execute correctly, regardless of the injection approach. The overhead of the prototypes proposed in [111; 24; 23] is very high, but a hardware implementation would be beneficial.

ISR cannot prevent any attacks other than code injection attacks, including return-to-libc or non-control-diverting attacks. Moreover, [192; 213] propose a strategy for an attacker to incrementally guess the key used by the randomisation algorithm. So varel et al. [192] also present a technique for packaging injected code which reduces the number of key bytes an attacker must determine.

2.1.7 Non-control-diverting Attacks

In December 2010, Sergey Kononenko posted a message on the exim developers mailing list about an attack on the popular exim mail server. The news was slash-dotted three days later. The remote vulnerability in question concerns a heap overflow [183] that causes adjacent heap variables to be overwritten, for instance an access control list (ACL) for the sender of an e-mail message. A compromised ACL is bad enough, but in exim the situation is even worse. Its powerful ACL language can invoke arbitrary Unix processes, giving attackers full control over the machine. (Refer to Figure 2.4 for more details about the vulnerability.) The attack is a typical heap overflow, but what makes it hard to detect is that it does not divert the program’s control flow at all. It only overwrites non-control data. W⊕X, canaries, dynamic taint analysis—all fail to stop or even detect the attack.

Since the consequences of the non-control-diverting attacks can be as serious as of the control-diverting ones, the research community has proposed various solutions to address them. In the remaining part of this section, we review the most popular approaches, and we comment on their effectiveness.

Before we continue with defence mechanisms, refer to one more non-control-diverting attack illustrated in Figure 2.5. In this simple buffer overflow, an attacker is capable of modifying the value of authorised. As we will come back to this example later in this section, observe only that the attacker does not manipulate any pointers. This is different from the example in Figure 2.2(b), where a malicious client controls a pointer to a data buffer, i.e., the server’s name pointer.

Defence: Data integrity

Data flow integrity (DFI) [40] complements control flow integrity discussed in Section 2.1.6 by handling non-control-diverting attacks. It uses the Phoenix compiler framework [135] to compute a static data-flow graph of the program being compiled. For each read instruction instr accessing a variable var, it computes the set of all possible instructions writing var which precede instr. To enforce the data flow integrity policy, DFI instruments each read and write instruction to check for unexpected, i.e., invalid, memory addresses. DFI does not cause false positives, and can
2.1. BUFFER OVERFLOWS

Figure 2.4: Exim heap overflow vulnerability. The string_vformat() function adds formatted messages to the log file, and in doing so adds bytes to the log_buffer. However, it does so without checking whether the buffer is full, allowing attackers to overwrite the adjacent access control list.

also detect many out-of-bounds reads.

WIT [6] extends DFI to improve coverage when the static analysis is imprecise. Like DFI, it also combines static points-to analysis with runtime instrumentation. However, the analysis performed by WIT is somewhat different: it assigns a colour to each object in memory and to each write instruction in the program, so that the colour of memory always matches the colour of an instruction writing it. Additionally, WIT inserts small guards between objects, which cannot be written by any instruction. They are meant to provide an extra protection against sequential buffer overflows, even if the static analysis is not precise enough. To limit runtime overhead, WIT does not checks for out-of-bounds reads.

Both DFI and WIT are implemented very efficiently: the average runtime overhead of WIT only is 7%. However, both techniques require recompilation, cannot protect fields within structures, and face problems when using libraries that are not compiled in the same fashion.

Defence: Bounds checkers

In Section 2.1.6, we have discussed two classic approaches to perform bounds checking: object-based and pointer-based. In the current section, we present recent tech-
Figure 2.5: Example of a non-control-diverting attack which does not involve any pointer dereferences.

```c
void get_private_medical_data (int uid) {
  int i=0;
  int authorized = check(uid); // result=0 for attacker.
  char patientid[8];

  printf ("Type patientid, followed by the '\$' key\n'');
  while (((c=getchar())!='$') patientid[i++]=c;

  if (authorized) print_medical_data (patientid);
  else printf ("Sorry, you are not authorized\n'";)
```

Techniques which also implement bounds checking, but try to overcome the limitations of the previous solutions. Both SoftBound [140] and BBC [7] introduced below are implemented at the level of the compiler, so require recompilation.

SoftBound [140] transforms a program by inserting code for runtime propagation and checking of pointer bounds. It enforces spatial safety using the pointer-based approach, but unlike prior systems, it does not change the representation of pointers. Instead, it uses a table data structure to map an address of a pointer in memory to the metadata for that pointer, i.e., its base and bound. The approach adopted by SoftBound allows it to easily narrow the bounds of a pointer, and in turn thwart internal object overflows, such as the example in Figure 2.2. However, in some rare cases it might also cause false positives. For example, a program that uses pointer arithmetic to traverse structures would cause false violations. SoftBound incurs an average runtime overhead of 67%. To reduce it, the system can operate in a store-only checking mode, when bounds checks are inserted only for memory writes. In this case, the average overhead is 22%.

Baggy Bounds Checking (BBC) [7] proposes a novel solution to the bounds checking problem: for each object, it enforces its allocation bounds instead of its precise bounds specified by a programmer. The key idea is, that if a memory buffer is padded with some unused bytes, then overwriting them results only in a benign buffer overflow. BBC exploits this observation to implement a remarkably efficient solution. In a nutshell, all objects are padded and aligned to a power of two. These tricks enable a concise metadata limited to a one-byte binary logarithm of a pointer’s allocation size, as compared to the base and bound of each pointer in other approaches. This information suffices to derive the base of a pointer, and perform bounds checking, while splay trees or other complex data structures used to store metadata are no longer necessary. As a result, BBC has very low overhead in practice - only 8% throughout decrease for Apache. However, it cannot protect internal fields within structures.

C. Probabilistic defences

As we discussed in Section 2.1.6, probabilistic defences aim to make attacks fail with a high probability. Address space layout randomisation and instruction set ran-
domisation, which were proposed as a countermeasure to control-diverting attacks, randomise the address space or the instruction set of a process, respectively. Both approaches focus on the code which gets executed, and are ineffective against data-only attacks. In the current section, we explore non-control-diverting attacks, so we seek solutions that prevent attackers from successfully changing the value of any variable.

**Randomise heap object sizes** DieHard [26] and Archipelago [129] use randomisation to achieve probabilistic memory safety through the illusion of an infinite-sized heap. Their memory managers randomise both the size and the location of heap objects. As a result, it is likely that buffer overflows end up overwriting only empty space. The impact on performance is modest for many applications. For example, DieHard achieves an average 8% overhead for the SPECint2000 benchmark. While DieHard consumes much more memory than conventional memory allocators, Archipelago improves on DieHard’s memory usage.

Both approaches provide a probabilistic protection against heap overflows. Thus it becomes difficult for an attacker to modify a buffer by overflowing an adjacent one. Also, since all heap metadata is kept separately from the heap, the classic heap smashing attacks introduced in Section 2.1.6 are ruled out. However, these systems offer no countermeasures against attacks inside a single heap object. For example, consider a structure consisting of an array followed by a field, either a pointer or a non-control data. In this case, nothing prevents an attacker from overflowing the array, and controlling the value of the field. Finally, both DieHard and Archipelago limit their protection to the heap.

**Data space randomisation** The idea behind data space randomisation (DSR) [28; 38] is to randomise the representation of data stored in memory. It assigns a random mask to each buffer, and it instruments the code to encrypt values written to memory: the data is xor’ed with the mask. Similarly, read operations decrypt values read from memory, again using the mask assigned to a given object. DSR [28] has an average performance overhead of around 15%.

As we discussed in Section 2.1.6, ASLR prevents attacks by making it difficult for an attacker to predict an address of a chosen memory location. DSR, on the other hand, does not hamper accessing that location, but it makes the result of a read or a write operation unpredictable. DSR is thus effective against both control-diverting and non-control-diverting attacks.

When an instruction can access, e.g., two distinct buffers, then DSR needs to decide on the relevant mask. To handle this problem, it can either select the mask dynamically, or assign both buffers the same mask. For performance reasons, DSR implements the latter solution. It uses a points-to static analysis to compute the set of objects which can be referred to by each instruction operand in the program. All objects which belong to one class share the mask. As a result, in the case of
an attack, if the overflown and the overwritten buffers share the mask, the attack succeeds. Another limitation of DSR is that, just like all other solutions we have discussed so far, it does not protect individual fields within structures, such as the ones presented in Figure 2.2.

Observe that libraries used by a protected application need special treatment, as they cannot be simply provided with encrypted arguments. One of the proposed solutions are wrappers of the library functions, which know how the masks ought to be applied. Finally, in order to protect a program, DSR needs to recompile it first.

2.2 Format String Attacks

Format string attacks [147] exploit the vulnerabilities caused by passing invalid format strings to the `printf(fmt,...)` family of functions. Internally, `printf` uses the `fmt` pointer to sweep over the format string. While processing the format string, it assumes that a list of arguments corresponding to the format directives (e.g., `%d`, `%n`) is located on the stack. In the case that `printf` is provided with no arguments, the contents of the stack is read by the function, and interpreted according to the directives contained in the format string. For example, if `fmt` points to `%n`, an integer corresponding to the number of characters written so far is stored at the memory location specified by a value which happens to be stored on the stack. By using the `%x` format directive to skip 4 bytes on the stack, the attacker can precisely set both the destination address, and the value stored there. Refer to Figure 2.6 for an example format string attack. Effectively, this vulnerability allows to overwrite any memory location with arbitrary data.

**Figure 2.6:** Example of a traditional format string attack.
2.3 Conclusions

Defence: Formatguard

FormatGuard [64] is a patch to glibc that provides a general protection against format bugs. In a nutshell, it parses the format string to determine how many arguments to expect, and if the format string requires more arguments that the actual number of arguments, FormatGuard raises an intrusion alert and kills the process.

2.3 Conclusions

We have seen that memory corruption attacks come in various guises, and have prompted a myriad of defence mechanisms. Despite all these efforts, buffer overflows alone rank third in the CWE SANS top 25 most dangerous software errors [70]. From clients to servers and from big iron to mobile phones—all have fallen victim. Moreover, security experts expect the attacks against non-control data, for which we currently have no real solutions working for binaries, to increase in the near future [191].
Part I

Dynamic Taint Analysis for Attack Detection: Usefulness and Limitations
Dynamic Taint Analysis

Dynamic Information Flow Tracking (DIFT), pioneered by Denning et al. [77], is a mechanism to monitor how a program uses and propagates certain data, e.g., user input or network data. Since the technique can be used to accurately reason about the effects of code execution, it has become an important topic in systems security research. Multiple tools, such as Vigilante [62] and Argos [161], employ an incarnation of information flow tracking to prevent attacker-provided data from executing. Others use the technique to enforce access control policies, e.g., in operating systems like HiStar [220], or modified JVMs like Trishul [141]. Researchers have also used information flow tracking to analyse malware [82; 219], automatically reverse engineer a protocol format [37; 126], or an application’s security configuration [209].

In this part of the thesis, we focus on dynamic taint analysis, a branch of dynamic information flow tracking, frequently used in systems security to detect control-diverting attacks. In a nutshell, untrusted data from the network is tagged as tainted, and its propagation is tracked throughout a program’s execution. An alert is generated (only) if an exploit takes place, e.g., when the address of a function to be invoked is tainted. The technique proves to be reliable and to generate few, if any, false positives.

In Chapter 2, we distinguish between two types of attacks: (1) control-diverting and (2) non-control-diverting. Dynamic taint analysis has been employed to combat the control-diverting attacks, and we will refer to it as basic dynamic taint analysis. Encouraged by the success of basic dynamic taint analysis, the research community has investigated ways of extending the technique to detect all memory corruption attacks, also the non-control-diverting ones. As a result, it proposed pointer tainting. The method was meant to handle both types of memory corruptions. To detect malicious memory accesses, pointer tainting implements a strict policy which disallows tainted pointers, i.e., it makes sure that tainted data never influences the address of a memory location accessed by an application. However, as we shall see in Chapter 3, the method in its pure form leads to false positives, and it needs to be curbed.

A variant of pointer tainting has been also employed to detect privacy breaching malware, such as keyloggers. It basically tracks the flow of keystrokes through an operating system and checks whether they end up in unexpected places. To detect malicious keystroke flows, this branch of pointer tainting propagates taint on a
wider range of instructions than basic dynamic taint analysis. As we will discuss in Chapter 3, this frequently leads to difficulties in controlling taint propagation. False positives are unavoidable then.

Part I of the thesis explores dynamic taint analysis in detail. We start with a discussion of both branches of the technique (Chapter 3). Because they differ significantly, both in terms of policy and efficacy, we present them separately. Section 3.2 is dedicated to basic taint analysis, and Section 3.3 — to pointer tainting.

In the following chapters of Part I of the thesis, we revisit basic taint analysis, and we explore its effectiveness in attack analysis and signature generation.

Pure basic dynamic taint analysis aims at attack detection. In our work, we try to extend the technique so that it not only raises an alert, but also helps fingerprint the attack. We want to understand what an attacker should do to make the exploit succeed. It is important for both analysis of the attack (e.g., by human security experts), and signature generation. Chapter 4 is dedicated to Prospector, a system employing basic dynamic taint analysis to analyse buffer overflow attacks. Instead of limiting the analysis to sheer taint propagation, we tag data from the network with an age stamp, so that in the case of an exploit, we can accurately pinpoint the network data which contributed to the attack. Age stamps increase the precision of our analysis. For example, they let us tell the offending bytes apart from unrelated tainted data. As we shall see in Chapter 4, the information collected by Prospector trivially yields reliable attack signatures that cater to polymorphic attacks.

In Chapter 5, we show how the technique can be extended to cater also to detection and analysis of attacks over encrypted channels. Paradoxically, a security technique like encryption makes it hard to detect, fingerprint and stop exploits. Indeed, we cannot easily inspect the contents of the network data anymore in order to generate an attack signature or check whether the network trace contains a known attack. To deal with the problem, we describe Hassle, a honeypot which uses dynamic taint analysis to detect attacks over both encrypted and decrypted channels. Upon detecting an attack, we correlate tainted memory blocks with the network trace to generate various types of signatures. As correlation with encrypted data is difficult, we retain data on encrypted connections, making tags point to decrypted data instead. This way both attack fingerprinting and signature matching become feasible.

Outline In Part I of the thesis, we explore dynamic taint analysis in detail. First, we discuss both branches of the technique: basic dynamic taint analysis and pointer tainting (Chapter 3). In Chapter 4, we show how we extend the mechanism to not only detect an attack, but also fingerprint the vulnerability. Finally, in Chapter 5, we discuss how we apply the technique over encrypted channels.
Chapter 3

Dynamic Taint Analysis for Attack Detection

3.1 Introduction

Dynamic taint analysis – the ability to monitor a program during an execution – has become a fundamental technique in systems security. It runs a (potentially vulnerable) application and observes which instructions are affected by a predefined taint source, such as a network connection. To detect an attack, the mechanism checks whether tainted data is used in a wrong way. For example, it prevents an attacker-provided shellcode from executing.

As we have mentioned already, we can divide dynamic taint analysis into two main branches: basic dynamic taint analysis and pointer tainting.

Basic dynamic taint analysis, pioneered by Denning et al. [77] and first realised for attack detection in TaintCheck [149] and Vigilante [62], is one of the most reliable methods for detecting control-diverting attacks (refer to Section 2.1). Apart from its efficacy and accuracy, the popularity of dynamic taint analysis in systems security stems from its ease of deployment. The technique focuses on protecting software without requiring any changes to target applications and/or operating systems. It is also often OS-agnostic, i.e., tools implementing the technique are not bound to a specific operating system.

Even though basic dynamic taint analysis is a very powerful and reliable method, it cannot spot all misuses of tainted data. For example, it leaves the non-control-diverting attacks undetected. To handle those, the research community has proposed pointer tainting [43; 44], an extended version of basic dynamic taint analysis. Pointer tainting disallows an attacker to directly influence memory pointers used by a vulnerable program. Although this policy stops many of the non-control-diverting attacks discussed in Section 2.1, it incurs false positives, limiting the method’s applicability.
Researchers have proposed to further extend pointer tainting and employ it to detect privacy breaching malware, such as keyloggers. By tracking the flow of keystrokes through an operating system, the mechanism checks whether data typed in by a user ends up in places where it should never be stored. For instance, some projects ensure that untrusted and unauthorised programs never receive keystroke data [219], or that Browser Helper Objects loaded by a web browser do not leak any sensitive data outside of the address space of the browser [82]. As we shall see in Section 3.3, unfortunately this incarnation of pointer tainting also incurs false positives. We argue that without external knowledge about the applications being protected, the method is probably not suited for detecting privacy-breaching malware like keyloggers.

Outline This chapter is organised as follows. We explore basic dynamic taint analysis, its most popular architectures, and limitations in Section 3.2. Section 3.3 is dedicated to both incarnations of pointer tainting, the one used to detect malware, and the other checking for keyloggers spying on users’ behaviour.

3.2 Basic Dynamic Taint Analysis

Basic dynamic taint analysis is one of the most reliable methods for detecting control diversions. It is based on the observation that to successfully compromise a system, an attacker needs to manipulate certain critical control values, such as jump targets, function addresses, or function return addresses (as discussed in Section 2.1). The technique marks all data that comes from a suspect source, like the network, with a taint tag. Tags are kept in separate (shadow) memory, inaccessible to the program under analysis. Further, taint is propagated through the system to all data derived from tainted values. An alert is raised when a tainted value affects a program’s flow of control.

We start this section by describing how basic dynamic taint analysis systems detect attacks (Section 3.2.1), next we review popular architectures (Section 3.2.2). Finally, we discuss the method’s limitations (Section 3.2.3), and the most popular extensions (Section 3.2.4).

3.2.1 Definition and Policies

To define basic dynamic taint analysis, we need to discuss three properties: how taint is introduced to a taint tracking system, how taint propagates throughout a program execution, and when an alert is raised. In this section, we explain policies employed by Argos [161]$.^1$ While the taint propagation rules, the granularity of the tagging,
or various implementation features often differ across systems, the fundamental idea stays the same. We review other popular architectures in Section 3.2.2.

Argos extends the Qemu [25] system emulator by providing it with the means to taint and track registers and memory. Qemu is a fast and portable dynamic translator that emulates multiple architectures such as x86, x86_64, POWER-PC64, etc. Unlike other emulators such as Bochs, Qemu is not an interpreter. Rather, entire blocks of instructions are translated and cached so the process is not repeated if the same instructions are executed again. Argos’ implementation extends Qemu’s Pentium architecture.

Taint introduction

Basic dynamic taint analysis marks all data that comes from a suspect source with a taint tag. Systems aiming at intrusion detection, e.g., Argos, track the propagation of data that comes from the network. Thus, whenever data is received from a network connection, the memory locations or program variables where the data is written are marked tainted. In some systems, for instance Vigilante [62] or Prospector [186], the data is also tagged with sequence numbers corresponding to an offset in the network stream. This way, in the case of an intrusion, the offending bytes can be correlated with a network packet.

Taint propagation

Taint tracking tools propagate taint through the system to all data derived from tainted values. It means that, when a tainted value is copied, the destination becomes tainted. If tainted data are involved in an arithmetical operation, many DTA approaches mark the result as tainted as well.

Tracking tainted data involves allocating extra memory storage to hold the appropriate taint flags of registers and memory. There are eight general purpose registers in the x86 architecture, and Argos allocates a taint tag for each of them. Segment registers and the instruction pointer register (EIP) are not tagged and are always clean. Since they can be altered only implicitly, and because of their role, they belong to the protected part of the system. To keep track of the physical memory, Argos implements a per byte memory tagging. Each byte of memory has a corresponding tag in the shadow memory map used to store taint flags.

To accurately track tainted data, Argos instruments Qemu translation mechanism. We have classified instrumented instructions in the following categories:

• 2 or 3 operand ALU instructions: these are the most common, and include add, sub, and, xor, etc. If the destination operands are not tainted, they result in copying the source operand tags to the destination operand tag.

• Data move operations: these operations move data from register to register, copying the source’s tag to the destination’s tag.
• **Single register operations:** shift and rotate operations belong to this category. The tag of the register is preserved as it is.

• **Memory related operations:** all load, store, push and pop belong here. These operations retrieve or store the tags from or to memory, respectively.

• **FPU, MMX, or SSE operations:** since FPU, MMX and SSE registers are involved in very specific operations that are rarely, if ever, involved in attacks, for the sake of performance, Argos ignores the corresponding operations by default.

• **Operations that do not directly alter registers or memory:** some of these operations are `nop`, `jmp` etc. For most of them we do not have to add any instrumentation code for tracking data, but for identifying their illegal use instead, as we describe in the following section.

Fortunately, we do not have to worry about special instruction uses such as `xor eax, eax` or `sub eax, eax`. These are used in abundance in x86 to set a register to zero, because unlike RISC there is no zero register available. Qemu makes sure to translate these as a separate function that moves zero to the target register. The corresponding register tags are cleaned.

**Preventing invalid use of tainted data**

Most of the observed attacks today gain control over a host by redirecting control to instructions supplied by the attacker (e.g., shellcode), or to already available code by carefully manipulating arguments (return-to-libc or ROP). For these attacks to succeed, the instruction pointer of the host must be loaded with a value supplied by the attacker. In the x86 architecture, the instruction pointer register `EIP` is loaded by instructions such as `call`, `ret`, `jmp`, `jz`, etc. By instrumenting these instructions to make sure that a *tainted* value is not loaded in `EIP`, we manage to identify all attacks employing such methods.

While these measures capture a broad category of exploits, they alone are not sufficient. For instance, they are unable to deal with format string vulnerabilities. As we explained in Section 2.2, these vulnerabilities provide an attacker with an effective way of controlling the value of an arbitrary variable in memory, e.g., a function pointer. By pointing the overwritten pointer to a malicious shellcode, the attacker can change the control flow of an application. Since the overwritten variable is not tainted, the value of the instruction pointer `EIP` remains clean, and the attack could stay off the radar. Therefore, we have extended dynamic taint analysis to also scan for code-injection attacks that would not be captured otherwise. This is easily accomplished by checking that the memory location pointed to by `EIP` is not tainted.

### 3.2.2 Popular Solutions

Basic dynamic taint analysis systems come in multiple guises. All of them aim to detect intrusion attempts, but they differ in multiple aspects, for instance the protec-
tion is provided in a system-wide or per application manner, tracking is introduced at the source code or the binary level, etc. In this section, we review the popular architectures along few dimensions: (1) the approach they employ to implement taint tracking, and (2) the scope of taint tracking. Further, we compare (3) taint propagation policies, and (4) ways of preventing invalid use of tainted data.

A. Approaches to Implement Taint Tracking

Based on distinct assumptions, and imposing different requirements with regard to practicality, range of attacks detected, or the overhead incurred, the research community has recently proposed multiple approaches to implement taint tracking systems. The popular ones include: (1) hardware-based approaches, in which a specialised hardware is employed to keep track of tainted data, (2) dynamic translation approaches, in which additional taint tracking instructions are inserted to the original binary at runtime, (3) source-to-source transformations, in which the source code of an application is augmented with taint tracking statements, and (4) interpreter-based approaches, in which interpreter programs provide a taint mode. We explain the approaches below.

**Hardware-based approaches** Minos [66] is one of the first practical systems using taint tracking. For a cost-effective deployment it relies on implementation in hardware, while the proof of concept solution is implemented on the Bochs [122] emulator. Aiming at a hardware highly optimised architecture, Minos, Chen et al. [44], or Suh et al. [196], all have to sacrifice flexibility for (potential) performance. These architectures usually extend each register and memory location (or a word of data) with a one bit taint tag. They have a single hardcoded security policy that targets memory corruption attacks, but cannot be modified to support attack analysis, or to address high-level vulnerabilities which might be language or OS-dependent.

Since hardware solutions are attractive in terms of performance, Raksha [73] proposes to combine the flexibility and robustness offered by software systems with the speed and practicality of hardware approaches. Raksha is a hardware platform, which implements dynamic taint analysis. It extends each register and memory word by four tag bits in hardware. Each bits corresponds to an independent security policy specified by software. Using policy configuration registers, software defines the rules for tag propagation and checks performed by hardware. The current prototype is developed for the SPARC architecture, and mapped onto an FPGA board.

Even though these mechanisms are interesting and (potentially) efficient, they were never widely adopted in the real world hardware, and remain only a proof of concept.

**Dynamic binary translation approaches** As dynamic translation-based taint analysis offers flexible and accurate intrusion detection systems, it has become very popular in the systems and security community - we have witnessed a series of
publications in the last few years in top venues, including SOSP [62], CCS [219], NDSS [149], EuroSys [161; 100], Usenix [49], RAID [31], and ACSAC [186; 160].

Taint analysis systems implementing dynamic translation usually do not require any changes to the target applications and/or OS being protected. They achieve it by transparently modifying the runtime environment (frequently using virtualisation), and instrumenting instructions to perform taint tracking. For instance, Vigilante [62] and Taintcheck [149] use binary instrumentation frameworks (Nirvana [132] for Windows and Valgrind [146] for Linux, respectively). Argos [161], Prospector [186], XenTaint [100], and Aftersight [49] are built on top of the Qemu [25] processor emulator, while Minemu [31] uses a dedicated lightweight emulator designed with taint analysis in mind.

Dynamic translation-based approaches offer great flexibility. For example, some systems do not stop at attack detection, but using more informative tags than just the binary taint flag, they generate attack signatures (e.g., self-certifying alerts in Vigilante) or perform vulnerability analysis (e.g., Prospector). A recent project, DiskDuster [17], also goes much further than just attack detection. By employing extended taint analysis, DiskDuster aims at system recovery by removing all traces of complicated attacks in a semi-automated manner.

While flexible, instrumentation-based mechanisms usually incur significant overhead (15-20x, and often more). To decrease the overhead, XenTaint [100] dynamically switches execution between a heavily instrumented Qemu and fast Xen, depending on whether tracking is required. The overhead for an I/O bound application, netcat, is modest. The worst case scenario, however, a CPU bound ssh, incurs expensive transition overheads (switching between Xen and Qemu), and slows down the performance 150x compared to native. As a result, all these systems are too slow to be used in production environments. In practice, their usage is limited to non-production machines like honeypots or malware analysis engines.

Minemu [31], a recently published x86 emulator, significantly improves the state of the art. Specifically, it brings down the slowdown due to taint analysis to 1.5-3x for real applications. Minemu uses a dedicated lightweight emulator designed with taint analysis in mind. By using SSE registers instead of the normal general purpose registers for tainting, Minemu alleviates the register pressure that might otherwise occur due to dynamic taint analysis. Also the memory layout is especially crafted to make it cheap to propagate taint to and from the taint map.

**Source-to-source transformation** Another group of approaches to implement taint tracking involves source-to-source transformation. Xu et al. [216] achieve modest overhead (10% for server applications) and detect a wide range of attacks by employing a program transformation. Taint introduction, tracking and policy enforcement are implemented using a source-to-source transformation on C programs.

Observe that systems which perform taint tracking at hardware- or emulator level operate on machine instructions, and are function agnostic. (Some of them, such as
Argos, are even process agnostic.) As a result it is not trivial for them to define security policies in high level terms, e.g., function arguments. In the case of source-to-source transformations though, more fine grained information is available. For instance, Xu et al. [216] may define the taint policies in terms of SQL commands or printf arguments. For example, they check for tainted control-characters or commands in SQL queries, or preclude tainted format directives, such as $n. As a result, the system detects not only memory corruption attacks, but also SQL injections or format string attacks.

A drawback of the source-to-source transformation approach is that it requires source code available for recompilation. In practice, source code is not always accessible due to copyright issues, and in some cases it might not be available at all. As a consequence, a lot of software cannot be protected, and is left at the mercy of attackers.

**Interpreter-based approaches** Various environments for interpreted languages offer automatic tainting for language variables. These include the taint mode in Perl [156], and Python [71], or safe levels in Ruby [199]. They constrain data to be untainted (trustworthy) or previously sanitised when reaching sensitive sinks.

**B. Scope of Taint Tracking**

Another characteristic that differentiates between dynamic taint analysis systems is the scope of taint propagation: operating system-wide or per-process. Representatives of the first group (OS-wide) include: Minos [66], Argos [161], and Xen-Taint [100]. Per-process solutions are exemplified by Vigilante [62], TaintCheck [149], Minemu [31], and Xu et al. [216].

Per-process mechanisms protect specific applications, and not whole operating systems. They work with virtual addresses, and as such are not able to handle DMA or memory mapping. Also, they usually instrument individual services and do not protect the OS kernel at all. On the other hand, per-process solutions tend to be more cost-effective, as taint tracking is performed for selected applications only.

**C. Taint Propagation Policies**

The policies employed by Argos (discussed in Section 3.2.1) offer a good overview of the basic taint propagation rules. First, an analysis engine marks all data that comes from a suspect source, like the network, with a taint tag. Next, it propagates taint through the system to all data derived from tainted values. Specifically, when tainted values are copied, the destinations are also tainted, when they are used as sources in ALU operations, the destinations are also tainted, etc. The vast majority of taint tracking systems implement the above rules. An exception is Vigilante, which limits taint propagation to copy operations.
D. Ways of Preventing Invalid Use of Tainted Data

Basic dynamic taint analysis was designed to handle control-diverting attacks. For most of these attacks to succeed the instruction pointer must be loaded with a value supplied by the attacker, that is tainted. As a result, the protection policy on the x86 architecture might be simply implemented by instrumenting the small set of instructions which load EIP: call, ret, jmp, jz, etc. This generic OS agnostic rule is employed by most of the hardware- and instrumentation-based mechanisms.

Systems implementing per-process analysis are not OS agnostic, so they can achieve more fine-grained taint tracking, and cater to higher level attacks. For instance, they often detect format string attacks by inspecting format directive passed to printf-family functions. Alternatively, they can stop SQL injections by e.g., precluding tainted commands in SQL queries. In a similar way they protect against attacks that are based solely on altering arguments of critical functions such as system calls, e.g., by checking whether execve arguments are tainted.

It is worth mentioning that some of the instrumentation-based OS agnostic approaches, e.g. Argos, when hinted about the guest OS, also insert additional checks of the system calls arguments, such as execve.

Finally, a few systems attempt to detect non-control-diverting attacks by employing pointer tainting, which we have briefly mentioned before. Since the method poses serious problems, we defer the explanation to Section 3.3.

3.2.3 Limitations

Basic dynamic taint analysis is a powerful technique, effectively used to detect control-diverting attacks. However, it does not come without its shortcomings. In this section, we explain two main drawbacks of the method: (1) it is powerless in face of non-control-diverting attacks and privacy-breaching malware, and (2) it does not detect attacks immediately when a memory corruption happens, but only later when a corrupted pointer is about to be used. Clearly, arbitrary, possibly malicious, actions may be performed by the victim program between the time of detection and the time of attack. It causes several difficulties, e.g., the attack analysis becomes much harder.

As we have mentioned already, researchers try to address the first limitation with an incarnation of dynamic taint analysis, known as pointer tainting. We discuss it in Section 3.3. To partially overcome the second drawback, in Chapter 4 we propose Prospector, a system employing extended basic dynamic taint analysis to figure out which tainted bytes contributed to a buffer overflow attack. As the analysis still takes place a posteriori, in Chapter 7 we describe BodyArmour, a solution which is capable of detecting memory corruption attacks at the moment of an overflow.
3.2. BASIC DYNAMIC TAINT ANALYSIS

Memory corruption and the (in)effectiveness of basic tainting

As we have seen in Chapter 2, for exploits, the root cause of numerous control-diverting and non-control-diverting attacks is the dereference of a pointer altered by an attacker. However, exploitable vulnerabilities which do not involve any pointer manipulations are also likely (refer to Figure 2.5).

For instance, a stack smashing attack overflows a buffer on the stack to change the function’s return address. Heap corruption attacks typically use buffer overflows or double frees to change the forward and backward links in the doubly linked free list. Alternatively, buffer overflows may overwrite variables on the heap or stack directly, e.g., function pointers or other critical variables such as the authorised variable from the example in Figure 2.5. In a format string attack, a member of the printf() family is given a specially crafted format string to trick it into using an attacker-provided value on the stack as a pointer to an address where a value will be stored.

Basic taint analysis raises alerts only for dereferences due to jumps, branches, and function calls/returns. All non-control-diverting attacks stay off the radar. Thus, a memory corruption attack that modifies a pointer to a program variable, or the variable itself, e.g., a user’s privilege level, would go unnoticed.

Privacy-breaching and the ineffectiveness of basic tainting

One may want to employ dynamic taint analysis to detect whether a ‘possibly malicious’ program is spying on users’ behaviour. A basic approach could work by marking the keystrokes typed by the user as tainted, and monitoring the taint propagation in order to inspect whether the software in question accesses tainted sensitive data.

However, basic taint analysis is weak in the face of translation tables that are frequently used for keystrokes. Assuming variable \( x \) is tainted, basic taint analysis will not taint \( y \) on an assignment such as \( y = a[x] \), even though it is completely dependent on \( x \). As a practical consequence, data from the keyboard loses its taint almost immediately, because the scan codes are translated via translation tables. The same is true for ASCII/UNICODE conversion, and translation tables in C library functions like \( \text{atoi()} \), \( \text{to_upper()} \), \( \text{to_lower()} \), \( \text{strtol()} \), and \( \text{sprintf()} \).

As a corollary, basic taint analysis is powerless in the face of privacy-breaching malware. As data loses its taint early on, it is impossible to track if it ends up in the wrong places.

Time of detection versus time of attack

Dynamic taint analysis raises an alert when tainted values are used in an unsafe way, e.g., as a jump target. It means that the method does not detect the memory corruption itself, but only malicious control flow transfers. Unfortunately, the control flow transfer occurs at a (sometimes much) later stage. As a result, there is no guarantee
that the program integrity has not been violated before the attack is detected. Also, at this point, some of the offending bytes which contributed to the attack might have already been overwritten making the attack analysis harder. To actually find an exact moment of the attack, expensive external methods building on record and replay would be needed.

3.2.4 Extensions to Basic Dynamic Taint Analysis

Multiple systems extend the set of basic taint propagation policies, discussed in Section 3.2.1, to cater to a wider range of attacks. Two popular additions are (1) propagating taint on implicit information flows, and (2) propagating taint or raising an alert when a tainted pointer is dereferenced.

Implicit information flows

'Pure' dynamic taint analysis misses implicit information flows. Implicit information flows have no direct assignment of a tainted value to a variable (which would be propagated by basic tainting). Instead, the value of a variable is completely determined by the value of tainted data in a condition. For instance, if \( x \) is tainted, then information flows to \( y \) in the following code: \( \text{if } (x=0) \ y=0; \text{ else } y=1; \). As basic dynamic taint analysis does not track implicit information flows, false negatives are quite likely. As pointed out by Cavallaro et al. [41], purely dynamic techniques cannot detect implicit flows. The authors explain that it is necessary to reason about the assignments that take place on the unexecuted program branches, and also provide a number of reasons that make the problem intractable for x86 binaries.

Several approaches try to address implicit information flows by finding a trade-off between false positives and false negatives. A few systems supplement dynamic analysis with static analysis to enforce information-flow policies [174; 53]. Others exploit additional high level knowledge. For example, Xu et al. [216] apply a source-to-source transformation and track indirect assignments through carefully selected control flows. Taint is propagated on conditional statements which contain a single equality check and a constant assignment, e.g., \( \text{if } (\text{data} == '+' ) \ast \text{dst} = '\' \). The authors claim that this policy is sufficient to handle common scenarios which involve implicit information flows, and at the same time does not introduce false positives. The problem becomes much more complex when dynamic taint analysis is applied to binaries. The current state of the art is DTA++ [109], which concentrates on the common case of information-preserving transformations, such as the conversion of data from one format to another. First, DTA++ diagnoses branches responsible for dropping taint, such as the assignment in the conditional statement above. (This is known as under-tainting.) Next, using an offline analysis, the system determines the extra propagation rules required. Finally, the new rules are applied during future runs of the pre-analysed binary. This approach significantly reduces false positives, but its scope is currently limited to a narrow range of
3.3. EVALUATING THE PRACTICALITY OF POINTER TAINING

applications.

**Pointer tainting**

The second extension of basic dynamic taint analysis is known as *pointer tainting*, and is explicitly designed to handle non-control flow diverting attacks. The research community has also proposed to use a variant of this method to track the flow of keystrokes through the operating system, and ensure that untrusted and unauthorised programs never receive keystroke data. Again, we postpone the discussion of these techniques until the following section.

### 3.3 Evaluating the Practicality of Pointer Tainting

This section evaluates *pointer tainting*, an incarnation of dynamic information flow tracking, which has recently become an important technique in system security. Two variants of pointer tainting have been used for the following purposes: detection of privacy-breaching malware (e.g., trojan keyloggers obtaining the characters typed by a user), and detection of memory corruption attacks against non-control data (e.g., a buffer overflow that modifies a user’s privilege level). In both of these cases the attacker does not modify control data such as branch targets, so the control flow of the target program does not change. Phrased differently, in terms of instructions executed, the program behaves ‘normally’. As a result, these attacks are exceedingly difficult to detect.

**Pointer tainting as advertised is attractive.** It is precisely these difficult to detect, stealthy non-control-diverting attacks that are the focus of pointer tainting [44]. At the same time, the technique works against control-diverting attacks also. We will discuss pointer tainting in more detail in later sections. For now, it suffices to define it as a form of dynamic information flow tracking (DIFT) [196] which marks the origin of data by way of a taint bit in a shadow memory that is inaccessible to software. By tracking the propagation of tainted data through the system (e.g., when tainted data is copied, but also when tainted pointers are dereferenced), we see whether any value derived from data from a tainted origin ends up in places where it should never be stored. For instance, we shall see that some projects use it to track the propagation of keystroke data to ensure that untrusted and unauthorised programs do not receive it, e.g., Panorama [219]. By implementing pointer tainting in hardware, as in Raksha [73], the overhead is minimal.

Pointer tainting is very popular because (a) it can be applied to unmodified software without recompilation, and (b) according to its advocates, it incurs hardly (if any) false positives, and (c) it is assumed to be one of the only (if not the only) reliable techniques capable of detecting both control-diverting and non-control-diverting attacks without requiring recompilation. Pointer tainting has become a unique and
extremely valuable detection method especially due to its presumed ability to detect non-control-diverting attacks. As mentioned earlier, non-control-diverting attacks are more worrying than attacks that divert the control flow, because they are harder to detect. Common protection mechanisms like address space randomisation and stackguard [29; 63] present in several modern operating systems are ineffective against this type of attack. The same is true for almost all forms of system call monitoring [163; 94]. At the same time, the consequences of a successful non-control-diverting attack may be as severe as with a control-diverting attack. For example, exploiting a buffer overflow that modifies a user’s privilege level gives attackers full control of the machines.

An ability to tell privacy breaching malware from legitimate binaries sounds also very attractive. We have witnessed already trojan keyloggers that have been active for years (often undetected). In one particularly worrying case, a keylogger harvested over 500,000 login details for online banking and other accounts [168; 197].

However, pointer tainting is not working as advertised. Inspired by a string of publications about pointer tainting in top venues [44; 45; 219; 82; 73; 218; 206; 74], several of which claim zero false positives, we tried to build a keylogger detector by means of pointer tainting. However, what we found is that for privacy-breaching malware detection, the method is flawed. It incurs both false positives and negatives. The false positives appear particularly hard to avoid. There is no easy fix. Further, we found that almost all existing applications of pointer tainting to detection of memory corruption attacks are also problematic, and none of them are suitable for the popular x86 architecture and Windows operating system.

In this section, we analyse the fundamental limitations of the method when applied to detection of privacy-breaching malware, as well as the practical limitations in current applications to memory corruption detection. Often, we will see that the reason is that “fixing the method is breaking it”: simple solutions to overcome the symptoms render the technique vulnerable to false positives, false negatives, or evasion.

Others have discussed minor issues with projects that use pointer tainting [72], and most of these have been addressed in later work [74]. To the best of our knowledge, nobody has investigated the technique in detail, nobody has shown that it does not work against keyloggers, and we are the first to report the complicated problems with the technique that are hard to overcome. We are also the first to show the implications experimentally.

In summary, the contributions of our work presented in this section are:

1. an in-depth analysis of the problems of pointer tainting on real systems which shows that it does not work against malware spying on users’ behaviour, and is problematic in other forms also;
2. an analysis and evaluation of all known fixes to the problems that show that they all have serious shortcomings.

We emphasise that this section is not meant as an attack on existing publications. In our opinion, the authors of previous papers underestimated the method’s problems. We hope that our work will be useful for others to avoid making the same mistakes we made when we worked on our ill-fated keylogger detector, and perhaps allow them to develop improved detection techniques.

Outline The remainder of this section is organised as follows. In Section 3.3.1 we introduce pointer tainting, and in Section 3.3.2 the test environment. The meat of the work is in Sections 3.3.3-3.3.8: Sections 3.3.3 and 3.3.4 provide an analysis of the problem of false positives and negatives in pointer tainting, Sections 3.3.5, 3.3.6 and 3.3.7 evaluate possible counter-measures, and finally in Section 3.3.8 we give an overall assessment of the technique. We discuss related work in Section 3.3.9 and conclude in Section 3.3.10.

3.3.1 Definition

Pointer tainting is explicitly designed to handle non-control-diverting attacks. Because of the two different application domains, pointer tainting comes in two guises, which we will term limited pointer tainting (for detecting non-control data attacks) and full pointer tainting (for detecting privacy breaches). Unfortunately, both have shortcomings. To clarify the problems, we first explain the two variants in detail.

For now, we just describe the basic ideas. We will see later that they both need to be curtailed to reduce the number of false positives. Since both limited and full pointer tainting extend basic dynamic taint analysis, discussed in Section 3.2.1, we only briefly summarise the rules for taint propagation applied in the basic case:

1. all data from suspect sources is tainted;
2. when tainted data is copied, or used in arithmetical calculations, the taint propagates to the destination;
3. taint is removed when all traces of the tainted data are removed (e.g., when the bytes are loaded with a constant) and a few other operations.

Limited pointer tainting (LPT): alerts on dereferences. Systems that aim at detecting non-control data attacks apply a limited form of pointer tainting [44; 74]. Defining a tainted pointer as an address that is generated using tainted data, taint analysis is extended by raising an alert when a tainted pointer is dereferenced. So:

4a. if $p$ is a tainted pointer, raise an alert on any dereference of $p$. 
Doing so catches several non-control-diverting attacks, but cannot be realistically applied in the general case. For instance, any pointer into an array that is calculated by way of a tainted index would lead to an alert when it is dereferenced, causing false positives. Again, this is common in translation tables. For this reason, LPT implementations in practice prescribe that the taint of an index used for a safe table lookup is cleaned. In Sections 3.3.5 and 3.3.6 we evaluate various such cleaning techniques. As a consequence, however, LPT cannot be used for tracking keystrokes. As soon as the tainted keystroke scan-code is converted by a translation table, the taint is dropped and we lose track of the sensitive data.

**Full pointer tainting (FPT): propagation on dereferences.** Full pointer tainting extends basic taint analysis by propagating taint much more aggressively. Rather than raising an alert, full pointer tainting simply propagates taint when a tainted pointer is dereferenced. So:

4b. if $p$ is a tainted pointer, any dereference of $p$ taints the destination.

FPT looks ideal for privacy-breaching malware detection; table conversion preserves the original taint, allowing us to track sensitive data as it journeys through the system. Panorama [219] is a powerful and interesting example of this method. It tries to detect whether a new application $X$ is malicious or not, by running it in a system with FPT. Sensitive data, such as keystrokes that are unrelated to $X$ (e.g., a password you type in when logging to a remote machine) are tagged tainted. If at some point, any byte in the address space of $X$ is tainted, it means that the sensitive data has leaked into $X$, which should not happen. Thus, the program must be malicious and is probably a keylogger.

### 3.3.2 Test Environment

To get a handle on the number of false positives, we track the spread of taint through the system for increasingly sophisticated versions of pointer tainting. The idea is that we mark sensitive data as tainted and monitor taint propagation over the system. If taint spreads to benign applications that should never receive tainted data, we mark it as a false positive.

For the experiments we use Qemu 0.9 [25] with vanilla Ubuntu 8.04.1 with Linux kernel 2.6.24-19-386 and Windows XP SP2. Depending on the test, we modified the Qemu emulator to taint either all characters typed at the keyboard, or all network data. We then propagate the taint via pointer tainting (using rules 1, 2, 3, and either 4a or 4b). Whether network or keyboard is tainted will be clarified when we discuss our experiments. The taint tag is a 32-bit value, so that each key stroke or network byte can have a unique colour, which helps in tracking the individual bytes.

To measure the spread of taint we repeatedly inspect the taintedness of registers at context-switch time. Tainted registers in processes that do not expect tainted input
indicate unwanted taint propagation. The more registers are tainted, the worse the problem. The situation is particularly serious if special-purpose registers like the stack pointer (ESP) and the frame pointer (EBP) are tainted. Once either of these registers is tainted, data on the stack also gets tainted quickly. Indeed, many accesses are made by dereferencing an address relative to the value of ESP or EBP.

The measurements are conservative in the sense that even if the registers are clean at some point, there may still be tainted bytes in the process’ address space. Moreover, we only check taint in registers during context switch time, probably again underestimating processes’ taintedness. Taint may also leak across the kernel-userspace boundary in other ways, e.g., when tainted bytes are memory mapped into a process’ address space. In other words, the real situation may be worse than what we sketch here. However the conservative approach we have implemented is sufficient to present the severity of the problem of false positives.

Context switches in Linux occur at just one well-defined point: the `schedule()` function. The scheduler is called either directly by a blocking call that will lead to a call to `schedule()` in the kernel (a voluntary context switch), or by interrupts and exceptions (a forced switch). For instance, a timer interrupt handler discovers that a process has used up its quantum of CPU time and sets a flag of the current process to indicate that it needs a reschedule. Just prior to the resumption of the user space process, this flag is checked and if it is set, the `schedule()` function is called.

The two methods differ in the way registers are saved. In particular, the general purpose x86 registers EAX, ECX and EDX are not saved on the call to `schedule()` on the voluntary context switch. The calling context is responsible for saving the registers and restoring them later. On interrupts and exceptions, all registers are saved at well defined points. The implication is that on voluntary switches, when we measure the state of the registers on return from `schedule()`, we ignore the taintedness of the above three registers. Whether they are tainted or not is irrelevant, as they will be overwritten later anyway. On a forced switch, when we inspect the condition of the process on the return from interrupt/exception handler, we look at all the registers. Summarising, in any case the state of the registers being presented is captured once the original values are restored after the context switch. That reflects the state of processes rather than the state of kernel structures.

For a complete picture we also monitor the taintedness inside the kernel, during the `context_switch()` function.

As we cannot perform detailed analysis of Windows, we measure the state of the registers whenever the value of the cr3 register changes. This x86 register contains the physical address of the top-level page directory and a change indicates that a new process is scheduled. For user mode processes the measurement is performed once the processor is operating in user mode. This way we are sure that we present the state of the process, and not some kernel structures used to complete the context switch.
3.3.3 Problems with Pointer Tainting: Experimental Results

When we started implementing a keylogger detector by means of pointer tainting, we observed that taint spread rapidly through the system. We analyse now the problem of taint explosion both experimentally (the current section) and analytically (Section 3.3.4).

False Positives in LPT

To confirm the immediate spread of taint in limited pointer tainting (LPT), we used the emulator that taints data coming from the network. Both for Linux and Windows alerts were quickly raised for benign actions like configuring the machine’s IP address using `ifconfig` on Linux, and allowing for automatic network configuration in Windows.

This is wrong, but not unexpected. We have already seen the causes in the LPT discussion in Section 3.3.1: without appropriate containment mechanisms, LPT propagates taint when combining an untainted base pointer and a tainted index. Further, dereferencing such an address triggers an alert. This is exactly what happened in our experiment. We discuss ways of addressing this problem in Sections 3.3.5 and 3.3.6.

Taint Explosion for FPT

To evaluate the spread of taint in full pointer tainting, we introduce a minimal amount of (tainted) sensitive information, and observe its propagation. After booting the OS, we switch on keystroke tracking (which taints keystroke data), and invoke a simple C program, which reads a user typed character from the command line. This is all the taint that is introduced in the system. Afterwards we run various applications, but do so using `ssh`, so no additional taint is added.

Figure 3.1 shows how taint spreads to the kernel and some of the most frequently scheduled processes. Aside from a few boxes on the very left, almost all applications and the kernel have at least half of the considered registers and EBP and ESP tainted. Clearly, taint spreads very rapidly to all parts of the OS. Moreover in both this and the remaining experiments, `tar` and `gzip` should be completely clean as we use a bash script hardcoding the input and output filenames.

Figure 3.2 shows a similar picture for Windows XP. Here, performing simple tasks, we provide the guest operating system with new tainted keystrokes during the whole experiment. In more detail, after the system has booted we first switch on keystroke tagging. Thus, from this point onward data typed by the user is considered tainted. Next, we launch Internet Explorer, `IEXPLORE.exe`, and calculator, `calc.exe`. We perform simple web browsing, thus delivering tainted data to the Internet Explorer process. However, we do not provide the calculator with any typed characters, but we use solely the mouse. Finally, we switch off keystroke tagging,
3.3. EVALUATING THE PRACTICALITY OF POINTER TAINTING

Figure 3.1: The taintedness of the processes constituting 90% of all context switches in Linux. In this and all similar plots the following explanation holds. The y-axis shows the processes responsible for 90% of all context switches. Multi-threaded application are collapsed to one line. The x-axis is divided into scheduling intervals, spanning 50 scheduling operations each. Time starts when taint is introduced in the system. In an interval, several processes are scheduled. For each of these, we take a random sample from the interval to form a datapoint. So, even if gzip is scheduled multiple times in an interval, it has only one datapoint. A datapoint consists of two small boxes drawn on top of each other, separated by a thin white line. The smaller one at the top represents the taintedness of \texttt{EBP} and \texttt{ESP}. The bottom, slightly larger one represents all other registers. We use three colours: lightgrey means the registers are clean, darkgrey means less than half of considered registers are tainted, and black means that half or more are tainted (very dirty). Absence of a box means the process was not scheduled. The figure shows that apart from a first few datapoints, all applications become very tainted very quickly.

and consult the kernel debugger to dump values of the \texttt{cr3} register of running processes.

3.3.4 Problems with Pointer Tainting: Analysis

The above results show that pointer tainting without some containment measures is not useful. It is not possible to draw meaningful conclusions from the presence of taint in a certain process. A crucial step in the explosion of taint through the system is the pollution of the kernel. Once kernel data structures are polluted, many processes that share accesses to these structures pick up the taint. As LPT simply raises an alert (and we have already seen how quickly this happens in a table lookup with tainted index), this section focuses on the more interesting case of FPT, and we consider how taint spreads through the system in practice. In this section, we give an example that shows how taint pollution happens in practice in common Linux kernel code.

As mentioned earlier, incorrectly tainting \texttt{EBP} or \texttt{ESP} is particularly bad, as ac-
processes to local variables on the stack are made relative to EBP, and a ‘pop’ from the stack with a tainted ESP will taint the destination. Unfortunately, the Linux kernel has numerous places where EBP and/or ESP incorrectly pick up taint. Rather than discussing them all, we explain as an example how a common operation like opening a file, ends up tainting EBP, as well as various lists and structures in the kernel. The main purpose is to show that taint pollution occurs on common code paths (like opening files), and can be the result of complex interactions.

A. Taint Pollution by Opening Files - a Case Study

Taint pollution occurs due to calls to the open() system call in various ways. For the following analysis, we extended the emulator with code that logs the progression of taint through the system at fine granularity. We then manually analysed the propagation through the Linux source code by mapping the entries in the log onto the source.

The Linux Virtual Filesystem [32] uses dentry objects to store information about the linking of a directory entry (a particular name of the file) with the corresponding file. Because reading a directory entry from disk and constructing the corresponding dentry object requires considerable time, Linux uses a dentry cache to keep in memory dentry objects that might be needed later. The dentry cache is implemented by means of a dentry_hashtable. Each element is a pointer to a list of dentries that hash to the same bucket. The d_hash field of the dentry object contains pointers to the adjacent elements in the list associated with a single hash value.
The real work in the open() system call is done by the filp_open() function which at some point accesses the dentry cache by means of the __d_lookup() function to search for the particular entry in the parent directory (see Figures 3.3 and 3.4). The second argument, struct qstr * name, provides the pathname to be resolved, where name->name and name->len contain the file name and length, and name->hash its hash value.

**Phase 1: taint enters kernel data structures.** To see how taint propagates incorrectly in __d_lookup(), let us assume that the filename was typed in by the user, so it is tainted. The hash is calculated by performing arithmetical and shift operations on the characters of the filename, which means that the hash in line 4 is tainted. The d_hash() function in line 5 first produces a new hash value from both the dentry object of the parent directory and the tainted hash of the filename (so that the new hash is also tainted) and then returns the head of the list of dentries hashing to this new hash value. This is the address of the element in the table with the index derived from the new hash value. The address is calculated as the table’s base pointer plus tainted index and is therefore tainted. head in line 5 becomes tainted.

As is common in the Linux kernel, the singly linked list of dentries is constructed by adding a struct hlist_node field in a type that should be linked; in this case the dentry node. Each hlist_node field points to the hlist_node field of the next element, and a hlist_head field points to the start of the list. We iterate over the list (line 9), searching for the dentry matching the name, which will be found, if the file has been opened previously (which is quite common). Notice that people often edit a file and then compile it, or edit and print it, or copy it and then edit the copy, and so the same file is repeatedly accessed, what causes a successful lookup. Also, some programs are used frequently, like bash for example, what again prompts the case of the dentry object located in the cache.

**Phase 2: dentry gets tainted.** During the iteration, head (and later node) contain pointers to the list’s hlist_head and hlist_node link fields. Of course, these fields themselves are not interesting and the real work takes place on the associated dentry object. Therefore, the macro hlist_for_each_entry_rcu in line 9 performs a simple arithmetical operation to produce the address of the dentry (the
container_of macro is used), which results in tainting dentry (line 9).

**Phase 3: EBP gets tainted.** Now that the dentry object is found in the cache, the `filp_open()` function calls `__dentry_open()`, passing to it the tainted address of the dentry object. This function almost immediately loads the EBP register with the tainted address of the received dentry object. As a result, taint spreads rapidly through the kernel’s address space.

**Phase 4: pollution of other structures via lists.** Taint spreads further across the kernel by dint of pointer arithmetic prevalent in structures and list handling. Linked lists are especially susceptible to pollution.

When we read a field of a structure pointed to by a tainted address, the result is tainted. Similarly, when we insert a tainted element elem to a list list, we immediately taint the references of the adjacent nodes. Indeed, the insertion operation usually executes the assignment `list->next=elem` which taints the next link. If we perform a list search or traversal, then the pointer to the currently examined element is calculated in the following fashion: (1) `curr=list`, (2) `curr=curr->next`, and so the taintedness is passed on from one element to another.

If a list element is removed from one list and entered into another, the second list will also be tainted. For instance, if a block of data pointed to by a tainted pointer is freed, free lists may become tainted. When the block is returned on a new (unrelated) memory allocation, the taint may spread to other structures and lists. By means of common pointer handling, the pollution quickly reaches numerous structures that are unrelated to the sensitive information.

Let us continue the example of opening files. As we explained earlier, the `__dentry_open()` function is provided with the tainted address of the dentry object. This function executes the instruction `inode=dentry->d_inode` to determine the address of the inode object associated with dentry. The assignment taints inode as its value is loaded from the tainted address dentry plus an offset. Next, once the new file object file is initialised, we execute `head(inode->i_sb->s_files)` as we insert the file into the list of opened files pointed to by head (i_sb is a field of the filesystem’s superblock), so the head is tainted. As a result, the file insert operation immediately taints the references of the adjacent nodes in the list.
Finally, when the kernel has finished using the file object, it uses the `fput()` function to remove the object from the superblock’s list and release it to the slab allocator. Without going into detail, we mention that dentry cache look-ups are lockless read-copy-update (RCU) accesses and that, as a result, the file objects end up being added to the RCU’s per-CPU list of callbacks to really free objects when it is safe to do so. The list picks up the taint and, when it is traversed, spreads it across all entries in the list. The callback is responsible for releasing the tainted object to the slab.

B. False Positives and Root Causes of Pollution

It is clear that due to false positives, limited pointer tainting and full pointer tainting in their naive, pure forms are impractical for automatically detecting memory corruption attacks and sensitive information theft, respectively. We have seen that taint leaks occur in many places, even on a common code path like that of opening a file. The interesting question is what the root causes of the leaks are, or phrased differently, whether these leaks have something in common.

After manually inspecting many relevant parts of the Linux kernel, we found three primary underlying causes for taint pollution. First, the tainting of EBP and ESP. These pointers are constantly dereferenced and once they are tainted, LPT raises alerts very quickly, while FPT spreads taint almost indiscriminately as the stack becomes tainted.

Second, not all pointers are tainted in the same way and not all should propagate taint when dereferenced. If A is a tainted address and B is an address calculated relative to A (e.g., \( B = (A + 0x4) \)), then B will be tainted. However, in many cases it might be unreasonable to mark \( *B \) as tainted. For example, let’s assume that tainted A points to a field of a structure, `file_name`. Next, B is derived to hold the base address of this structure, \( B = A - \text{offset}(\text{file\_name}) \), and B becomes tainted. Now, depending on a security policy, we may or may not wish to mark \( B->\text{file\_handler} \) as tainted. However, if all these structures are organised in a list, we certainly do not want to propagate taintedness to the next element of a list, \( B->\text{next} \). On the other hand, if a pointer is itself calculated using tainted data (\( C = A+EAX \), where EAX is tainted), the taint should be propagated, as C might be pointing to a field in a translation table. Notice that all these cases are hard to distinguish for emulators or hardware.

Third, if pointer tainting is applied only for detecting memory corruption attacks on non-control data, rather than tracking keystrokes and other sensitive data, taint may leak due to table lookups, as discussed in Section 3.3.1.

C. False Negatives: is Pointer Tainting Enough?

While false positives are more serious than false negatives for automatic detection tools, a system that misses most of the attacks is not very useful either. ‘Pure’
pointer tainting in LPT or FPT does not have many false negatives, but even without any containment of taint propagation, pointer tainting does not detect all the attacks it is designed for. For instance, LPT will detect modification of non-control data by means of a format string attack, or a heap overflow that corrupts freelist link pointers. However, it will miss modification of non-control data by means of a direct buffer overflow. Limited mitigation may be possible by checking system call arguments, as is done in Vigilante [62], but the fundamental problem remains. Consider for instance, an attack that modifies a variable indicating the user’s privilege level. No tainted pointer is dereferenced, so the attack remains undetected.

Similarly, FPT and LPT both miss implicit information flows. Implicit information flows have no direct assignment of a tainted value to a variable (which would be propagated by pointer tainting). Instead, the value of a variable is completely determined by the value of tainted data in a condition. For instance, if \( x \) is tainted, then information flows to \( y \) in the following code: \( \text{if} \ (x=0) \ y=0; \text{else} \ y=1; \). As we do not track implicit information flows, false negatives are quite likely. This is particularly worrying if FPT is used to detect potential privacy-breaching malware, as it gives the untrusted code an opportunity to ‘launder’ taint. As pointed out by Cavallaro et al. [41], purely dynamic techniques cannot detect implicit flows. The authors explain that it is necessary to reason about the assignments that take place on the unexecuted program branches, and also provide a number of reasons making the problem intractable for x86 binaries.

If we are to have any hope at all of catching sophisticated privacy-breaching malware with FPT, we need to detect and raise an alert immediately when taint enters the untrusted code, lest it be laundered. As soon as untrusted code is allowed to run it can trick the system into cleaning the data.

Like most work on pointer tainting, we assume that false negatives in pure pointer tainting are not the most important problem. However, to deal with the false positives, we are forced to contain taint propagation in various ways. Doing so will reduce the false positive ratio, but the opportunities for false negatives will increase significantly.

### 3.3.5 Containment for LPT and FPT: EBP/ESP Protection

We have seen that without containment, pointer tainting is not usable. We now evaluate ways to control the spreading.

The first cause of pollution in FPT (and false positives in LPT) discussed in Section 3.3.4-B is tainting of ESP and EBP. We can simply remove it with minimal overhead by never applying pointer tainting to tainted values of EBP or ESP. However, on occasion EBP is also used as a temporary general purpose register. Having analysed a number of scenarios that involved a tainted EBP, we devised a simple heuristic, and clean EBP whenever its value is big enough to serve as a frame pointer on the stack. Doing so introduces false negatives into the system in case EBP is used as a temporary general purpose register and serves as a pointer.
3.3. EVALUATING THE PRACTICALITY OF POINTER TAINTING

We implemented the above restriction in our emulator and again evaluated the spread of taint through the system. The results for Linux, shown in Figure 3.5, indicate that while the spread has slowed down a little compared to Figure 3.1, taint still propagates quickly. Observe that EBP is still tainted occasionally. This is correct. It means that EBP was used as a container. We do not show the plot for Windows, but we will show a combined plot later (Figure 3.8).

3.3.6 LPT-specific Containment Techniques

The most important cause of false positives in LPT involves taint pollution via table lookups. As LPT uses pointer tainting only to detect memory corruption attacks, rather than tracking sensitive data, we should try to prevent taint from leaking due to lookups. Proposed solutions revolve around detecting some specific pointer arithmetic operations, as proposed by Suh et al. [196], bounds-checking operations [44; 73], and more recently pointer-injection [110; 74]. Because of conversion tables, none of these techniques are suitable for FPT.

A. Detecting and Sanitising Table Accesses

Suh et al. [196] sanitise table lookups even when the index is tainted and assume that the application has already performed the necessary bounds checks. The method is impractical as it requires us to recognise table lookups, while many architectures, including the popular x86, do not have separate instructions for pointer arithmetic. On x86, we can only instrument those instructions that calculate an address from base and index. Then we propagate the taint of the base and skip that of the index.
However, the use of \texttt{add} and \texttt{mov} instructions to calculate pointers is extremely common in real-world code and these cannot be monitored in the same way. As a result, this method leads to many false positives.

\textbf{B. Detecting Bounds Checks}

Chen et al. [44] argue that most table lookups are safe even if the index is tainted, as long as the index was properly bounds-checked. Thus, to reduce false positives, we may try to detect bounds-checks at runtime, and drop the operand’s taint. Bounds-checks are identified by a \texttt{cmp} instruction of the index with an untainted operand. As pointed out in Raksha [73], the \texttt{and} instruction is also frequently used for bound checks. Thus we additionally clean the first source operand of \texttt{and} if the second operand is clean and has a value of $2^n - 1$.

While simple and fast, the method suffers from false positives and negatives, some of which were noted by others [72; 73; 74]. We are the first to find the last two discussed below.

In many conversion tables, a lookup simply returns a different representation of the input and cleaning the tag leads to false negatives. For instance, the taintedness of suspicious input is dropped as it passes through translation tables, even if the data is then used for a buffer overflow or other exploit. Incorrectly dropping taint in a way that can be exploited by attackers is known as taint laundering (Section 3.3.4-C). False negatives also occur when the \texttt{cmp} and \texttt{and} instructions are used for purposes other than bounds checking.

In addition, the method is prone to false positives if code does not apply bounds-checking at all or uses different instructions to do so. Many lookups take place with an 8-bit index in a 256-entry table and are safe without bounds check.

Furthermore, taint often leaks to pointers in subtle ways that are not malicious at all. For instance, many protocols have data fields accompanied by length fields that indicate how many bytes are in the data field. The length may be used to calculate a pointer to the next field in the message. A subtle leak occurs when the length is bounds checked, but the check is against a value that is itself tainted. For instance, a check whether the length field is shorter than IP’s total length field (minus the length of other headers). A comparison with tainted data does not clean the index.

Yet another way for taint to escape and cause false positives, is when check and usage of the index are decoupled. For instance, the index is loaded into a register from memory and checked, which leads us to clean the register. However, by the time the index is actually used, it may well have to be loaded from the original (tainted) memory address again, because the register was reused in the meantime. This again leads to a tainted dereference and thus an alert. We see that for various reasons, raising alerts immediately on tainted dereferences is likely to trigger many false positives.

We have just discussed why current solutions that revolve around detecting bounds checks and table accesses are insufficient and incur both false positives and false neg-
Chapter 3

3.3. EVALUATING THE PRACTICALITY OF POINTER TAINTING

atives. We implemented these policies, and experiments performed on the emulator confirm our objections: control flow diversions were reported instantly. In addition, on architectures like $x86$, there is little distinction between registers used as indices, normal addresses, and normal scalars. Worse, the instructions to manipulate them are essentially also the same. As a result this problem is very hard to fix, unless a particular coding style is enforced.

C. Pointer Injection Detection

This brings us to recent and more promising work by Katsunuma et al. [110], which prevents memory corruption attack by detecting when a pointer is injected by untrusted sources. The most practical implementation of this approach, Raksha [74], identifies valid pointers, which it marks with a $P$ bit, and triggers an alert if a tainted pointer is dereferenced that is not (derived from) a valid pointer.

For this purpose, it scans the data and code segments in ELF binaries to map out in advance all legitimate pointers to statically allocated memory and marks these with a $P$ bit. To do so, it has to rely on properties of the SPARC v8 architecture that always uses two specific instructions to construct a pointer and has regular instruction format. In addition, it modifies the Linux kernel to also mark pointers returned by system calls that allocate memory dynamically ($mmap$, $brk$, $shmat$, etc) with a $P$ bit. Furthermore, as the kernel sometimes legitimately dereferences untrusted pointers it uses knowledge of SPARC Linux to identify the heap and memory map regions that may be indexed by untrusted information (and uses the kernel header files to find the start and end addresses of these regions).

The method effectively stops false positives, but false negatives are possible. For instance, it is possible to overflow a buffer to modify an index that is later added to a legitimate address. The resulting pointer would have the $P$ bit set and therefore a dereference would not trigger alerts.

More worrying is that the method is very closely tied to the Linux/SPARC combination and portability is a problem. For instance, it would not work well on $x86$ processors running Windows. First, $x86$ makes it much harder to detect pointers to statically allocated memory. Second, we cannot modify the kernel, so that we are forced to add specific handling for several system calls in hardware or emulator (and the number of system calls in Windows is large). Third, we cannot identify kernel regions that may be indexed with untrusted data. To err on the safe side, we would have to assume that certain data values are pointers when really they are not, and that the entire kernel address space could be pointed to by untrusted data. As a result, we expect many false negatives on $x86$. Even so, while limited to OS/architecture combinations similar to Linux/SPARC, Raksha is the most reliable LPT implementation we have seen.
3.3.7 FPT-specific Containment Techniques

Section 3.3.4-B identified primary causes for pollution. We now try to remove them without crippling the method.

White lists and black lists

The simplest solution is to whitelist all places in the code where taint should be propagated using pointer tainting, or alternatively, blacklist all places where it should not be propagated. Neither scheme works in practice. Whitelisting is impossible unless we know all software in advance (including the userspace programs) and well enough to know where taint should propagate. This is certainly not the case when we are monitoring potential malware (e.g., to see if it is a keylogger). It is also difficult for large software packages (say, OpenOffice, or MS Word), or any proprietary code. Finally, whitelisting only a small subset of the places reduces FPT to taint analysis with a minimal extension.

Blacklisting also suffers from the problem that we have no detailed knowledge over all programs running on a system. In addition, the number of taint leaks is enormous and blacklisting them all is probably not feasible. Notice that even if we managed to blacklist part of the software, including the Linux kernel and a few applications, for instance, that still would not be enough. Assume that one of the programs we do not blacklist causes unrelated data to be tainted. Next, if such data is communicated to other processes, they become tainted, and a false alarm is raised. Such unrelated tainted data can enter kernel structures during system calls.

Finally, blacklisting and whitelisting both have a significant impact on performance. Thus, we do not consider whitelisting or blacklisting a feasible path to remedy FPT.

Landmarking

Easy fixes like EBP/ESP protection and white/black-listing do not work. In this section, we discuss a more elaborate scheme, known as landmarking, that contains taint much more aggressively. Unfortunately, as a side effect, it significantly reduces the power of pointer tainting which leads to many false negatives. In addition, it still incurs false positives and significantly increases the runtime overhead. Nevertheless, this is the most powerful technique for preventing taint explosion we know. A similar technique appears to have been used in Panorama [219], but as our landmarking is slightly more aggressive and, hence, should incur fewer false positives, we will limit the discussion to landmarking.

Recall that the second primary cause of unwanted taint propagation is due to pointers being relative to a tainted address: if \( A \) is a tainted address, and an address \( B \) is calculated relative to \( A \) (e.g., \( B = A + 0x4 \)), then \( B \) is tainted as well, even though tainting \( *B \) is often incorrect. As a remedy, we will let \( B \) influence the taintedness of \( *B \) only if it \textit{itself} was calculated using tainted data. So, with \( A \) and \( EAX \) tainted,
3.3. EVALUATING THE PRACTICALITY OF POINTER TAINTING

```c
typedef struct test_t {
    int i;
    struct test_t* next;
} test_t, *ptest_t;

ptest_t table[256] = ...; // initialised
ptest_t i1 = table[index]; // tainted
ptest_t i2 = i1->next; // clean
int i3 = i1->i; // clean
```

**Figure 3.6:** An example of landmarking.

...we will exclude $B = A + 0 \times 4$ from taint propagation, but keep $C = A + EAX$. Unfortunately, it is difficult for hardware or emulators to decide which operations involved in the calculation of an address should make the resultant pointer influence the taintedness of a dereference, and which not.

For this purpose, we introduce **landmarks**. Landmarks indicate that an address is 'ready to be used for a dereference’. We have reached a landmark for $A$, if all tainted operations up to this point were aimed at calculating $A$, but not a future value of $B$ derived from $A$. Rephrasing, as soon as a value is a landmark (and thus a valid and useful address), dereferences should propagate taint. However, values derived from the landmark have to be modified with tainted data again in order to make the derived value also qualify for pointer taintedness. Thus, we limit the number of times a tainted value can influence the taintedness of a memory access operation.

In practical terms, we say that a value forms a complete and useful address only when it is used as such. In other words, we identify landmarks either by means of a dereference, or by an operation explicitly calculating an address, such as the `lea` instruction on x86 that calculates the effective address and stores it in a register.

**Example** Consider the code snippet shown in Figure 3.6. We access the second item of a list rooted at `table[index]`, where the index is assumed to be tainted. First, in line 7, the pointer to the head of the list is fetched from the table. To calculate the memory load address, `(table + index * 8)`, we use a tainted operand, `index`, which has never been dereferenced before, and so `i1` becomes tainted. However, we have just reached the landmark for `i1`, meaning that dereferences of `i1` propagate taintedness, but addresses derived from `i1` in a clean way do not. Next, in the second assignment, line 8, we access memory at the address calculated by increasing `i1` by `4`, a clean constant. Thus `(i1 + 4)` when dereferenced does not propagate taintedness, and `i2` is clean. A similar reasoning holds for clean `i3`. Based on this example one might think that landmarking solves our problems, as we propagate the taintedness to the elements of a (translation) table, but we do not spread it over adjacent elements of a list.

**Problems with landmarking** Unfortunately, landmarking is not just a rather elaborate technique, it also cripples the power of pointer tainting and opportunities for...
false negatives abound.

Assume that \( p \) is a pointer whose calculation involves a tainted operand, and we load values \( v0 = *p \) and \( v1 = *(p+1) \) from memory. In a first possible scenario, the compiler translates the code such that \( p \) is calculated once for both load operations. In that case, the first of the loaded variables becomes tainted, and the other one is clean. So, depending on the order of instructions, \( v0 = *p; v1 = *(p+1); v0 = *p; v1 = *(p+1); \) vs. \( v1 = *(p+1); v0 = *p; \), we get different results. This is strange. On the other hand, if the compiler translates the code such that \( p \) is calculated twice, once for each of the variables, then both values are tainted. Such inconsistent behaviour makes it hard to draw conclusions based on the results of landmarked pointer tainting. Moreover, it clearly introduces false negatives.

Another example of false negatives stems from translation tables containing structures instead of single fields. Let’s refer to Figure 3.6 once more, where \( i1 \) (line 7), is tainted, but \( i3 = i1->i \) (line 9), is clean. Now imagine that the test_t structure contains various representations of characters, say ASCII and EBCDIC. In that case, the table access makes us lose track of sensitive data, which is clearly undesirable.

This weakness can also be exploited to cause leakage of secret data. Assume that a server receives a string-based request from the network and returns a field from a struct \( X \), pointed to by \( xptr \). If an attacker is able to modify \( xptr \) (for instance, by overflowing the request buffer with a long string), then the server returns the contents of \( xptr->field_offset \) which can point to an arbitrary place in the memory. For the same reasons as in the example above, the result will be clean.

The best thing about landmarking is that we contain the spread of taint very aggressively and it really is much harder for taint to leak out. The hope is that, in combination with EBP/ESP protection, landmarking can stop the pollution, so that (an admittedly reduced version of) FPT can be used for automatic detection of keyloggers.

The worst thing about landmarking is that it does not work. It still offers ample opportunities for false positives. This is no surprise, because even if we restrict taint propagation via pointer tainting in one register, nothing prevents one from copying the register, perhaps even before the dereference, and adding a constant value to the new register. As the new register was not yet used for dereferencing, taint will be (incorrectly) propagated.

Another possible reason for false positives arises when programs calculate directly the address of an element (or field within a struct), without going through an immediate pointer. Consider an array \( A \) of struct \( \{ \text{int} a; \text{int} b; \} \) and assume the index is tainted. If we first calculate the address of \( A[index] \) and use this to calculate the address of the field \( b \), everything will be fine. No taint is unduly propagated to \( b \). However, if we directly calculate a pointer to \( b \) (e.g., \( \text{int } *p = (\text{char }*)A+8*\text{index}+4; \)), we would propagate taint incorrectly to ‘b’. The array example is very simplistic and perhaps a bit contrived but the same (very real) problem may hold for queues, stacks, and hashtables.
We implemented landmarking and analysed the spread of taint when applying it together with EBP/ESP protection. The results are shown in Figure 3.7. Taint pollution takes considerably longer than with just EBP/ESP protection. Some of the processes that receive taint early (like the Xorg X server, or the screensaver), conceivably should have access to the tainted bytes. However, after some time taint again spreads to completely unrelated user processes (e.g., tar, nautilus, hald, python, apt-get, etc.), as well as to the kernel and kernel threads. The results for Windows (Figure 3.8) are even worse. Notice that all processes are occasionally tainted. The calc.exe process, for instance, should not get any taint at all, as we provide input using mouse, and not keyboard. wuaucld.exe is the AutoUpdate Client of Windows Update and is used to check for available updates from Microsoft Update, and thus it is not expected to process keyboard events either.

### 3.3.8 How Bad Are Things?

We conclude with an overall assessment of FPT and LPT.

**FPT on current hardware is fundamentally broken**

We have discussed a few solutions to contain the spurious taint propagation, but they are prone to false negatives, and only slow down the outbreak of false positives. However, the problem is even more serious as there are undecidable cases when the (most common) hardware itself is not able to firmly establish the taintedness of a
Figure 3.8: Taintedness of processes constituting 95% of all context switches in Windows XP with landmarking and EBP/ESP protection.

load operation. For instance, it may be impossible to distinguish accesses to translation tables from accesses to next fields in a linked list. We discuss such a scenario below and argue that without an external oracle (like a priori annotated translation tables or support from a source code analyser) we are not able to successfully apply FPT. We believe that on current hardware, FPT with current containment techniques is not feasible.

Assume that kbd_data is a tainted keystroke and that an application needs to find a lower case value associated with that keystroke. Clearly, we want to propagate taintedness to this derived value. The following code (similar to the GNU C library (glibc-2.7) to_lower() function) is used to obtain the lower case value:

```
[1] attributes = transl_table[kbd_data];
[2] lower_case = attributes->lower;
```

In line 1, in order to get the address of attributes, a non-tainted pointer (transl_table) is combined with a tainted index, kbd_data and the pointer is dereferenced. In other words, attributes is tainted. In line 2, this new tainted pointer is updated with a constant, and dereferenced to load the lower_case value. Unfortunately, taint is not propagated.

At the same time there are numerous cases where the emulator/hardware is executing a similar sequence of instructions where results should not be tainted. Consider the following code, which we already presented in Section 3.3.4

```
[1] struct hlist_head *head = d_hash(parent, hash);
[2] struct dentry *dentry = head->first;
[3] /* (...) */
[4] dentry = dentry->next;
```
3.3. EVALUATING THE PRACTICALITY OF pointer tainting

Line 1 again combines a non-tainted pointer (the address of the hashtable) with a tainted index derived from hash. Next, in line 2 (and in line 4), the newly derived tainted pointer, head (or dentry in line 4), is again altered using a constant index to further fetch a next entry from the list. An equivalent sequence of instructions as in the previous example should now not propagate taint. Since we have argued that black/white listing cannot solve the problem fully, false positives would be unavoidable.

We do not believe that hardware can distinguish the two cases without an external oracle. Of course, we may throw more heuristics at the problem. For instance, we could apply landmarking such that it allows for dereferences of one level deep, so that while dentry is tainted, a dereference does not propagate taint. However, the opportunity for false positives would increase. Moreover, since we may have structures with multiple levels of nesting, it is hard to see where we should draw the line.

Challenges for LPT on popular hardware

As we saw in Section 3.3.6-C, pointer injection detection seems to be a promising technique for containing taint propagation. If we can get it to work on commonly used hardware, LPT may be a powerful technique for detecting attacks against non-control data. Unfortunately, it seems impossible to achieve this unless we are able to recognise existing system pointers reliably. Raksha [74] shows that this is possible for Linux on the SPARC by dint of architectural ‘features’ (e.g., two specific instructions are used to form a pointer).

It is an open challenge to do something similar on x86-like architectures, or to invent completely new techniques for containment of taint. Meeting this challenge means that we salvage an important technique for detecting non-control data attacks. Until that time, we think pointer tainting has serious problems that prevent it from being used in real systems.

3.3.9 Other Pointer Tainting Approaches

In Section 3.2.2, we saw a lot of popular approaches to basic tainting. Now, we discuss other projects employing pointer tainting.

The technique of pointer tainting for non-control data attack detection was formally introduced by Chen et al. [43], and later evaluated in a hardware design [44; 45]. Any dereference of tainted pointer triggers an alert and the technique is evaluated with a number of synthetic, and some real attacks, and six SPEC2000 applications. No false positives are reported.

Dalton et al. [72] point out that naive propagation rules that trigger an alert when a pointer dereference has any of its source operands tainted are problematic. Indexing an array with a tainted index in many cases need not be unsafe if it was properly bounds-checked. The proposed solution is to propagate only the taint bit of the base
pointer. While DIFT [196] optimistically assumes that bounds checking is always done by the applications, this is clearly not true in practice. The authors suggest to apply untainting only for instructions dedicated to input validation which requires an architecture that can distinguish such instructions.

Raksha [73] is yet another hardware (FPGA-based) approach with support for pointer tainting. However, the system is quite flexible and able to handle various DIFT models besides pointer tainting.

Panorama [219] differs from the previous projects in that it is designed solely to check whether sensitive data leaks into software that may or may not be malicious. All data from sensitive source is marked tainted and the potential malware is run in a system under pointer tainting. The program under scrutiny is considered suspicious if it processes tainted data. Egele et al. [82] propose a very similar approach, but they focus on detecting spyware in a web browser. HookFinder [218] wants to determine whether a piece of malicious code has implanted a hook in the OS. Dynamic taint analysis, including full pointer tainting, is used to track what they refer to as impacts on the OS made by the untrusted software, and checking whether they exhibit a desired hooking behaviour. An alert is triggered whenever tainted data leaks to the program under scrutiny. While the authors did not observe false positives, we suspect that this may be because they ran the code only for short periods of time.

Xu et al. [216] present a dynamic taint analysis technique to detect input validation attacks, implemented as a source-to-source transformation. For fear of false positives the authors do not track dereferences, only the direct array access is supported.

3.3.10 Conclusions

We have analysed pointer tainting, considered one of the most powerful techniques to detect keyloggers and memory corruption attacks on non-control data. Both in the analysis and in experiments, the method proved problematic due to the large number of false positives, even when we apply all methods that we know of for containing the spread of taint. We argued that full pointer tainting is probably not suited for detecting privacy-breaching malware like keyloggers. Moreover, it is even unclear whether limited pointer tainting can be applied to detect automatically memory corruption attacks on the most popular PC architecture (x86) and the most popular OS (Windows).
Chapter 4

Prospector: Buffer Overflows
Analysis

4.1 Introduction

Polymorphic network attacks are difficult to detect and even harder to fingerprint and stop. This is especially true if the exploit itself is polymorphic (refer to Fogla et al. [92]). We define fingerprinting as the process of finding out how an attack works, i.e., what an attacker should do to make the exploit succeed. It is important for two reasons: analysis of the attack (e.g., by human security experts), and signature generation.

Signature generation is hard because of the complex and conflicting list of constraints. Signatures should incur a negligible ratio of false positives, while the number of false negatives should be low. Also, we should be able to check signatures at high rates and cater to polymorphic attacks with polymorphic exploits. We further aim for fast, one-shot generation without the need to replay the attack.

In this chapter, we address the problem of control-diverting polymorphic buffer overflow attacks on heap and stack. (Refer also to Section 2.1.) Given their long history and the wealth of counter-measures, it is perhaps surprising that buffer overflows are still a very popular attack vector. For instance, approximately one fourth of all vulnerabilities notes reported by US-CERT in 2010 consisted of buffer overflows [203]. As the US-CERT’s database contains many different types of vulnerabilities (leading to denial of service, privacy violation, malfunctioning, etc.), the percentage of buffer overflows in the set of vulnerabilities leading to control over the victim is likely to be higher. Even Windows Vista, an OS with overflow protection built into the core of the system, has shown to be vulnerable to such attacks [166].

Polymorphic attacks demand that signature generators take into account properties other than simple byte patterns. For instance, previous approaches have examined such properties as the structure of executable data [116], or anomalies in
process/network behaviour [87; 118; 130].

In contrast, in this work we asked a simple question that is surprisingly hard to answer: what bytes contribute to an attack? As we will see, an answer to this question also trivially yields reliable signatures that meet the requirements listed earlier. Like Brumley et al. [34], we focus on the vulnerabilities rather than specific attacks, which makes the signatures impervious to polymorphism. However, besides signatures, we believe the answer to the above question is invaluable for later analysis by human experts.

The full system is known as Prospector, a protocol-specific detector of polymorphic buffer overflows. It deals with both heap and stack overflows in either the kernel or user processes and while it was implemented and evaluated on Linux, the techniques apply to other OSes also.

In a nutshell, the idea is as follows (see also Figure 4.1). We use Argos [161], an emulator-based honeypot with dynamic taint analysis, to detect attacks and to locate both the exact address where a control flow diversion occurs and all the memory blocks that originate in the network (known as the tainted bytes). For details about Argos refer to Section 3.2.

Next, we track which of the tainted bytes took part in the attack. For instance, in a stack overflow we walk up the stack looking for tainted bytes. However, we must weed out all the memory that, while tainted, had nothing to do with the attack (e.g., stale data that was part of an old stack frame, such as the bytes marked $x$ in the figure). To do so, we track the age of data at runtime, so that we know whether memory on the heap or stack is a leftover from an older allocation. As a result, we can distinguish between relevant bytes and memory to be ignored.

When we recognise the protocol governing the malicious network message, we also generate a signature capable of identifying polymorphic versions of the attack. Once we know which bytes were in the buffer overflow and we can trace them to the bytes that arrived from the network, we find out which protocol fields contributed to the attack. If $n$ fields were involved in the overflow with a combined length of $N$, we know that any similar protocol message with a combined length for these fields greater or equal to $N$ will also lead to a buffer overflow. Using the maximum length of a (single) protocol field as the signature of a polymorphic attack was first proposed in Covers [124]. While Covers may be considered an inspiration for our work, we will see that it suffers from both false positives and false negatives. In this sense our signature generator remedies and extends the Covers technique.

Still, this method is inappropriate for attacks based on messages that contain a specially crafted (wrong) length field, misspecifying the length of another protocol field. As we will see, to detect such attacks, the signature pinpoints the length field and specifies when misbehaviour occurs.

**Contributions.** Our main contribution is the identification of all bytes contributing to an overflow. The identification is performed in a single interaction (i.e., without need for replaying attacks) and is sufficiently fast to be used in honeypots. The signature generator is intended to demonstrate the usefulness of such data in prac-
4.1. INTRODUCTION

While the end result is a powerful signature generator in its own right, very different signature generators could also be built on this technique. For instance, we essentially yield Snort-like patterns which may be used if the attack is not polymorphic. In addition, it could generate a wealth of information for human security experts.

A second contribution is that we extend taint analysis in the temporal domain. In its simplest form, taint analysis is zero-dimensional and consists of a single bit for every data item to indicate whether or not it originates in a suspect source. More advanced analysis extends the analysis in the spatial dimension, by tracking exactly where the data originated (e.g., Vigilante and Argos both maintain a pointer in the network trace). In this paper, we extend tracking in the temporal domain by storing when the data is tainted. We argue that this is essential information for signature generators that allows us to separate relevant bytes from unrelated tainted memory.

A third contribution is that we first show that well-known existing vulnerability-based signatures based on the length of a protocol-field (e.g., Covers [124]) are weak and frequently incur both false positives and false negatives. Then, we remedy the weakness so as to make false positives virtually impossible and false negatives implausible.

A fourth contribution is that we extend the vulnerability signatures to include attacks based on protocol messages that contain a specially forged (wrong) length field. For instance, such fields specify the length of another protocol field and by providing a wrong value, the attack coerces vulnerable programs into allocating buffers that are too small and that overflow when the actual data exceeds the specified length. We will discuss more advanced attacks of this type also. Few existing projects address such attacks.

There are other contributions as well (e.g., a novel way to monitor process
switches from an underlying emulator), but as they are not the focus of this work, we will not dwell on them in this section, and defer the discussion to the relevant sections. Finally, we extended Prospector with an attack vector-specific module to make it deal with double free attacks.

Besides the functionality, of course, one of the main questions concerns performance: is the tracking fast enough to be of practical use? While emulation, taint analysis, and age tracking all incur a fair amount of overhead, we believe that Prospector is well-suited for honeypots. Indeed, the slow-down compared to the ArgoS honeypot on which it is based is less than 20%. For full taint-analysis, Argos is considered a fast emulator, so we believe the overhead is acceptable.

Outline The remainder of this chapter is organised as follows. In Section 4.2, we place our work in the context of related work. Section 4.3 discusses heap and stack overflows and highlights factors that complicate the analysis. Sections 4.4 and 4.5 describe the design and implementation of Prospector, respectively. The system is evaluated in Section 4.6, and conclusions are drawn in Section 4.8.

4.2 Related Work

Recent worms have started using polymorphic engines like ADMmutate [119]. They work by inserting garbage and NOP insertions, and/or by substituting code by equivalent code, shuffling registers, and encryption. While more constrained, even exploits are made polymorphic (refer to Fogla et al. [92]).

Previous work on detection of polymorphic attacks focused on techniques that look for executable code in messages, including: (a) abstract or actual execution of network data in an attempt to determine the maximum executable length of the payload [201], (b) static analysis to detect exploit code [46], (c) sled detection [8], and (d) structural analysis of binaries to find similarities between worm instances [116].

Taint-analysis has been used in several projects for signature generation [149; 62]. For accurate signature generation, replaying of the attack is required, which we believe is not trivial [67]. Also, the level of polymorphism that can be handled is limited. However, none of the existing projects provide an answer to the question of which bytes were involved. Enhanced tainting [216] expands the scope of tainting to also detect such attacks as SQL injection and XSS, but requires source code transformation.

Transport-layer filters independent of exploit code are proposed in Shield [207] with signatures in the form of partial state machines modelling the vulnerability. Specific protection against instruction and register shuffling, as well as against garbage insertion is offered by semantics-aware detection [51].

A related project, PolyGraph [148], fingerprints attacks by looking at invariant substrings present in different instances of suspicious traffic. The idea is to use these
4.3 Attacks and Factors Complicating the Analysis

Prospector caters to both heap and stack overflows. Stack overflows are conceptually simple. Even so, they prove to be hard to analyse automatically. Essentially, a vulnerable buffer on the stack is overflown with network data until it overwrites a target that may lead to control flow diversion (typically the return address). An
void read_from_socket(int fd) { // fd is the socket descriptor
    int n;
    char vuln_buf[8]; // the vulnerable buffer
    char unrelated[8]; // a safe buffer, unrelated to the attack
    read(vuln_buf, fd, 32); // from socket: taints all data in 'vuln_buf'
    // and above (overflow possible)
    read(unrelated, fd, 8); // from socket: taints all data in 'unrelated' (no overflow possible)
    n = 1; // untaints 4 bytes of data that was previously untainted, creating a gap
    return;
}

Figure 4.2: Tainted data: gaps and dirt (unrelated tainted data).

important observation here is that the data that is used for the overflow may originate in more than one set of bytes in the network flow (examples in practice include the well-known Apache-Knacker exploit [181]). In Figure 4.1 this is illustrated by regions b1 and b2. Taking into account either fewer or more protocol fields may lead both to false positives and negatives. Covers [124], by using a single protocol field, therefore lacks accuracy in a multi-field attack.

There is another, more subtle reason why this may occur, even if the attack does not use multiple fields: the protocol dissector used to generate signatures may work at different protocol field granularities than the application. For instance, the dissector may identify subfields in a record-like protocol field as separate fields, while the application simply treats it as a single protocol field. As a consequence, the two types of misclassification described above may occur even for ‘single-field’ exploits. As we often do not have detailed information about the application, this scenario is quite likely. Again, solving the problem requires handling ‘multi-field’ attacks properly.

Gaps. The naive solution for finding the bytes that contribute to the attack is to start at the point of attack (the target in Figure 4.1) and grab every tainted byte below that address until we hit a non-tainted byte. Unfortunately, while all bytes that contributed to the attack were tainted at some point, such a naive solution is really not adequate. First, there may be gaps in the tainted block of memory that was used in the attack. For instance, the code in Figure 4.2 may lead to a gap, because the assignment to n occurs after the overflow.

Unrelated taints. Second, the naive solution gathers tainted blocks that are unrelated to the attack. An example is the region marked by x in Figure 4.1. It may be caused by tainted data left-over from an old stack frame, or by safe buffers adjacent to the vulnerable buffer, such as the buffer unrelated in Figure 4.2. In this chapter, we will informally refer to such unrelated tainted data as unrelated taints.

Heap corruption. As we discussed in Chapter 2, heap corruption can be more complex than a stack overflow and potentially more powerful. For instance, heap overflows were among the first techniques to work around stack protection techniques like StackGuard [63]. A heap corruption attack can take two main forms. A simple overflow occurs when critical data (e.g., a function pointer) is overwritten
from a neighbouring chunk of memory, or from another field of a structure. In a more advanced form, the attacker overflows link pointers that are used to maintain a structure keeping free regions. It allows an attacker to overwrite virtually any memory location with any data [12]. The problem is caused by the implementation of memory allocation functions which store control data together with the actual allocated memory, thus providing attackers potential access to information used by the operating system memory management.

The problem of gaps and unrelated taints also exists for heaps and is mostly similar to that of the stack. For heap overflows, instead of the occurrence of stale tainted data from a previous function call, we may encounter stale tainted data used in a previous function that allocated the memory region. In addition, there may be taints in adjacent fields of a structure. Advanced heap corruption attacks yield an additional complication. Since the attacker can overwrite any memory location with any contents, it is possible that at detection time the memory region which was holding the vulnerable buffer is reused and contains unrelated data. If left unhandled, such a scenario would prevent us from pin-pointing exactly the data responsible for the intrusion attempt.

Length field attacks. Finally, numerous protocols have fields specifying the length of another field, say $l_f$ defining the length of field $f$. Attackers may manipulate this length value, and via heap overflows take control of the host. First, a malicious message may provide $l_1$ instead of $l_f$, with $l_1 \gg l_f$ and close to the maximum size of an integer. The application allocates $l = l_1 + k$ bytes (where $k$ bytes are needed to store some application-specific data), and $l$ ends up being a small number because of the integer wrap-around, $l \ll l_1$. As a result, the application copies $l_1$ bytes into the buffer leading to overflow. In the second scenario, rarely seen in the wild, the attacker provides $l_2$ smaller than expected, $l_2 < l_f$, the application allocates a buffer of size $l_2$ which is not sufficient to hold the data, and a subsequent copy operation without boundary checks spills network data over adjacent memory. Notice that we cannot draw any conclusions about a message containing such attacks by relying only on the observation that $n$ fields where involved in the overflow with a combined length of $N$.

We conclude this section with an assumption. We assume that overflows occur by writing bytes beyond the high end of the buffer. While not strictly necessary, it makes the explanation of our work easier. However, the techniques described in this paper can be trivially extended to handle the reverse direction also (i.e., attacks that overwrite memory below the start of a vulnerable buffer).

### 4.4 Design

The main steps of Prospector’s attack analysis are sketched in Figure 4.1. In this section, we first describe how we instrument the execution and what data is produced by our taint-analysis emulator (Sections 4.4.1–4.4.6). From this data we formally
derive a set of properties of tainted regions in Section 4.4.7. We then show in Section 4.4.8 how we use these properties to determine the exact bytes in the attack. The memory that constitutes these bytes will be referred to as the crucial region. Finally, we correlate the information with protocol fields in network data to obtain signatures.

### 4.4.1 Argos

Prospector uses Argos [161] to detect attacks. As we explained in Section 3.2, Argos is an efficient and reliable hardware emulator that uses basic dynamic taint analysis [149] to tag and track network data. Data originating in the network is marked as tainted, and whenever it is copied to memory or registers, the new location is tainted also. Argos raises an alert whenever the use of such data violates security policies. To aid signature generation, Argos dumps the contents of all registers, as well as tainted memory blocks to a file, with markers specifying the address that triggered the violation, the memory area it was pointing to, etc. In addition, Argos allows us to keep track of the exact origin of a tainted memory area, in the form of an offset from the start of the network trace. In practice, the offset is used as a (32 bit) tag.

Even with such accurate administration of offsets, the problem of identifying crucial regions remains. We therefore extended the tracking in the temporal domain. In the next few sections we will explain the blocks that together form our information correlation engine. We start with support for an advanced heap corruption attack, and then explain how we pinpoint the relevant tainted memory region.

### 4.4.2 Dealing with Advanced Heap Overflows

In the case of stack overflows and simple heap corruption attacks, we know from where to look for the crucial regions: in the memory area beneath the violation address reported by Argos. In contrast, advanced heap corruption attacks require us to find first the memory region containing the vulnerable buffer. Only then can we start marking the bytes that contributed to the attack.

As mentioned earlier, such attacks may easily lead to a situation in which at detection time, the memory region that was holding the vulnerable buffer is reused and contains unrelated data. To deal with this scenario, the emulator marks the bytes surrounding an allocated chunk of memory as red. If tainted data is written to a red region, indicating an overflow (although not necessarily an attack, see also Section 4.5.2), we keep the application running, but we dump the memory region covering the whole vulnerable buffer for potential later use, i.e., we traverse down the heap, storing data until we come across a red marker indicating the beginning of the vulnerable buffer. This works as common memory management systems store control data in-line together with allocated chunks of memory. Consequently the ‘red’ bytes surrounding an allocated buffer contain control data, which should never be overwritten with data coming from the network.
4.4. DESIGN

In the case of an intrusion attempt, we search for the violation address and the network index in the dumped heap areas in order to find a memory region containing the buffer that contributed to the attack. These chunks of memory allow us to perform further analysis to overcome all the difficulties described in Section 4.4.8. The red markers are vaguely reminiscent of StackGuard’s canary values [63], but different in that they are maintained by the emulator and trigger action immediately when they are overwritten.

4.4.3 Dealing with Malformed Messages in Heap Overflows

To handle heap corruption attacks that use malformed length fields, we check whether allocating a chunk of memory relies on remote data. Whenever an application calls `malloc(size)` with the `size` variable being tainted, we associate the network origins of the length parameter with the new memory chunk. In the case of an intrusion attempt, it enables us to understand the reasons for failure, and generate a correct signature. (Refer to Section 4.4.9.)

4.4.4 Age Stamps

In order to distinguish between stale and relevant data both on stack and heap we introduce an age stamp indicating the relative age of data regions. First we will describe when the age stamp is used and updated on the Argos side (in the runtime), and after that we will sketch the way it supports the analysis part that is performed only in the case of an intrusion attempt. AgeStamp is a global counter, common to the entire OS running on Argos. The need for a system-wide global variable stems from the fact that memory may be shared. AgeStamp is increased whenever a function is called (a new stack frame is allocated) or returns. To be precise, we update AgeStamp \( v_1 \) to \((v_1 + 1)\) only if in epoch \( v_1 \) a tainted value was stored in the memory. Otherwise it is not necessary, as we shall soon see. If a tainted value is copied to memory, we associate the current AgeStamp with the destination memory location, i.e., for each tainted value we remember the epoch in which it was stored. In addition, for each process and lightweight process we allocate a history buffer, where we store information about allocation and release of stack frames, as follows: for each function call and return we store the value pair (stack pointer, AgeStamp). When an application allocates a buffer on the heap, we associate the current AgeStamp with this memory region. When a memory field becomes untainted, we do not clean the age stamp value.

We observe that the order of age stamps in the crucial region right after the overflow (before gaps appear) is nondecreasing. We will use this observation in the analysis phase in Section 4.4.8 to spot tainted bytes stored later than the crucial tainted memory region, and so forming a gap, or an area of unrelated taints. For instance, the unrelated buffer in Figure 4.2 has the corresponding age stamps bigger than `vuln_buf`, and so we can conclude that it does not belong to the crucial tainted
memory region.

The explanation of the observation is straightforward. If the buffer was overwritten with one call to a copying function, all tainted bytes have the same age stamp, and so the observation holds. Otherwise, the observation results from the assumption that buffers overflow from low to high addresses. Indeed, if the lower part of the buffer was filled by a function $\text{fun}_1$ in the time period $\text{AgeStamp}_1$, and later on the higher part - by a function $\text{fun}_2$ in the period $\text{AgeStamp}_2$, then $\text{AgeStamp}_1 < \text{AgeStamp}_2$.

### 4.4.5 Additional Indicators

Even though age stamps provide a crude separation of unrelated taints, they are not powerful enough. Let us consider an example vulnerable $\text{a_fun}$ function in Figure 4.3. For simplicity we discuss a stack example, (i.e., the target is a return address) but the method is used for the heap also. The figure illustrates a series ($a$–$f$) of the emulator’s memory maps presenting values associated with local variables of $\text{a_fun}$. For now, we limit our interest to the first two columns containing information about taintedness and $\text{AgeStamp}$, respectively. We assume that the example function is executed in an epoch with $\text{AgeStamp}$ equal to 20, so that the few existing tainted bytes with $\text{AgeStamp}$ 19 are stale. Note that by using while and for loops, $\text{a_fun}$ copies network data without any calls and thus without incrementing $\text{AgeStamp}$.

Even though we duly raise an alert after step 4.3f, when the function returns and is about to jump to an address influenced by the attacker, the memory dump and the age stamps do not provide the means to separate the relevant bytes from the unrelated buffer $\text{buf}$. The reason is that $\text{vuln_buf}$ and $\text{buf}$ have the same $\text{AgeStamp}$.

To remedy this situation we introduce two extra 1-bit indicators for each memory location to let us establish the order in which the buffers were filled: $\text{PFT}$ (Previous address Freshly Tainted) and $\text{FTS}$ (First Tainted Store), respectively. Intuitively, $\text{PFT}$ indicates for address $a$ whether $a - 1$ was assigned fresh tainted contents. If $a$ is tainted, then $\text{PFT}$ signifies that the contents of $a - 1$ is more recent than that of $a$. The $\text{FTS}$ bit indicates that the tainted store at address $a$ was the first such store to $a$ after $a - 1$ was tainted. However, their exact meanings are defined by the algorithm in Figure 4.4. As the semantics of these two additional indicators are complex, we introduce them by way of a detailed example.

### 4.4.6 Example Explained

We return to the example in Figure 4.3 and examine values of $\text{PFT}$ and $\text{FTS}$, i.e., the values in the last two columns of the memory maps.

The assignment operation in line 5 sets memory associated with $p$ as untainted, and leaves Prospector’s markers untouched (Figure 4.3b).

This brings us to the execution of the while loop in lines 6-7. The first iteration marks $\text{addr_vuln_buf}$ tainted, sets $\text{AgeStamp}(\text{addr_vuln_buf})$ to the current
4.4. DESIGN

value of AgeStamp, and PFT(addr\textsubscript{vuln} buf+1) to 1. We informally interpret it as addr\textsubscript{vuln} buf telling (addr\textsubscript{vuln} buf+1): “I have tainted contents, more fresh than yours”. We still need to decide about FTS(addr\textsubscript{vuln} buf). As we do not know the value of PFT(addr\textsubscript{vuln} buf), let us assume, for example, that PFT is unset. In this case, addr\textsubscript{vuln} buf has already ‘consumed the message’ from (addr\textsubscript{vuln} buf-1), and so the current store operation is not the first since (addr\textsubscript{vuln} buf-1) became tainted. We record this information by unsetting FTS(addr\textsubscript{vuln} buf).

Figure 4.3c presents the second iteration of the while loop in lines 6-7. We mark addr\textsubscript{vuln} buf+1 as tainted, set AgeStamp(addr\textsubscript{vuln} buf+1) to the current value of AgeStamp, and PFT(addr\textsubscript{vuln} buf+2) to 1, thus informing the memory location above it that addr\textsubscript{vuln} buf+1 has freshly tainted contents. This time we know that no tainted store operation was executed since addr\textsubscript{vuln} buf became tainted. We set FTS(addr\textsubscript{vuln} buf+1) to 1, and also unset PFT(addr\textsubscript{vuln} buf+1), since the tainted value of addr\textsubscript{vuln} buf+1 is more recent than that of addr\textsubscript{vuln} buf.

Figure 4.3d illustrates the memory map just after the while loop. Observe that all bytes inside the tainted memory region which contributed to the attack have PFT unset, and FTS set to 1.

This brings us to the for loop in lines 8-9, the first iteration of which can be examined in Figure 4.3e. While storing the first byte in the gap, we set PFT(addr\textsubscript{buf}+1) to 1, and also check that the current store operation is not the first one since addr\textsubscript{buf}+1 became tainted. Indeed, the assignment in the fifth iteration of the while loop held this property. So, we unset FTS(addr\textsubscript{buf}).

Finally, Figure 4.3f presents the whole gap formed by buf. Observe that the gap internally has PFT negated, and FTS set, just like a ‘typical’ tainted region. However, the byte just above the gap has PFT set to 1, as a result of the store in the fourth iter-
Figure 4.4: Algorithm for updating the indicators.

ation of the for loop. In that iteration, (addr(buf+3) informed the memory location above it about its freshly tainted contents. Since this was the last byte of the unrelated buffer, no store operation has 'consumed this message'. Similarly, the bottom byte of the gap has both indicators negated.

Now that we have an intuitive grasp of the use of the additional indicators, we are ready to turn to more formal definitions (Section 4.4.7) and analysis (Section 4.4.8).

### 4.4.7 Formal Specification of the Properties of Tainted Data and Gaps

In this section, we use the indicators defined above to derive properties of regions of tainted memory.

**Observation 1** Let buf be a crucial tainted region of size $n$. Then:

(a) \( \forall i = 0 \ldots (n-1): \text{buf}(i) \) is tainted,

(b) \( \forall i = 0 \ldots (n-1): \text{AgeStamp} \geq \text{AllocAgeStamp} \), where AllocAgeStamp is the epoch in which the buffer was allocated,

(c) \( \forall i, j = 0 \ldots (n-1), i < j: \text{AgeStamp} \leq \text{AgeStamp} \),

(d) \( \forall i = 1 \ldots (n-1): \text{PFT}(\text{buf}(i)) \) is unset, and \( \text{FTS}(\text{buf}(i)) \) is set (as the store at \( \text{buf}(i) \) finds PFT set).

**Observation 2** Let gap be a non-tainted discontinuity located inside a crucial tainted memory region buf, i.e., a region in buf where Observation 1.a does not hold.
Since neither age stamps nor indicators are changed when a memory location becomes untainted, Observations 1.b–1.d also hold within gap.

**Observation 3** Let gap be a tainted discontinuity of size $m$ inside a crucial tainted memory region buf. Then:

(a) $\forall i = 0 \ldots (m - 1)$ gap[$i$] is tainted,

(b) $\forall i = 0 \ldots (m - 1)$ AgeStamp(gap[$i$]) $\geq$ AgeStamp(gap[$m$]).\(^1\)

(c) gap[$m$] has both indicators PFT and FTS set to 1, while gap[0] has both indicators set to 0.

(d) $\forall i = 1 \ldots (m - 1)$: PFT(gap[$i$]) is unset, and FTS(gap[$i$]) is set.

Of course, a gap containing unrelated taint may adjoin a similar gap. In that case, they simply merge as follows. If gap is a tainted discontinuity located inside a crucial region buf, and the bottom (top) part of gap adjoins another tainted discontinuity gap_b (gap_t), then both holes merge together forming a single discontinuity for which all properties listed in Observation 3 hold.

### 4.4.8 Analysis

To find the bytes that contributed to the attack (the crucial region), we traverse the memory downwards starting at the violation address and continue as long as the bytes we come across conform to Observation 1. In this section we discuss how to start this process and how to overcome the complicating factors mentioned in Section 4.3.

#### The age of allocation

We start the analysis by figuring out AllocAgeStamp, the age (or epoch) in which the vulnerable buffer containing the violation address was allocated. We need it to distinguish between fresh and stale data.

In the case of a heap corruption attack, the age stamp was explicitly maintained for each chunk of memory. In a stack smashing attack (i.e., when the violation address is not smaller than the value of the stack pointer register ESP), we check the history of stack frames associated with the vulnerable process for the most recent entry above the violation address. If the malicious data was spilled over the adjacent stack frame as well, we may find an age stamp of a caller function instead. However this does not prevent the correct analysis, because when we start looking for the whole crucial region later, we will figure out the most recent and proper AgeStamp.

---

\(^1\)While gap has only $m$ bytes and gap[$m$] strictly speaking does not exist, we use it as a C-like shorthand for ‘the byte above gap’.
Gaps

One of the difficulties identified in Section 4.3 concerned gaps in the crucial region’s tainted data. Such discontinuities occur for instance when the program assigns a new value to a local variable allocated in the crucial region after the overflow took place. They can also arise if the whole or parts of the vulnerable buffer are refilled by the application. Let us assume for now that the discontinuity is fully included in the crucial tainted memory region, i.e., it does not reach the bottom. In other words, below the gap there is at least one byte which contributed to the attack.

Again, to find the crucial region we traverse the memory as long as the bytes encountered are in accordance with Observation 1. However, we now come across a discontinuity before we reach the region’s bottom. To handle such gaps, we look for the end of the discontinuity to find out how many bytes of the crucial region we are missing. In the following, assume that $addr_1$ is a memory location at variance with at least one of the properties of Observation 1.

If we find a byte at variance only with Observation 1.a (i.e., it is not tainted), we conclude that it belongs to a non-tainted discontinuity. We traverse the memory further until we encounter tainted data. Since we assume that the gap does not reach the beginning of the vulnerable buffer, we will eventually spot a tainted byte. In the worst case, this will be $buf[0]$.

We can also find a byte in memory location $addr_1$ at variance with Observation 1.d. This means that the values of indicators are corrupted: $PFT(addr_1)$ is not equal to 0 and/or $FTS(addr_1)$ is not equal to 1. If both indicators are set to one, then the memory location below has freshly tainted contents. Observation 3, defining gaps, says that it is probable that we have just spotted a tainted discontinuity. We now traverse the memory until we encounter a memory location with the two indicators not set. Let us now assume that the inconsistency with Observation 1.d means that $FTS(addr_1)$ is equal to 0. At first sight, one may think that a new tainted store operation at $addr_1$ caused the change of the indicator, but then the memory location above it ($addr_1 + 1$) would have $PFT$ set to 1, which would also have conflicted with Observation 1.d. As we did not detect this, such a case will not occur.

Similar reasoning yields that we will never discover the top of a gap by coming across a byte with an AgeStamp more recent than expected (i.e., at variance with Observation 1.c). Indeed, a tainted store operation at $addr_1$ changes $PFT(addr_1 + 1)$, which we will encounter first.

Summarising, we know how to detect boundaries of a discontinuity established in a crucial memory region. Note that without the extra indicators we would not be able to identify the tainted gap established in the same age as the remains of the vulnerable buffer.

Excess of data

We now discuss how to determine the beginning of the vulnerable buffer $buf$, thus we address the problem of unrelated taints. For the sake of simplicity, we again
assume that there is no discontinuity at the beginning of \texttt{buf}. It means that at the point of intrusion detection, \texttt{buf[0]} contains the byte that contributed to the attack. By traversing the memory we eventually encounter this byte.

Consider the successive possible instances of the beginning of the vulnerable buffer. For each of the scenarios we explicitly discuss the contents of essential variables at the time of the overflow and at the time of detecting the intrusion. For the sake of clarity let us denote the memory location of \texttt{buf[0]} by \texttt{addr}_B, and the address below \texttt{buf[0]} by \texttt{addr}_A. We assume that traversing the memory as discussed above led us to byte \texttt{addr}_B, and we check whether appropriate conclusions enabling to spot correctly the buffer boundary can be drawn.

1. **Overflow:** \texttt{PFT(addr}_B\texttt{)} equals 0 (then \texttt{addr}_A is untainted); we set \texttt{FTS(addr}_B\texttt{)} = 0.

   **Detection:** we encounter a byte with \texttt{FTS} set to 0, and we are not inside a discontinuity. We conclude that at the time of the overflow \texttt{addr}_A was untainted. Thus we have just encountered the beginning of \texttt{buf}. To make the conclusion clear, note that inside a tainted vulnerable buffer there is only one possibility for a byte to have the \texttt{FTS} indicator unset, namely at the beginning of a gap (Observations 1 and 3.)

2. **Overflow:** \texttt{PFT(addr}_B\texttt{)} is equal to 1, but \texttt{addr}_A contains stale tainted data; we set \texttt{FTS(addr}_B\texttt{)} and unset \texttt{PFT(addr}_B\texttt{)}.

   **Detection:** We encounter a byte with \texttt{FTS} set and \texttt{PFT} unset, which has the stored age stamp of the address beneath it. Observe that since the data at \texttt{addr}_A is stale, \texttt{AgeStampaddr}_A is less than the current age stamp, and recall from Figure 4.4, step i.3 that we will have stored the age stamp in this case. We compare this age stamp with \texttt{AllocAgeStamp} of \texttt{buf} to conclude that at the time of overflow \texttt{addr}_A’s value was stale, so we have just encountered the beginning of the vulnerable buffer.

3. **Overflow:** \texttt{PFT(addr}_B\texttt{)} is equal to 1, \texttt{addr}_A contains fresh data; we set \texttt{FTS(addr}_A\texttt{)} and unset \texttt{PFT(addr}_B\texttt{)}. Since \texttt{addr}_A merged with \texttt{buf} together form an area that conforms to all the properties of a crucial region (see Observation 1), we will treat \texttt{addr}_A as a part of the tainted buffer we are looking for. Note that we cannot detect that \texttt{addr}_A belongs to a distinct variable. Most compilers (including \texttt{gcc}) allocate stack memory for a few local variables at once, making it impossible to see the boundaries between successive buffers. Similarly, on the heap, memory is allocated for a structure as a whole, rather than for the individual fields separately.

   **Detection:** We come across a byte with \texttt{FTS} set to 1. Regardless of the existence of the stored age stamp of the memory location below it, we will conclude that at the moment of overflow \texttt{addr}_A’s value was fresh, and so is supposed to belong to the vulnerable buffer. We will simply go on with the analysis looking for gaps and end of the buffer as if nothing had happened.
Depending on the application behaviour between the moment of overflow and that of detection, we will end up either adding unrelated taints to the crucial tainted memory region or spotting a contradiction with Observations 1-3 and reversing to the last correct byte encountered, $addr_B$. The first possibility comes true only if (a) we reach a buffer that is totally filled with network data, (b) the possible area between this buffer and $addr_B$ appears exactly like an unrelated tainted gap, and (c) additionally, the whole region containing the buffer, the unrelated tainted gap, and the crucial tainted memory region is in accordance with Observations 1-3. Note however, that even in this unlikely case we could only incur false negatives, and never false positives, since the unrelated tainted buffer needs to be filled totally.

We have not discussed what happens if the discontinuity in the vulnerable buffer reaches the buffer’s bottom. In principle, the analysis is analogous to the one presented above. What is worth noting, is the fact that we cannot determine the length of the discontinuity and we may miss part of the crucial tainted memory region, since the bottom part of the vulnerable buffer gets overwritten.

### 4.4.9 Signature Generation

Whenever we recognise the protocol governing the malicious network message, we generate a signature capable of detecting polymorphic variants of the attack. After the preceding steps have identified the malicious data in memory and generated a one-to-one mapping with bytes in the network trace, we first list the protocol fields including the crucial tainted memory region. Due to possible excess of tainted data in rare scenarios described in Section 4.4.8, we include a protocol field in a signature either if it contains the violation address, or if a cohesive part of it including at least one boundary can be mapped to the indicated malicious data. We call these fields critical.

Note that vulnerable code usually handles specific protocol fields. Thus, attackers wishing to exploit a certain vulnerability within this code, embed the attack in these protocol fields (or sets of protocol fields in the case of exploits like Apache-Knacker [181]). If values in such fields contain more bytes than can be accommodated by the buffer, an overflow is sure to occur.

**Vulnerabilities rather than attacks**

We generate signatures for stack and heap overflows by specifying the vulnerability rather than the attack itself. We do so by indicating the protocol fields that should collectively satisfy a condition. In particular, in the current version the signature specifies that the fields should collectively have a length $L$ that does not exceed some maximum, lest they overflow important values in memory (e.g., a function pointer on the heap, or the return address of a function). In the simple case with
only one protocol field responsible for the attack, \( L \) describes the distance between the beginning of the protocol field and the position in the network trace that contains the value that overwrites the target. Otherwise, \( L \) is augmented with the lengths of the remaining critical fields. In both cases \( L \) is greater or equal to the length of the vulnerable buffer. Signatures can be checked by a protocol dissector (similar to Ethereal) that yields the fields in a flow.

**Heap overflows founded on malformed length.** As mentioned earlier, signatures for heap corruption attempts that manipulate a length field need to relate the critical fields to the length field. Thus, after having determined the crucial tainted memory region \( \text{buf} \) of length \( l \), we check in the network trace for the length value provided by the attacker \( l_a \). If it is bigger than \( l \), we specify that a message contains an attack if the cumulative length of the critical fields is smaller then \( l_a \) with the length field greater or equal \( l_a \). In the second scenario, with \( l_a < l \), we must be more cautious, since the value provided by the attacker does not need to define the number of bytes, but it could describe amount of integers or any other structures. For now we describe the malicious message similarly as in the case of overflows regarding static-length buffers, requiring conformity of the length value with the actual size of the protocol fields. Thus as a value for \( L \) we provide the length field. To assure that the signature is indeed correct we need to verify it by checking whether Prospector or Argos spot an illegal operation if we send a message with critical fields filled with arbitrary bytes in the size slightly exceeding \( \text{length\_field} \). If it appears that we are wrong, the only thing we can do is use the semantics of the protocol for a description of the length field.

**Multiple fields.** By handling multiple fields, Prospector fixes and generalises the signature generation in Covers [124]. Also, unlike Covers, we do not require the protocol dissector to match the granularity in which the application works with protocol messages. The granularity of the dissector may be larger or smaller than that of the application. For instance, the dissector may indicate that a message contains two fields \( F_1 \) and \( F_2 \), while the application copies them in one in a single buffer in one go (essentially treating them as a single field \( F \)).

**False positives.** Observe that whenever an application with a given vulnerability receives network data containing the corresponding critical fields with a collective length exceeding \( L \) bytes, it will not fit in the application buffer, even if it does not contain any malicious data. Consequently passing it to the application would be inappropriate. In other words, regardless of content, the signatures will not incur false positives in practice. However, in an unlikely scenario it is possible that we cannot correctly determine the crucial tainted memory region, missing a protocol field. This may happen if the gap in crucial tainted memory region reaches the beginning of the buffer, and contains an extra protocol field not encountered before.
Notice however, that when we analyse a heap corruption attack which overwrote control data (a red region) on the heap, we will not miss any protocol fields, since the memory dump is performed exactly at the moment of corruption.

**Polymorphism.** By focusing on properties like field length, the signatures are independent of the actual content of the exploit and hence resilient to polymorphism. By focusing on the vulnerabilities, they also detect attacks with different payloads. Such behaviour is quite common, especially if part of the payload is stored in the same vulnerable buffer. As the signatures generated by Prospector identify vulnerabilities, they are application specific. As a result, we may generate a signature that causes control flow diversion in a specific version of an application, but there is no guarantee that this is also the case for a different version of the same application. In other words, we need precise information about the software we want to protect. The implication is that Prospector runs at the edge of the network.

**Value fields.** The critical fields and the condition that should be satisfied constitute the first, unpolished signature. In practice, however, we may want to characterise more precisely what messages constitute an attack. For instance, when the URL field is the critical field that overflows a buffer in a Webserver, it may be that the overflow only works on GET requests and not for POST requests. In our protocol-specific approach we therefore add a protocol module that determines per protocol which fields may be considered important (e.g., the request type in HTTP) and should therefore be added to the signature. We call such fields value fields as explained in the next section.

**The final form of Prospector’s signatures**

Every signature consists of a sequence of value fields and critical fields. A value field specifies that a field in the protocol should have this specific value. For instance, in the HTTP protocol a value field may specify that the method should be GET for this signature to match, or it could provide the name of a vulnerable Windows .dll. Critical fields, on the other hand, should collectively satisfy some condition. For instance, they should collectively have a length that is less/not less than $L$. We can also put some boundaries on given fields, like in the case of heap overflows based on malformed messages. Example signatures can be found in Section 4.6.1.

### 4.4.10 Double-free Errors

We added a module to Prospector to make it deal with double free attacks. Memory managers are sometimes exploited when a programmer makes the mistake of freeing a pointer that was already freed. Double-free errors do not share the characteristics of heap-corruption attacks in the sense that they do not overflow a buffer, and so when
considering the analysis they require special treatment. Prospector also contains a module that provides a means of analysis for such attacks.

Double-free exploits may overwrite any location, resembling the complex heap corruption attacks. Similarly, it is highly probable that when a violation is detected, the memory region that was holding the vulnerable buffer is reused and contains unrelated data. To deal with this issue, whenever `free` (or `realloc`) is called, we check for a potential double free error, assuring that the given memory location indeed points to the beginning of an allocated buffer. Otherwise we store the adjacent tainted memory region for possible later use.

To make the module working efficiently, we store a red-black tree of currently allocated memory regions for each address space. It guarantees fast access to the nodes when necessary.

Double free errors do not lead to buffer overflows like the other heap and stack corruption attacks. The current implementation of Prospector produces fairly trivial signatures for them by identifying a protocol field which should contain a selected substring of the crucial region. The crucial memory region is determined in the same way as for the complex heap corruption attack. When we pinpoint in the heap dump the address that caused the violation, we take its non-stale tainted neighbourhood as the invariable bytes for this attack. Notice that these bytes contain fake heap control data, and so are not supposed to be different in each instance of the message exploiting the vulnerability. However, there is a lot of space for improvement here, and we believe that the accurate data provided by Prospector can be used to produce a more powerful signature.

4.5 Implementation Details

Prospector was implemented for Linux on the Argos x86 emulator. In this section we discuss the most important implementation issues.

4.5.1 Monitoring Process Switches from the Hardware

Prospector stores information about the allocation and deallocation of stack frames in each process. Thus we need a means to monitor context switches on the level of processor emulator. This is not a trivial problem, as the hardware emulator has no knowledge of processes. The solution for IA-32 proposed in Antfarm [107] tracks changes of the `cr3` (or page directory base) register, which stores the physical address of the page directory. As a rule, a switch implies changing the set of active page tables, and thus loading `cr3` with the value stored in the descriptor of the process to be executed next. However, the solution is problematic, as Linux avoids this operation in the following cases: (1) when performing a switch between two regular processes that use the same set of page tables, i.e., lightweight processes, and (2) when performing a process switch between a regular process and a kernel thread.
Kernel threads do not have their own set of page tables; rather they use the page tables of the regular process that was scheduled last for execution on the CPU.

Proper tracking of context switches proved a very challenging problem. We sketch our solution that is accurate for Linux, so we do not have to worry about missing context switches. In Linux, each execution context that can be independently scheduled has its own process descriptor. Therefore even lightweight processes and kernel threads, have their own task_struct and thread_info structures. For each process, Linux keeps a memory area, at the beginning of which resides the thread_info structure, and the kernel mode process stack grows downward from the end. The length of this memory area is fixed, usually 8K. For reasons of efficiency the kernel stores the 8K memory area in two consecutive page frames with the first page frame aligned to a multiple of \(2^{13}\). Thus the 19 most significant bits of a memory location inside the kernel mode stack are the address of the thread_info structure, which we refer to as \(P\), and serves as a unique process identifier.

Whenever the CPU operates in kernel mode, we can determine \(P\) by taking the 19 most significant bits of the present stack pointer (ESP). As Qemu, on which Argos is based, translates all guest instructions to host native instructions by dynamically linking blocks of functions that implement the corresponding operations, we can check \(P\) right before the Argos emulator in kernel mode executes a block of instructions. On each context switch the OS always executes at least a few instructions in kernel mode, and so we always have a correct value of the process identifier.

### 4.5.2 Heap Protection

As explained in Section 4.4, to deal with complex heap corruption attacks, we mark the bytes surrounding allocated chunks of heap memory as red. Since we cannot monitor allocations at the level of the emulator, we interpose the malloc and free (also calloc and realloc) functions in the guest OS, and by means of argos calls inform Qemu about changes on the heap. Argos calls are analogous to system calls in Linux, and are called by trapping with an unused interrupt number (0x82). Whenever Argos receives this interrupt, it passes control to a handler corresponding to the argos call number. We use library interposition in this implementation. In principle, however, such method could be detectable by malware. In that case, we can move the interposition to the emulator (by looking for addresses of “interesting functions”). We have done this in Howard, described in Chapter 6.

When malloc returns, we provide Argos with the address addr and length len of the newly allocated chunk. Argos marks the bitmap’s entry corresponding to (addr - 1) as red, and also stores the value of len as a ’fake’ taint value for (addr - 1). This is safe as this address contains non-tainted control information used by the memory allocator and so will never really be tainted. Argos makes use of len when a memory chunk is released. We also set the red indicator for the byte directly above the chunk of memory. Similarly, when free is called, Argos cleans the red markers adjacent to the released memory buffer.
To support the heap corruption attacks based on malformed messages, we also check whether \( \text{len} \) is tainted. If so, we store its network origins in the red-black tree mentioned in Section 4.4.10, in a node corresponding to the newly allocated memory chunk.

### 4.5.3 Prospector Tagging

To deal with memory tagging Argos introduces a structure similar to page directories in Linux consisting of \textit{pagemaps} and \textit{bytemaps}. A pagemap is an array, where each entry corresponds to a bytemap keeping tags for a particular physical page. Here Argos stores all tags on the guest operating system memory, e.g., the network offsets that serve as taint tags. Initially only the pagemap is allocated. Bytemaps are added on demand, when tainted data is copied to a particular physical page for the first time. The network offset tags associated with each byte are 32 bits, what provides network indices of a reasonable size. (They are capable of addressing 4GB of network data before they wrap.) To support signature generation we doubled the size of the tag, yielding an additional 32 bits. Of these 32 bits, we designate one bit for the \textit{PFT} and \textit{FTS} indicators, one bit for the red marker denoting critical data on the heap, and the remaining 29 bits for the age stamp. We emphasise that age stamps serve only to compare tainted data, so they need only be incremented if a given value was used as a tag to mark tainted data. As most functions and indeed most processes never touch such data, the age stamp may remain untouched. As a result, the age stamp will wrap much more slowly. We will address age stamp wrapping in Section 4.5.4.

Qemu translates all guest instructions to host native instructions by dynamically linking blocks of functions that implement the corresponding operations. With the aim of tracking tainted data being copied to memory we instrument the \texttt{store} function to perform the operations of keeping track of age stamps and setting the extra indicators \texttt{(PFT} and \texttt{FTS}) described in Section 4.4.5. Here we also check whether the destination memory location is not marked as \texttt{red} (which indicates an overflow and perhaps a complex heap corruption attack, and therefore leads to a dump of the adjacent tainted memory).

### 4.5.4 AgeStamp Wrapping

\texttt{AgeStamp} is a 29-bit global variable used to draw conclusions about the age in which data coming from the network was copied to a buffer. Thus we wish to avoid problems due to \texttt{AgeStamp} wrapping. We could add more bits, but this may not be necessary. We measured the time needed by \texttt{AgeStamp} to wrap depending on its length. The tests were performed on the guest OS running Apache, receiving 45 re-

![Figure 4.5: Time to wrap for different sizes of AgeStamp.](image)
quests per second (a rate before it saturates on our emulator). Figure 4.5 presents the results. In the optimised version we increase AgeStamp by 1 only if it was actually used to tag network data in memory. Otherwise the value is updated on each call and ret.

In the optimised version AgeStamp of a guest OS running Apache needs almost 16 hours to wrap. We can use either of the following solutions to avoid the undesirable scenario described above: (1) restart the honeypot running Argos and Prospector twice a day, (2) dump all tags when AgeStamp wraps. This dump can be used for later analysis and separation of the values from the previous epoch. In the light of the long time needed by the counter to reach the limit both solutions are feasible. Even though we have currently implemented solution 1, the continuous operation of solution 2 is preferable.

### 4.5.5 Stale Red Markers

As mentioned earlier, to handle complex heap corruption attacks, we mark bytes surrounding allocated chunks of memory as red. If tainted data is written to a red region, this indicates illegal operations which trigger bookkeeping: the memory region is dumped. As we cannot rely on applications releasing all allocated memory, we may end up with stale red markers, possibly leading to unnecessary dumps of memory regions. We describe here how we solve this problem by removing false red indicators.

First of all, we keep counters indicating the number of red markers associated with each physical page in memory. To deal with the problem in the case of pages for the user stack or kernel memory, we monitor new entries added to the TLB as follows. We keep a table of physical pages associated with the identifier of the last process using it. Whenever a new entry corresponding to a kernel address or the user stack is added to the TLB buffer, we check whether the page has a new owner, and if so, we make sure that it does not contain any red markers. If so, we know that neither the user stack nor kernel memory contains the markers.

For the heap we cannot use this method, since dynamically allocated memory can easily be shared between processes, which could remove our markers. Thus, whenever a new buffer is allocated, we assure that its contents do not contain any red regions. First, we check the counter of red markers associated with the given page (or pages) and, if necessary, clean the memory.

### 4.6 Evaluation

We evaluate Prospector along two dimensions: effectiveness (Section 4.6.1) and performance (Section 4.6.2). While performance is not critical for a honeypot, it needs to be fast enough to generate signatures in a timely fashion. In Section 4.6.3 we also assess whether signatures generated by Prospector can be checked at high rates.
4.6. EVALUATION

4.6.1 Effectiveness

To test our analysis and signature generation, we launched a number of real attacks (as well as hand-crafted ones) against Linux on top of Argos. We have not experimented with Microsoft Windows since a small part of the functionality in Prospector is OS-specific, i.e., malloc and free function interposition and (partly) process switch monitoring. For launching attacks, we used the Metasploit framework\(^2\) and Milw0rm\(^3\). To make sure that Prospector does not incur false positives, we manually analysed the source code of vulnerable applications and verified generated signatures. We assured that any message matching our signatures will not fit in a buffer and will end up overwriting adjacent data in memory. In this section we illustrate how Prospector deals with four representative stack- and two heap overflow attacks.

**PeerCast Stack Overflow.** A remote overflow exists in PeerCast v0.1216 and earlier [182]. It fails to perform correct bounds checks on parameters passed in a URL, resulting in a stack-based overflow. An overly long query overwrites EIP stored on the stack. Thus when the function is about to return, Argos raises an alert and Prospector starts the signature generation phase. Our analysis engine correctly identified stale data on the stack. It encountered a 4-byte discontinuity in the critical tainted memory region, skipped it, and continued down the memory buffer. The final signature for this attack contains the following fields:

\[(\text{application: PeerCast, version: v0.1212,})\]
\[(\text{type: value_field, name: method, value: GET,})\]
\[(\text{type: critical_field, name: query,})\]
\[(\text{type: critical_length, value: 476}).]\]

**Subversion Stack Overflow.** There is a remote overflow in Subversion 1.0.2 [151] which fails to bounds check when calling sscanf() to decode old-styled date strings. By sending a crafted request via the get-dated-rev svn command, a remote attacker can cause a buffer overflow. In our experiment, an overly long week day overwrites EIP stored on the stack. As in the previous exploit, Argos raises an alert, and analysis starts. This time no discontinuities on the stack were encountered. The resulting signature contains the following fields:

\[(\text{application: Subversion, version: 1.0.2,})\]
\[(\text{type: value_field, name: command, value: get-dated-rev,})\]
\[(\text{type: critical_field, name: week_day,})\]
\[(\text{type: critical_length, value: 20}).]\]

**AlsaPlayer Stack Overflow.** A remote buffer overflow exists in AlsaPlayer 0.99.76 and earlier [142]. A long “Location” field sent by a web server triggers an overflow in the reconnect function in reader/http/http.c. The overflow results in EIP being overwritten. When the function returns, Argos raises an alert. Our analysis engine encountered a 4-byte discontinuity in the critical tainted memory region,


\(^3\)Milw0rm, www.milw0rm.com; now moved to www.exploit-db.com.
skipped it, and continued down the memory buffer. The final signature for this attack contains the following fields:

```
(application: AlsaPlayer, version: v.0.99.76,
(type: value_field, name: response header, value: Location),
(type: critical_field, name: Location Header),
(type: critical_length, value: 1032)).
```

**WvTftp Heap Overflow.** A heap-based overflow in the `WvTFTPServer::new_connection()` function in `wvtftpserver.cc` for WvTftp 0.9 allows remote attackers to execute arbitrary code via a long option string in a TFTP packet [178]. A TFTP packet has option name value pairs. They are given as a NULL terminated option name, followed by an ascii representation of the number value. The function `atoi()` is used on the value string, and as long as the original part of the string equals a value $> 8$ and $< 65464$, the string is `strcpy`d into the heap buffer. By supplying a long string for the value, the buffer can be overflown, and the attacker gains control over heap management structures. Argos correctly noticed that the heap control `red` region was overwritten with network data, and dumped adjacent memory for later use. In the analysis phase, no discontinuities on the heap were encountered. The resulting signature contains the following fields:

```
(application: WvTFTP, version: 0.9,
(type: value_field name: Opcode, value: Read Request (1)),
(type: critical_field, name: Blocksize option),
(type: critical_length, value: 557)).
```

**Asterisk Heap Overflow.** Asterisk is a popular software telephony server. The Asterisk Skinny channel driver for Cisco SCCP phones in v1.0.11 and earlier, v1.2.12 and earlier (`chan_skinny.so`) incorrectly validates a length value in the packet header. An integer wrap-around leads to a heap overwrite, and arbitrary remote code execution [179]. Asterisk checks whether the inequality `(length_value + 8 \leq 1000)` holds to convince itself that the user-supplied message fits in the local buffer of size 1000. Because of the integer wrap, the result of the comparison is positive. And then, the 4 bytes length are copied to the vulnerable buffer, and a `read` operation is performed storing `(length_value + 4)` bytes of the rest of the message on the heap. The attack benefits from the fact that `read` takes an unsigned value as the last parameter, so `length_value` is interpreted as a very large number instructing `read` to write beyond the allocated 1000 byte length of the vulnerable buffer. Argos detects that the control `red` region on the heap gets overwritten with network data, and dumps the corresponding memory area. In the analysis phase, we first come across the whole SKINNY message but the length field (this part has the same age stamp). Next, we include the 4 bytes underneath it, forming the length, in the crucial tainted memory region (since it is a tainted region with correctly fitting age stamps). Thus the signature specifies the whole SKINNY Packet for Asterisk 1.0.10 not to exceed 1000 bytes. Notice, that even though the length field does not need to be included in the signature, the attack description is still absolutely correct.
4.6. EVALUATION

libmusicbrainz Stack Overflow. libmusicbrainz is an open source library used in many multimedia programs for querying MusicBrainz servers. A boundary error within the Download function in lib/http.cpp (v. 2.1.2 and earlier) can be exploited to cause a buffer overflow via a large “Location” field in an HTTP redirection received from a malicious MusicBrainz server [143]. The overflow results in EIP being overwritten. When the function returns, Argos raises an alert. Our analysis engine encountered a 4-byte discontinuity in the critical tainted memory region. The final signature follows:

(application: libmusicbrainz, version: v.2.1.2,
(type: value_field, name: response header, value: Location),
(type: critical_field, name: Location Header),
(type: critical_length, value: 73)).

4.6.2 Analysis Performance

For realistic performance measurements we compare the speed of code running on Argos and Prospector with that of code running without emulation. Note that while this is an honest way of showing the slowdown incurred by our system, it is not necessarily the most relevant measure. After all, we do not use Prospector as a desktop machine and in practice hardly care whether results appear much less quickly than they would without emulation. The only moment when slowdown becomes an issue is when attackers decide to shun slow hosts, because it might be a honeypot. To the best of our knowledge, automated versions of such attacks do not exist in practice.

Performance evaluation was carried out by comparing the observed slowdown at guests running on top of various configurations of Prospector and unmodified Argos, with the original host. The host used during these experiments was an Intel(R) Xeon(TM) CPU at 2.8GHz with 2048KB of L2 cache, and 4GB of RAM, running Gentoo Linux with kernel 2.6.15.4. The guest OS ran Ubuntu Linux 5.05 with kernel 2.6.12.9, on top of Qemu 0.8, Argos and Prospector. To quantify the observed slowdown we used Apache 2.2.3. We chose Apache because it is a popular web server and thus it enables us to test the performance of a network service (a domain for which Argos was designed). We measured its throughput in terms of processed requests per second and the corresponding average response time. We used httperf for generating requests. Httperf is able to generate high rates of single-file requests to determine a web server’s maximum capacity.

Figure 4.6 shows the results of the evaluation. We tested the benchmark application at the guest running over Argos, and two different configurations of Prospector: both with and without the double free extension module. The graphs show see that the achieved throughput increases linearly with the offered load until the server saturates at a load of 48 calls per second in the case of Prospector and 57 for Argos.
This is also reflected in the figure presenting response times. The graph shows that the response time starts out at about 20-30ms, and then gradually increases until the server becomes saturated. Beyond this point, response time for successful calls remains largely constant at 3000ms.

Notice that there is no difference in performance between the two versions of Prospector. Calls to memory management related functions are rare in the context of the whole web server application, and so additional harmless operations on each malloc() and free() appear not to decrease performance.

We can conclude that the overhead expressed in throughput of a web server incurred by Prospector compared to Argos is approximately 16%. We have also performed measurements of slowdown in comparison with the original host (refer to [161] for the full performance evaluation of Argos.) Apache on Argos is about 15 times slower than the one run on the native operating system (on Prospector 18 times). We emphasise that we have not used any of the optimisation modules available for Qemu. These modules speed up the emulator to a performance of roughly half that of the native system. While it is likely that we will not quite achieve an equally large speed-up, we are confident that much optimisation is still possible. Moreover, even though the performance penalty is large, personal experience with Argos and Prospector has shown us that it is tolerable.

4.6.3 Signatures Checking Performance

The final part of the evaluation involves signatures matching performance. To assure that Prospector signatures can be checked at high rates, we have implemented a signature checking module for SafeCard [33].

SafeCard is a full intrusion prevention system (IPS) on an embedded network processor. SafeCard includes detection techniques at all levels of abstraction in communication: packets, reassembled TCP streams, application protocol units, and flow aggregates. The IPS is implemented as a pipeline on a single Intel IXP2400 network processor embedded on a network card. Its task is to enforce security policies on incoming traffic by means of in-depth analysis in the last hop toward the host. The system first receives traffic in a circular buffer and applies simple header-field filter-
Figure 4.7: Prospector’s signatures checking performance.

ing to determine which data needs further inspections. TCP streams that are classified as suspect are reassembled with an efficient in-place algorithm and fed into a per-stream pattern matching engine, similar to Snort [172]. For all streams that are not blocked by the pattern matching engine SafeCard checks whether higher-level protocol-specific rules exist and if so, checks them against these also. It is this last step where Prospector’s signatures are checked.

High-volume traffic must be processed on the micro-engines of an Intel IXP2400 NPU used by SafeCard. However, because these processors are scarce and hard to program, some processing will usually take place on the slower XScale. We have implemented Prospector’s signatures checking module on the XScale embedded in Streamline. The volume of traffic that is handled by the XScale must be considerably smaller than the Gigabit traffic handled in the fast path. As Prospector on Streamline currently checks HTTP request headers only, the volume is indeed small.

Figure 4.7 compares throughput of Prospector to that of a payload-scanning function (we used Aho-Corasick). We show two versions of Prospector: the basic algorithm that needs to touch all header data, and an optimised version that skips past unimportant data (called Prospector+). The latter relies on HTTP requests being TCP segment-aligned. This is not in any specification, but we expect it is always the case in practise.

Each method processes 4 requests. These are from left to right in the figure: a benign HTTP GET request that is easily classified, a malicious GET request that must be scanned completely, and two POST requests of differing lengths. In the malicious GET case all bytes have to be touched. Since AC is faster here than both versions of Prospector we can see that under equal memory-strain we suffer additional computational overhead.

However, all three other examples show that if you do not have to touch all bytes (the common case), a protocol-deconstruction is more efficient than scanning. Looking at the right-most figure, the longest POST request, we can see that the gap quickly grows as the payload grows. The benign GET learns us additionally that skipping remaining headers when a classification has been made can result in a dramatic (here 2-fold) increase in worst-case performance.

Note that none of these example requests carry a message body. This would also
be skipped by Prospector, of course. Even without message bodies, performance is continuously above 18,000 requests per second, making the function viable for in-line protection of many common services.

4.7 Discussion

While Prospector is effective in analysing buffer overflow attacks, and generating signatures for them, there is always room for improvement. In this section, we discuss its current limitations, and some possible future work.

Possible inaccuracy of the analysis As we explained in Sections 4.4.8-4.4.9, when an attack exploits a vulnerable buffer on the stack, and when the application overwrites the bottom part of the critical region, Prospector might miss some protocol fields that contributed to the attack. However, if the overwritten region did not contain an entire new protocol field, but only a part of it, the signature generated by Prospector is still correct.

Observe also, that this kind of inaccuracy never goes unnoticed — our analysis comes across the top of a gap in the critical region, but never finds its bottom. We can issue a warning, and in case of doubt, a user might discard the signature generated by Prospector. We have never come across such a situation though.

How to identify value fields? To generate its protocol-specific signatures, Prospector relies on a protocol module that determines per protocol which fields may be considered important, and should therefore be added to the signature as value fields (refer to Section 4.4.9). Currently, we build the protocol module manually.

An interesting alternative solution is precondition slicing employed by Bouncer [61]. To compute an attack filter, Bouncer first analyses the path followed by the program processing an exploit, and collects the conditions on the input message checked on that path. Any input that satisfies these conditions makes the program follow the same execution path as the sample exploit. Next, Bouncer applies precondition slicing to refine this initial filter by removing unnecessary conditions without adding false positives. It combines static path slicing [103] with dynamic slicing [114; 221] to select instructions on the path whose execution is sufficient to ensure that the vulnerability is exploited.

We think that Prospector would benefit from a similar technique. By identifying parts of the message which are sufficient to ensure that the vulnerability is exploited, we could automatically select value fields.

Integer overflows Prospector does not offer a generic method to detect integer overflows. As discussed in Section 4.4.9, we support only the case of heap overflows founded on malformed length.
4.8 Conclusions

We have described Prospector, an emulator capable of tracking which bytes contribute to an overflow attack on the heap or stack. By careful analysis, and keeping track of the age of data, we manage to provide such information with greater accuracy than previous approaches while maintaining reasonable performance. The information is important for security experts. Whenever we recognise the protocol governing the malicious network message, we also use the information unearthed by Prospector to generate signatures for polymorphic attacks by looking at the length of protocol fields, rather than the actual contents. In practice, the number of false positives for the signatures is negligible and the number of false negatives is also low. At the same time, the signatures allow for efficient filters.
Chapter 5

Hassle: How to Deal with Encrypted Channels?

5.1 Introduction

Intended to enhance the security of network communication, encryption also makes it harder to detect and analyse attacks on the Internet. Strong encryption and pacing on network links lead to traffic that is more or less uniformly distributed in space and time, preventing the extraction of useful information. Methods relying on the observation of traffic characteristics no longer work. Examples include methods like Snort that uses byte patterns [172], analysis of executable code in the network [159], static analysis techniques [201; 116] when applied in the network, and analysis of protocol fields [124; 125]. In addition, while advanced honeypot systems like Vigilante [62], TaintCheck [149] and Argos [161] would detect attacks, most common techniques for signature generation cannot be directly applied. Paradoxically, the very nature of encryption may turn against the original security goals.

At the same time, the use of encryption is increasing in almost all network services, including file systems, web servers, VPNs, databases, P2P, instant messaging, etc. Rather than considering the fairly narrow set of exploits against encryption libraries themselves (like Linux’ Slapper [157], and Windows’ SSL Bombs [35]), this work is motivated by the larger class of attacks that exploit the applications using encryption. It is well-known that given a choice between port 80 (http) and port 443 (https), attackers tend to opt for 443 almost without exception [162]. The reason is that the content of these channels cannot be so easily inspected by firewalls and virus scanners.

Instead of providing a NIDS with copies of the servers’ private keys [131], something administrators may be reluctant to do, we prefer to push fingerprinting to the end-application on the host. On the other hand, we do not want to code manually a specific solution for each application. Rather, we are interested in methods for
signature generation that apply to a wide variety of applications.

We emphasise that channel encryption should not be confused with polymorphism. Even though encryption yields unique network appearances for all network attacks, the nature of the attacks (polymorphic or not) still surfaces after decryption. Phrased differently, encrypted channels may carry both mono- and polymorphic attacks.

This chapter discusses a version of Argos known as Hassle, a honeypot capable of detecting and fingerprinting monomorphic and polymorphic attacks on SSL-encrypted channels. We have chosen the SSL library, because of its popularity. However, the techniques apply to other forms of encryption (IPSec) also. Hassle is not application-specific and can be applied to any process that uses SSL for secure communication. We discuss both its design and its implementation on an x86 Linux-based architecture.

SSL encryption. Encryption can be applied at many layers in the protocol stack. The most common examples in practice include the data-link layer (WEP, WPA), the network layer (IPSec), and the application (SSL). As layer-2 encryption in the NIC reduces the problem to that of non-encrypted channels at the OS level and can therefore be handled easily by emulators with dynamic taint analysis [161], the most interesting design alternatives to consider in practice concern IPSec and SSL. Without loss of generality, we opted for implementing Hassle for SSL, as it is supported by many servers. Nevertheless, the same techniques can be applied at other layers.

Contribution. To the best of our knowledge, we are the first to address the problem of signature generation (and attack filtering) for encrypted communication, while also handling non-encrypted channels. Since the techniques we describe work by interposing encryption routines, they are applicable to most types of encryption and require no modification of the applications that need protection. In addition, we describe how Hassle generates Prospector- (refer to Chapter 4) and Snort-like signatures [172].

While the idea and the architecture of Hassle were designed by me, some parts of the implementation were carried out by Michael Valkering [204].

Outline  The remainder of this chapter is organised as follows. Sections 5.2 and 5.3 discuss architecture and implementation, respectively. Hassle is evaluated in Section 5.5. In Section 5.6, we discuss related work, while conclusions are drawn in Section 5.7.

5.2 Architecture

At the highest level, our system consists of a detection engine, a signature generator, and a filter, as illustrated in Figure 5.1. The detector is a honeypot based on a full-system hardware emulator that provides taint analysis, a modified version of Argos [161], discussed in Section 3.2.
Assume for now that no encryption is used. All data from the network is logged to a rolling trace file. By means of taint analysis, Hassle tags and then tracks network data throughout the system, where a tag points to the origin of the data in the network trace. Whenever tainted data is used in a way that violates the security policy, we raise an alarm. Examples of such behaviour include attempts to execute tainted data, or loading tainted data in the x86 EIP register. At that point, Hassle dumps as much relevant data to disk as possible. For instance, for the process or kernel under attack, we save all tainted memory blocks with their tags, the names of the executable and the libraries used, and the address that triggered the alert together with its origin.

The signature generator correlates the data dumped by Hassle with those of the network trace to determine a signature. For instance, we can generate Prospector’s signatures discussed in Chapter 4. In this case the system determines which protocol fields were responsible for a buffer overflow and computes an upperbound on the combined length of the protocol fields as a signature. Any message in which the length of these protocol fields exceeds the upperbound is guaranteed to result in an overflow. We use the signature to block the attack elsewhere in the network without needing heavy-weight instrumentation.

Unfortunately, in case of encryption, correlation between network trace and memory dump is not possible as all memory tags point to seemingly meaningless, uniformly distributed bytes in the network trace. To solve this we want to restore a meaningful correlation, albeit not to the network trace directly. For encrypted channels we retain the tagged data after decryption. Concretely, we use library interposition to place a small amount of code between the application and the encryption library. The interposer requests the emulator to retain data using offsets in the decrypted streams as tags. The decrypted data is stored in a log for future use.

Signature generation now progresses much like that of non-encrypted channels, albeit at a higher level in the protocol stack. When working at this level, well above the transport layer, we cannot simply dissect the network data stream from the first network packet onwards to determine the protocol fields that were used in

![Figure 5.1: Interposition, retainting, and signature generation.](image-url)
the attack. Instead, we use the retainted decrypted network stream. Similarly, the filters that block traffic that contains the signature also must operate at higher-level protocol units \(^6\). We implement them as interposer filters that sit between the SSL library and the application, and flag or drop all traffic towards the application that matches the signature.

5.2.1 Retainting

Argos employs network tracking that keeps track of the network origin of tainted memory. Notice however that origin pointers are useless in the case of encrypted channels. The reason is that without the key we cannot perform the one-to-one mapping between bytes in memory and bytes in the network trace.

For this reason, Hassle retaints all encrypted data. Rather than pointing to a specific byte in the encrypted network trace, we make it point to a specific byte in the decrypted SSL stream. Of course, the nature of decrypted streams is different from that of the network trace. For instance, layer 2-4 headers are not visible and TCP flows have already been reassembled. As a result, we will have to adjust the signature generation and filtering components accordingly.

Two implementation issues remain. First, after separating encrypted and non-encrypted data we must retaint the data right after decryption in such a way that a tag used for retainting is unique across all streams. As a result, the tag cannot be a simple offset into any one particular SSL stream. Second, we should be able to uniquely identify SSL conversations and associate incoming data with an SSL stream. In the next two sections, we discuss our solution to each problem separately.

Determining the Tag

As decryption occurs in user space, we employ light-weight interposing libraries between the application and the SSL functions. Whenever a read is performed on an SSL stream, the data will be decrypted. At that point the interposer requests a retaint for the decrypted data and logs the decrypted data to file. Beyond that, the interposer serves as a low-overhead relay between the SSL library and the application.

While the interposer trivially knows the offset of decrypted data in the corresponding SSL stream, determination of the tag should not take place there. Given a tainted data item, Hassle must be able to identify exactly the decrypted SSL block in which it originates. In other words, a tag must be unique not only within its own SSL stream, but across all streams. Doing such retainting in the interposer requires adding a unique SSL stream identifier to each tag, which is both complex and expensive in memory.

Instead, we perform trivial retaining in the emulator and push all complexity to detection time. For the remainder of this section, refer to Figure 5.2 which zooms in on the decrypted data log and shows a situation where three SSL connections are active; the decrypted data blocks in the channels are tagged by the emulator.
5.2. ARCHITECTURE

We maintain, conceptually, a single log for all SSL streams and let the emulator determine a tag consisting of an offset in this global log. In reality, we store each SSL stream in separate append-only logs identified by a unique SSL stream identifier (the nature of which will be discussed later). For instance, the tags in Figure 5.2 refer to an offset in the global input. That is, the blocks that are tagged 0, 10, and 20 indicate that the first block starts at global offset 0, and since the next block starts at offset 10, the first block contains 10 bytes. Similarly, the third block starts at offset 20, so the second block also contains 10 bytes. However, while this is the second block in the global input, it is the first block in SSL stream 2. For completeness, the figure also shows on the left some tainted data that has been copied, leading to tag propagation.

In other words, Hassle orders and tracks all updates to the decrypted data log in a global order, layering a virtual append-only global log over the individual SSL stream logs. The log for SSL stream 1 in Figure 5.2 contains two data blocks, containing 10 and 20 bytes respectively. The global log, on the other hand, consists of five data blocks. Blocks 1 and 2 both contain 10 bytes; block 3 contains 50, and so on. Hassle tags the decrypted data with an offset into the global log, trivially guaranteeing uniqueness. When an attack is detected, the tags of offending bytes point to a specific block in the global log. We maintain a simple index to find the corresponding SSL stream and hence all decrypted data.

Finally, for each SSL stream we also store the original tag of the first decrypted data block. This tag points to a byte in the encrypted network trace where it originated. Thus, we are always able to find the network flow that carried the attack, which in turn enables us to identify the IP addresses and port numbers of the attack.

In summary, for retainting the interposer asks the emulator to determine a new tag for the data as explained earlier. It then pushes the decrypted data to the decrypted data log. For the first chunk of decrypted data in the SSL stream, we also log the association between the decrypted data and the original tag, enabling us to recover conveniently the network flow (IP addresses, ports, etc.) in which an attack originated.
Identifying the SSL conversation

The construction of a unique identifier for a single conversation is not trivial. Ideally, the identifier should be a unique number derived from one or more fields of the SSL connection structure. Simply using the memory address of the ssl structure (see Listing 5.1) will not suffice, because new conversations may reuse the structures associated with old conversations.

However, the handshake phase of SSL (version 3) connections includes the exchange of unique challenges by client and server, which can be obtained from the SSL structure. Unlike client challenges, server challenges cannot be influenced by clients, and are thus well-suited for identifying the conversation. Unfortunately, SSL version 2 does not support server challenges. While older versions of SSL are not our main concern, we decided to add some support for version 2. For such conversations, we currently resort to a combination of the client challenge with the memory address of the SSL connection structure and the thread id of the process using the OpenSSL library. Admittedly a hack, the values of the latter two are not controllable by any attacker and the combination is pseudo-unique.

5.2.2 Interposition Details

SSL conversations start with a handshake phase that deals with authentication and creation of a session key. No application data is transmitted during this phase and we therefore do not monitor it. This phase also creates the SSL structure for the conversation.

Whenever an application calls SSL_read to decrypt and read data, we intercept the call by library interposition. Besides the call to SSL_read, we are interested in a small subset of other calls, including SSL_shutdown and CRYPTO_num_locks. As an example, we show the code for the SSL_read interposer in Listing 5.1. In the first few lines we find (line 2) and execute (line 3) the real SSL_read function as requested by the client. Next, we retain the data and log the decrypted stream using the retain_netidx and inform_logclient functions, respectively. None of the retain functions are visible to the caller, rendering the interposer transparent to the client.

Listing 5.1: Interposer for ssl_read library function

```c
01 int SSL_read (SSL *ssl, void *buffer, int length) {
02 int (*func)() = ((int(*)(void))dlsym(RTLD_NEXT, "SSL_read"));
03 int func_result = func(ssl, buffer, length);
04 retain_netidx(...); // now retain
05 inform_log(...); // log decrypted data (Fig. 1)
06 return func_result; // return original result
07 }
```

A call to SSL_shutdown simply leads to destruction of state maintained by Hassle. CRYPTO_num_locks is more complex. OpenSSL uses a number of global data structures that will be implicitly shared when multiple threads use the library. To use the library in the context of threads safely, we need locks to prevent simultaneous
access to the global structures and `CRYPTO_num_locks` returns the number of locks needed by the library to synchronise access. Because our interposing library also implements a global data structure, the number of locks should be increased by one. So we interpose this function to make another lock available to protect the global data structure.

### 5.3 Signature Generation

As illustrated in Figure 5.1, signature generation is devolved from detection and different generators can be plugged into the architecture. We currently support two main classes of generator that will be referred to as *pattern-based* and *vulnerability-based*, respectively. Pattern-based signatures are widely used in network intrusion detection systems, such as Snort [172]. They consist of a basic identification of the type of packet (e.g., TCP or UDP and port number), together with a byte pattern which is matched against traffic of the appropriate type.

In vulnerability-based signatures we focus on buffer overflows on the heap and stack and decouple the signature from the attack’s content in bytes completely. Whenever we recognise the protocol governing the decrypted network message, we generate Prospector’s signature (Chapter 4), which consist of a bound on the combined length of a set of protocol fields. Any message where such fields have a combined length that exceeds this bound will incur an overflow, regardless of their content, so these signatures cater well to polymorphic attacks.

The main difference between encrypted and non-encrypted channels, as far as signatures are concerned, is where they are applied. For non-encrypted channels, we are able to apply signatures in the network before the malicious traffic reaches the host (indicated by $\mathbb{5}$ in Figure 5.1). In contrast, encrypted channels require the filters to be applied at a higher level, i.e., as an *interposer filter* in user-space (indicated by $\mathbb{6}$). In addition, an interposer filter must know which signatures to apply. To do so, the signature generator consults the network tag that was stored in the decrypted data log to find the corresponding flow in the network trace. By means of the flow, we obtain the port numbers used in the attack (and possibly other network-specific information). Finally, the data generated by Hassle specifies details about the application under attack. This is then used by remote clients to determine which interposer filters should be applied.

### 5.4 Filters

Hassle filters for non-encrypted traffic consist of simple checks, either matching pattern-based signatures against network packets, or – in the case of known protocols – looking at the length of fields of specific protocol messages for vulnerability-based signatures. They can be applied in the network or in the operating system kernel.
For encrypted channels, similar procedures are used, except that they are applied by means of library interposition in user-space. Filters should only apply those signatures that apply to the specific application that uses the SSL library. Currently, this is done by explicitly associating a separate interposer filter library with every application we want to protect. Each interposer filter only picks up the signatures for the application it is protecting. Since Hassle provides the full name of the executable as part of the signatures, the association is trivial.

5.5 Results

For realistic performance measurements we compare the speed of code running on Hassle with that of code running without emulation. While this is an honest way of showing the slowdown incurred by our system, it is not necessarily the most relevant measure, as we use Hassle as a honeypot rather than a desktop machine. To our knowledge, no automated attacks exist that shun slow hosts, because they might be honeypots.

It should also be mentioned that encryption is known to be one of the most challenging applications for dynamic taint analysis, because decryption requires a large number of tainted operations. For instance, recent work on demand emulation [100] describes a technique to speed up emulation-based taint analysis by switching to fast VM-mode when possible. While many applications incurred as little as a factor 2 slowdown, SSL incurred a slowdown of 150.

**Performance.** To quantify the observed slowdown we used the Apache 2.2.3 web server using the OpenSSL library. The first simple test consisted of requests to read a 5MB block from the client to the server, which on top of a vanilla Qemu emulator took Apache 19.9s to complete (2.06Mbps). On Hassle, the same task took 23.47s (1.75Mbps), incurring a 15% overhead.

We also evaluated Apache throughput in terms of number of processed requests per second and the corresponding average response time. We used httpperf for generating requests. The experiments were conducted on a dual Intel® Xeon at 2.80 GHz with 2 MB of L2 cache and 4 GB of RAM. The system was running SlackWare Linux 10.2 with kernel 2.6.15.4.

The results for https (using SSL) are summarised in Table 5.1. The table lists results for three Apache configurations: (i) running natively, (ii) running on the Argos honeypot, and (iii) running on Hassle. We also show some results for non-encrypted http connections for comparison¹. The results are the best possible in the sense that at this rate the webserver was able to keep up fully with the request rate, while not yet incurring unreasonably long response times. For instance, for all reported rates the response times were below 200ms. Beyond these rates, response times shot up to many hundreds or even thousands of milliseconds. All these results measure the

¹We were unable to measure reliably the native version for plain http, because httpperf at the client side became the bottleneck.
overhead of a version of Hassle without Prospector tracking.

The experiments confirm that SSL is very expensive for dynamic taint analysis, incurring a slowdown of approximately a factor 100 over native code running SSL, and a factor 70 over non-encrypted channels using the same (emulated) configuration. Consequently, dynamic taint analysis for SSL encrypted channels is only viable on honeypots, and even here the number of connections should be limited. Note however, that slowness is not really a major issue for a honeypot as long as it is able to serve a request sufficiently fast. Moreover, in most deployments of honeypots like Argos a first-pass filter of low-interaction honeypots is used to shield the high-interaction honeypot from most requests. The second thing to observe is that there is little difference between Hassle and the original Argos.

Probing further, it appeared that most of the overhead is in the connection set-up where SSL uses asymmetric encryption. As a result, performance improves significantly when use is made of https sessions. For instance, for 100 sessions per connection, the reply rates for https Hassle are shown in Table 5.2.

**Table 5.1:** Maximum rates for https connections

<table>
<thead>
<tr>
<th>description</th>
<th>average (req/s)</th>
<th>standard deviation</th>
<th>relative to native</th>
<th>response time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>https/native</td>
<td>57.0</td>
<td>0.3</td>
<td>1.0</td>
<td>21</td>
</tr>
<tr>
<td>https/Argos</td>
<td>0.6</td>
<td>0.07</td>
<td>95.0</td>
<td>63</td>
</tr>
<tr>
<td>https/Hassle</td>
<td>0.55</td>
<td>0.12</td>
<td>103.6</td>
<td>87</td>
</tr>
<tr>
<td>http/Argos</td>
<td>38</td>
<td>1.8</td>
<td>n/a</td>
<td>147</td>
</tr>
<tr>
<td>http/Hassle</td>
<td>38</td>
<td>1.7</td>
<td>n/a</td>
<td>200</td>
</tr>
</tbody>
</table>

**Micro-benchmarks.** Retainting itself is not very expensive. We measured 200 $\mu$s on a Pentium M at 1.4GHz with 1GB of RAM, running Ubuntu Linux 6.0.6. The guest OS ran Ubuntu Linux 5.05 with kernel 2.6.12.9, on top of Qemu 0.8. Argos and Hassle. Similarly, the overhead of the entire interposition library to do the retainting is modest. We measured performance with and without the interposition library for both SSL reads and SSL writes for block sizes ranging from 100B to 16 KB bytes. For writes, the relative overhead lies between 17.5% for the largest blocks and 26% for the smallest ones. For reads, the results range from 24.5% for the largest to 50% for the smallest blocks. Likewise, in our evaluation the interposer filters that scan SSL streams for the occurrence of a signature incurred overheads between 2% and 15% compared to a system without the filters, for the most expensive (pattern-based) signatures.

### 5.6 Related Work

To our knowledge, we are the first to tackle the problem of one-shot signature generation for communication on encrypted channels. Dynamic taint analysis, on the other hand, is well-known and used in TaintCheck [149], Vigilate [62], and Ar-
Library interposition as a way of monitoring interaction with libraries is used frequently to analyse applications [69] and generate audit trails [120]. Liang et al. [125] propose library interposition to learn about program inputs that lead to crashes induced by buffer overflows. In essence, they consider library calls made from a given program context and raise an alert when an input is significantly longer than the maximum input length seen in the past. Interposition is also applied at the system-call level either to confine the application [96], or to monitor the compliance of a sequence of calls with a predefined application model [163; 94]. In contrast, we intercept library calls to switch to tracking decrypted network streams by adjusting the tags in dynamic taint analysis.

Application-level filtering is performed by virus scanners and Vigilante. Filters in interposing libraries are not very common. While the paper is a bit vague about it, we suspect that they are also used in ARBOR [125], although the filters are of a very different nature.

### 5.7 Conclusions

We have described Hassle, a honeypot system that is capable of generating signatures for attacks over both encrypted and non-encrypted channels. For encrypted traffic we retain the tainted data by making the tags point to the decrypted SSL streams. Different types of signature generator can be used in the system. Which one should be used is a tradeoff between simplicity and accuracy. In our opinion, pattern-based signatures are useful for simple, non-polymorphic attacks, while vulnerability-based signatures work well with more advanced, polymorphic exploits. To our knowledge, we are the first to develop a system capable of fingerprinting attacks over encrypted channels and cater to both monomorphic and polymorphic exploits.
Part II

Beyond Dynamic Taint Analysis: Protecting Legacy Binaries against Memory Corruption Attacks
The security of an information system is only as strong as the weakest link. This link may well be an obscure, older library or program that is vulnerable to memory corruption attacks. While security issues occur in all software, older programs can be especially vulnerable, as they were often not designed with security in mind. And if they were, the programmers probably did not have knowledge of the latest and greatest exploitation techniques. While some vendors use managed, type-safe languages for new software, this is not true for legacy binaries and low-level system code. The irony is that even if we know that code is ancient and/or sloppily written, we typically do not have access to the source code, and thus cannot fix it.

The research community has long recognised the problem, and multiple interesting solutions were developed. As we said earlier, despite all these efforts, buffer overflows alone rank third in the CWE SANS top 25 most dangerous software errors [70]. From clients to servers and from big iron to mobile phones—all have fallen victim. The reason is that attackers adapt their techniques to circumvent the protective measure. Non-control-diverting attacks, such as the recent attacks on exim mail servers (discussed in Chapter 2) are perhaps the best known example of such sophisticated attacks [45; 191].

Attacks on non-control data are very hard to stop, because they do not divert the control flow, they do not execute code injected by the attacker, they do not even change the program’s behaviour in a noticeable way. And we do not currently have reliable measures that do not require recompilation to stop them—or even detect them—at all.

Reliable defences against non-control data attacks all require access to the source code [104; 6; 7]. Existing security measures at the binary level are good at stopping control-flow diversions [62; 149; 3; 113; 85], but powerless against corruption on non-control data. Moreover, current binary instrumentation systems like control flow integrity [3] and dynamic taint analysis [62; 149] detect the manifestation of attacks, rather than the attacks themselves. For instance, they detect a control flow diversion that eventually results from the buffer overflow, but not the actual overflow that occurred perhaps hundreds or even thousands of cycles before. The lag between time-of-attack and time-of-manifestation makes the task of analysing the attack and finding the root cause much more difficult (see Prospector in Chapter 4).
In Part II of the thesis, we try to address these issues, and develop defence mechanisms that would detect all kinds of buffer overflow attacks, including the non-control-diverting ones, and at the same time be applicable to existing legacy binaries.

As we have seen in the first part of the thesis, dynamic taint analysis, or more general — information flow tracking, transparently monitors how a program uses and propagates certain data, to accurately reason about the effects of code execution. These properties, together with the fact that it can be applied to existing legacy binaries, make the technique very attractive, and we exploit it further in our work.

Dynamic taint analysis keeps track of data coming from a malicious source, i.e., the network, and raises an alarm, to detect when it is about to directly influence the control flow of a program. Observe that this mechanism does not have any knowledge about the application being protected, nor does it spot that a buffer is overflowed. Its policy is based on the invariant that untrusted information should never directly influence the address of an instruction being executed. As we have discussed in Section 3.3, this invariant cannot be practically extended to cater to non-control-diverting attacks.

Since non-control-diverting attacks come with no clear manifestations, we need to go beyond dynamic taint analysis, i.e., tracking network data, to detect them. A possible solution could monitor the execution of a protected binary to make sure that once a pointer is assigned to a certain buffer, it never accesses memory beyond the buffer’s boundaries. Observe that this basic rule precludes both control-diverting and non-control-diverting attacks. Even though the requirement sounds straightforward, it assumes lots of knowledge about the binary being protected, e.g., about buffer locations together with pointers associated with them, or instructions accessing these buffers. Since we aim at protecting legacy binaries for which we do not have symbol tables, in Part II of the thesis, we investigate how to employ information flow tracking to gather the necessary information, and implement the attack detection policy.

We designed and developed BodyArmour, a tool chain to bolt a layer of protection on existing C binaries to shield them from state-of-the-art memory corruption attacks. BodyArmour employs dynamic information flow tracking. First, it monitors the execution of a vulnerable application to understand the layout of memory, and unearth buffer locations and sizes. Later, it hardens the application so that buffer overflows are no more possible. BodyArmour protects not only against control-diverting, but also against non-control-diverting attacks, so it has no problems detecting an attack on exim explained in Chapter 2. Further, it raises an alert immediately when a vulnerable buffer is overflowed, and not when a attacker controlled pointer is used to influence a program control flow. Finally, the performance of BodyArmour is much better than that of taint analysis, e.g., the slowdown for gzip is only 1.7x.

As we explain in detail in Chapter 7, in order to proceed, BodyArmour needs a notion of memory buffers in a protected binary. At the same time, a goal of this thesis is to develop a solution, which works even if the binary is completely stripped
and no symbol tables are available. To address this problem, BodyArmour extracts data structures from the binary itself by means of static [167; 20; 21] or dynamic analysis. As far as BodyArmour is concerned, the approaches are complementary. However, since current static analysis techniques are weak in array detection, we developed a dynamic approach, called Howard, to data structure recovery, and we focus on this one in the thesis (see Chapter 6). Howard is a dynamic data structure excavator that follows the simple intuition that memory access patterns reveal much about the layout of data structures. Something is a structure, if it is accessed like a structure, and an array, if it is accessed like an array. And so on. Howard tracks how pointers in a program are derived from each other, and based on that draws conclusion about the underlying data structures.

Once provided with the information about (potentially) vulnerable buffers, BodyArmour protects binaries by instrumenting array accesses to make sure that they are safe from buffer overflows. The basic idea is that once a vulnerable program has used a pointer to access an array, BodyArmour precludes this pointer from accessing memory beyond the array boundaries.

Both Howard and BodyArmour intensively employ information flow tracking. However, instead of monitoring how a program uses data coming from a suspicious source, they monitor how the program calculates and uses pointers to memory objects.

Outline In the following chapters, we first introduce Howard, our approach to data structure recovery (Chapter 6), and then we explain how we use it to implement BodyArmour (Chapter 7).
Chapter 6

Howard: Dynamic Data Structure Excavation

6.1 Introduction

State of the art disassemblers are indispensable for reverse engineering and forensics. The most advanced ones, like IDA Pro [75] and OllyDbg [2], offer a variety of techniques to help elevate low-level assembly instructions to higher level code. For instance, they recognise known library functions in the binary and translate all calls to these functions to the corresponding symbolic names in the presentation to the user. Some are sufficiently powerful to handle even binaries that are statically linked and ‘stripped’ so that they do not contain a symbol table.

However, they are typically weak in reverse engineering data structures. Since real programs tend to revolve around their data structures, ignorance of these structures makes the already complex task of reverse engineering even slower and more painful.

The research community has been aware of the importance of data structures in reverse engineering for several years now, but even so, no adequate solution emerged. The most common approaches are based on static analysis techniques like value set analysis [20], aggregate structure identification [167] and combinations thereof [169]. Some, like CodeSurfer/x86, are available as experimental plug-ins for IDA Pro [75]. Unfortunately, the power of static analysis is quite limited and none of the techniques mentioned above can adequately handle even some of the most common data structures – like arrays.

Some recent projects have therefore resorted to dynamic analysis. Again, success has been limited. The best known examples are Laika [65] and Rewards [127]. Laika’s detection is both imprecise and limited to aggregates structures (i.e., it lumps together all fields in a structure). This is not a problem for Laika’s application domain – estimating the similarity of different samples of malware by looking at the
approximate similarity of their data structures. However, for forensics and reverse engineering this is wholly insufficient.

Rewards [127] builds on a technique originally pioneered by Ramalingam et al. on aggregate structure identification (ASI) [167]. The idea is simple: whenever the program makes a call to a well-known function (like a system call), we know the types of all the arguments – so we label these memory locations accordingly. Next, we propagate this type information backwards and forwards through the execution of the program. For instance, whenever labelled data is copied, the label is also assigned to the destination. Rewards differs from ASI in that it applies this technique to dynamic rather than static analysis.

Either way, by definition the technique only recovers those data structures that appear, directly or indirectly, in the arguments of system calls (or the well-known library functions). This is only a very small portion of all data structures in a program. All internal variables and data structures in the program remain invisible.

In this chapter, we describe a new technique known as Howard that greatly improves on these existing techniques. It is complementary to Rewards, but much more powerful as it also finds internal variables. Like Rewards and Laika, Howard is based on dynamic analysis.

The main goal of Howard is to provide data structures that allow us to retrofit security onto existing binaries. As we already mentioned, using Howard’s results we protect legacy binaries against buffer overflows.

In addition, we use Howard to furnish existing disassemblers and debuggers with information about data structures and types to ease reverse engineering. For this purpose, it automatically generates debug symbols that can be used by all common tools. We will demonstrate this with a real analysis example using gdb.

Finally, the information gather by Howard can be used to further reverse engineer a binary. We discuss a novel approach that is able to recover high-level pointer structures, e.g., singly-, and doubly-linked lists or trees. The analysis is performed by observing connections between memory objects at runtime, and based on that reasoning about the shape of a linked data structure.

High-level overview. Precise data structure recovery is difficult because the compiler translates all explicitly structured data in the source to chunks of anonymous bytes in the binary. Data structure recovery is the art of mapping them back into meaningful data structures. To our knowledge, no existing work can do this. The problem becomes even more complicated in the face of common compiler optimisations (like loop unrolling, inlining, and elimination of dead code and unused variables) which radically transform the binary.

Howard builds on dynamic rather than static analysis, following the simple intuition that memory access patterns reveal much about the layout of the data structures. Something is a structure, if it is accessed like a structure, and an array, if it is accessed like an array. And so on.
Like all dynamic analysis tools, Howard’s results depend on the code that is covered at runtime – it will not find data structures in code that never executes. This work is not about code coverage techniques. Rather, as shown in Figure 6.1, we use existing code coverage tools (like KLEE [39]) and test suites to cover as much of the application as possible, and then execute the application to extract the data structures.

In summary, Howard is able to recover most data structures in arbitrary (gcc-generated) binaries with a high degree of precision. While it is too early to claim that the problem of data structure identification is solved, Howard advances the state of the art significantly. For instance, we are the first to extract:

• precise data structures on both heap and stack;
• not just aggregate structures, also individual fields;
• complicated structures like nested arrays.

Since our dynamic analysis builds on Qemu [25] process emulation which is only available for Linux, we target x86 Linux binaries, generated by gcc (various versions and different levels of optimisation). However, there is nothing fundamental about this and the techniques should apply to other operating systems also.

Outline The remainder of this chapter is organised as follows. We first highlight the most related approaches (Section 6.2). In Section 6.3, we explain the challenges we have to face, and in Section 6.4 – our solutions. We discuss the method’s limitations in Section 6.5, while in Section 6.6, we comment on the influence of code transformation techniques on Howard. In Section 6.7, we describe two additional applications of our system. We evaluate Howard in Section 6.8, and conclude in Section 6.9.

6.2 Related Work

Recovery of data structures is most obviously relevant to the fields of debugging and reverse engineering. Still, even the most advanced tools (like IDA Pro [75], and CodeSurfer [20]) are weak at identifying data structures. The limited support for
data structure recovery they provide comes exclusively in the form of static analysis. These techniques are inadequate, as we shall see below. As far as we know, Howard is very different from any existing approach. Nevertheless, we have been influenced by certain projects. In this section, we summarise them and their relation to Howard.

**Static analysis.** A first stab at recovering data without executing the program is to try to identify all locations that look like variables and estimate the sets of values that they may hold [20]. Known as Value Set Analysis (VSA), this approach uses abstract interpretation to find (over-approximations of) the sets of possible values.

Of course, accurately pinpointing the locations that hold variables in a program is not simple, but studying the way in which the program accesses memory helps. Incidentally, this idea originates in efforts to deal with the Y2K problem in old COBOL programs and is known as Abstract Structure Identification (ASI) [167]. Translated into C terms, ASI attempts to partition memory chunks statically in `struct`s of arrays and variables, depending on accesses. For instance, if a stack frame holds 40 bytes for local variables, and the program reads the 4 bytes at offset 8 in the range, ASI classifies the 40 bytes as a struct with one 4-byte variable wedged between 2 arrays. As more addresses are referenced, ASI eventually obtains an approximate mapping of variable-like locations.

ASI has another clever trick to identify data structures and types, and that is to use the type information from system calls and well-known library functions. As the argument types of these calls are known, at every such call, ASI tags the arguments with the corresponding types and propagates these tags through the (static) analysis.

The culmination of these static techniques is a combination of VSA and ASI by Balakrishnan et al. [169]. This powerful static analysis method is also available as an experimental plug-in for IDA Pro, known as CodeSurfer/x86 [19].

At this point, however, we have to mention that all of these static techniques have problems handling even the most basic aggregate data structures, like arrays. Nor can they handle some common programming cases. For instance, if a C `struct` is copied using a function like `memcpy`, VSA/ASI will misclassify it as having many fields of 4 bytes, simply because `memcpy` accesses the memory with a stride of 4 (on a 32 bit machine). Also, they cannot deal with functions like `alloca`. Finally, the combination of VSA and ASI in [21] is context-sensitive, which leads to state space explosion. The reported results show that even trivial programs take exceedingly long to analyse. In contrast, Howard does not depend on static analysis at all.

In a more constrained setting, Christodorescu et al. [52] show how static analysis of x86 executables can help recover string values. Their technique detects C-style strings modified by the `libc` string functions.

**Dynamic analysis.** Eschewing static analysis, Laika [65] recovers data structures during execution in a novel way. First, Laika identifies potential pointers in the memory dump—based on whether the contents of 4 byte words look like a valid pointer—
and then uses them to estimate object positions and sizes. Initially, it assumes an object to start at the address pointed to and to end at the next object in memory. It then converts the objects from raw bytes to sequences of block types (e.g., a value that points into the heap is probably a pointer, a null terminated sequence of ASCII characters is probably a string, and so on). Finally, it detects similar objects by clustering objects with similar sequences of block types. In this way, Laika detects lists and other abstract data types.

On the other hand, Laika’s detection is both imprecise and limited to aggregates. For instance, it may observe chunks of bytes in what looks like a list, but it does not detect the fields in the structures. For debugging, reverse engineering, and protection against overflows, this is wholly insufficient. The authors are aware of this and use Laika instead to estimate the similarity of malware.

Originally, Howard borrowed from Laika the idea of dynamically classifying blocks of null-terminated printable characters as “probable string blocks” to improve the speed of string detection. However, as we improved our default string detection method, the additional accuracy provided by the Laika method was very small and we therefore removed it.

Rewards [127] builds on the part of ASI that propagates type information from known parameter types (of system calls and library functions). Unlike ASI, however, it does so dynamically, during program execution. Rewards recognises well-known functions (known as type sinks) and propagates the types of their arguments. All data structures that are used in, or derived from, system calls or known templates are correctly identified. However, Rewards is (fundamentally) not capable of detecting data structures that are internal to the program.

Howard emphatically does not need any known type to recover data structures, but whenever such information is available, it takes advantage of it to recover semantics. For instance, it may help to recognise a structure as a sock_addr structure, a file descriptor, or an IP address.

One of the more complicated features of Howard is its loop detector, which we use to detect array accesses. It is a bit similar to LoopProf [139], as it also keeps track of basic blocks executed, and detects loops dynamically.

**Dynamic protocol format reverse engineering.** Different in nature, but still related is the problem of automatic protocol reverse engineering. Systems like Polyglot [37], an approach proposed by Wondracek et al. [214], AutoFormat [126], Tupni [68], ReFormat [212], and Prospex [56], aim to analyse how applications parse and handle messages to understand a protocol’s format. They typically do so dynamically, although some are supplemented by static analysis. While different in various regards, these systems track data coming from the network and by observing the applications’ behaviour try to detect constant protocol fields, length fields, delimiters, and so on. The most advanced ones also cluster different types of messages to recover the protocol’s state machine. One of Howard’s array detection methods was influenced by
that of Polyglot [37]. Still, it is considerably more advanced. In addition, we add a second, completely new array detection technique.

6.3 Recovery by Access Patterns: Challenges

Howard aims to answer questions like: “What are the variables, structs, and arrays in the program and what are their layouts?” As explained in Section 6.2, for a subset of these variables (for which we have type sinks), we are also able to recover semantics, so that we can answer questions like: “Is the 4-byte field in this struct an IP address, or an integer?”

Howard recovers data structures by observing how memory is used at runtime. In the CPU, all memory accesses occur via pointers either using direct addressing or indirectly, via registers. The intuition behind our approach is that memory access patterns provide clues about the layout of data in memory. For instance, if $A$ is a pointer, then a dereference of $*\,(A+4)$ suggests that the programmer (and compiler) created a field of size 4 at $A$. Intuitively, if $A$ is a function frame pointer, $*\,(A+4)$ and $*\,(A-8)$ are likely to point to a function argument passed via the stack, and a local variable, respectively. Likewise, if $A$ is the address of a structure, $*\,(A+4)$ presumably accesses a field in this structure, and finally, in the case of an int[] array, $*\,(A+4)$ is its second element. As we shall see, distinguishing between these three scenarios is one of the challenges we need to address.

It is not the only issue. In the remainder of this section, we discuss the main obstacles that we had to remove to make Howard possible. Some are relatively straightforward and for these we discuss immediately how we solved them. Others are not, and we postpone their solution to Section 6.4, where we discuss our approach in full.

Even though we discuss many details, due to space limitations, we are not able to discuss everything in great depth. We realise that some readers are interested in all the details, and for this reason we made available a technical report that contains enough information to allow one to reproduce our system [187]. From time to time, we will refer readers interested in details to the report.

**Memory allocation context** Our work analyses a program’s use of memory, which includes local function variables allocated on the stack, memory allocated on the heap, and static variables. Static memory is not reused, so it can be uniquely identified with just its address. However, both the runtime stack and heap are reused constantly, and so a description of their data structures needs to be coupled with a context.

For the stack, each invocation of a function usually holds the same set of local variables and therefore start addresses of functions are sufficient to identify function frames. A possible exception occurs with memory allocated by calls to functions like `alloca`, which may depend on the control flow. As a result, the frames of different
invocations could differ. While Howard handles these cases correctly, the details are tedious and beyond the scope of this chapter (see [187]). For now, it suffices to think of function start addresses as the frame identifiers.

For heap memory, such simple pointers will not do. Consider a `my_malloc` wrapper function which invokes `malloc` and checks whether the return value is null. Since `my_malloc` can be used to allocate memory for various structures and arrays, we should not associate the memory layout of a data structure allocated by `my_malloc` with `my_malloc` itself, but rather with its caller. As we do not know the number of such `malloc` wrappers in advance, we associate heap memory with a call stack. We discuss call stacks in detail in Section 6.4.1.

**Pointer identification** To analyse memory access patterns, we need to identify pointers in the running program. Moreover, for a given address \( B = A + 4 \), we need to know \( A \), the base pointer from which \( B \) was derived (e.g., to find nested structures). However, on architectures like x86, there is little distinction between registers used as addresses and scalars. Worse, the instructions to manipulate them are the same. We only know that a particular register holds a valid address when it is dereferenced. Therefore, Howard must track how new pointers are derived from existing ones. We discuss our solution in Section 6.4.2.

**Missing base pointers** As mentioned earlier, Howard detects new structure fields when they are referenced from the structure base. However, programs sometimes use fields without reference to a base pointer, resulting in misclassifications. Figure 6.2 illustrates the problem. Field `elem.y` is initialised via the frame pointer register `EBP` rather than the address of `elem`. Only the update instruction 7 hints at the existence of the structure. Without it, we would characterise this memory region as composed of 3 separate variables: `pelem`, `x`, and `y` (but since the program here does not actually use the connection between the fields `x` and `y`, this partially inaccurate result would be innocuous). A missing base pointer is of course a fundamental limitation, as we cannot recognise what is not there. In practice, however, it does not cause many problems (see also Section 6.8).

**Multiple base pointers** Conversely, memory locations can be accessed through multiple base pointers, which means that we need to decide on the most appropriate one. Observe that field `elem.y` from Figure 6.2 is already referred to using two different base pointers, the frame pointer `EBP` and `pelem (EAX)`. While this particular case is tractable (as `pelem` is itself based on `EBP`), the problem in general is knotty. For instance, programs often use functions like `memset` and `memcpy` to initialise and copy data structures. Such functions access all bytes in a structure sequentially, typically with a stride of one word. Clearly, we should not classify each access as a separate word-sized field. This is a serious problem for all approaches to date, even the most advanced ones [169].
typedef struct {
    int x;
    int y;
} elem_t;

void fun() {
    elem_t elem, *pelem;
    elem.x = 1;
    elem.y = 2;
    pelem = &elem;
    pelem->y = 3;
}

Figure 6.2: The function initialises its local variable elem. Pointer pelem is located at offset -4 in the function frame, and structure elem at -0xc. Instructions 4 and 5 initialise x and y, respectively. Register EAX is loaded with the address of pelem in instruction 6, and used to update field y in 7.

One (bad) way to handle such functions is to blacklist them, so their accesses do not count in the analysis. The problem with blacklisting is that it can only cope with known functions, but not with similar ones that are part of the application itself. Instead, we will see that Howard uses a heuristic that selects the “less common” layout. For instance, it favours data structures with different fields over an array of integers.

Code coverage As Howard uses dynamic analysis, its accuracy increases if we execute more of the program’s code. Code coverage techniques (using symbolic execution and constraint solving) force a program to execute most of its code. For Howard, the problem is actually easier, as we do not need all code paths, as long as we see all data structures. Thus, it is often sufficient to execute a function once, without any need to execute it in all possible contexts. In our work, we use KLEE [39]. A recent work at EPFL allows it to be used on binaries [47]. Figure 6.1 illustrates the big picture. In reality, of course, KLEE is not perfect, and there are applications where coverage is poor. For those applications, we can sometimes use existing test suites.

6.4 Howard Design and Implementation

We now discuss the excavation procedure in detail. In the process, we solve the remaining issues of Section 6.3.

6.4.1 Function Call Stack

As a first step in the analysis, Howard keeps track of the function call stack. As Howard runs the program in a processor emulator, it can dynamically observe call and ret instructions, and the current position of the runtime stack. A complicating factor is that sometimes call is used not to invoke a real function, but only as part
of a call/pop sequence to read the value of the instruction pointer. Similarly, not every ret has a corresponding call instruction.

We define a function as the target of a call instruction which returns with a ret instruction. Values of the stack pointer at the time of the call and at the time of the return match, giving a simple criterion for detecting uncoupled call and ret instructions\(^1\).

Whenever we see a function call, we push this information on a Howard internal stack, which we refer to as HStack.

### 6.4.2 Pointer Tracking

Howard identifies base pointers dynamically by tracking the way in which new pointers are derived from existing ones, and observing how the program dereferences them. In addition, we extract root pointers that are not derived from any other pointers. Howard identifies different root pointers for statically allocated memory (globals and static variables in C functions), heap and stack.

For pointer tracking, we extended the processor emulator so that each memory location has a tag, MBase(addr), which stores its base pointer. In other words, a tag specifies how the address of a memory location was calculated. Likewise, if a general purpose register holds an address, an associated tag, RBase(reg), identifies its base pointer.

We first present tag propagation rules, and only afterward explain how root pointers are determined.

When Howard encounters a new root pointer \(A\), it sets MBase(A) to a constant value root to mark that \(A\) has been accessed, but does not derive from any other pointer. When a pointer \(A\) (root or not) is loaded from memory to a register \(reg\), we set \(RBase(reg)\) to \(A\).

The program may manipulate the pointer using pointer arithmetic (e.g., add, sub, or and). To simplify the explanation, we assume the common case, where the program manipulates pointers completely before it stores them to memory, i.e., it keeps the intermediate results of pointer arithmetic operations in registers. This is not a limitation; it is easy to handle the case where a program stores the pointer to memory first, and manipulates and uses it later.

During pointer arithmetic, we do not update the RBase(reg), but we do propagate the tag to destination registers. As an example, let us assume that after a number of arithmetic operations, the new value of \(reg\) is \(B\). Only when the program dereferences \(reg\) or stores it to memory, do we associate \(B\) with its base pointer which is still kept in \(RBase(reg)\). In other words, we set MBase(B) to \(A\). After all, the program accessed this memory location via base pointer \(A\). This way we ensure that base pointers always indicate valid application pointers, and not intermediate results of pointer arithmetic operations.

\(^1\)In rare cases, functions are reached by a jump. Howard merges these functions with the caller. We discuss the impact on the analysis in [187].
Extracting root pointers We distinguish between 3 types of root pointers: (a) those that point to statically allocated memory, (b) those that point to newly allocated dynamic memory, and (c) the start of a function frame which serves as a pseudo root for the local variables.

Dynamically allocated memory. To allocate memory at runtime, user code in Linux invokes either one of the memory allocation system calls (e.g., mmap, mmap2) directly, or it uses one of the libc memory allocation routines (e.g., malloc). Since Howard analyses each memory region as a single entity, we need to retrieve their base addresses and sizes. Howard uses the emulator to intercept both. Intercepting the system calls is easy - we need only inspect the number of each call made. For libc routines, we determine the offsets of the relevant functions in the library, and interpose on the corresponding instructions once the library is loaded.

Statically allocated memory. Statically allocated memory includes both static variables in C functions and the program’s global variables. Root pointers to statically allocated memory appear in two parts of an object file: the data section which contains all variables initialised by the user - including pointers to statically allocated memory, and the code section - which contains instructions used to access the data. To extract root pointers, we initially load pointers stored in well-defined places in a binary, e.g., ELF headers, or relocation tables, if present. Next, during execution, if an address \( A \) is dereferenced, \( \text{MBase}(A) \) is not set, and \( A \) does not belong to the stack, we conclude that we have just encountered a new root pointer to statically allocated memory. Later, if we come across a better base pointer for \( A \) than \( A \) itself, \( \text{MBase}(A) \) gets adjusted.

Stack memory. Function frames contain arguments, local variables, and possibly intermediate data used in calculations. Typically, local variables are accessed via the function frame pointer, EBP, while the remaining regions are relative to the current stack position (ESP).

As we do not analyse intermediate results on the stack, we need to keep track of pointers rooted (directly or indirectly) at the beginning of a function frame only (often, but not always, indicated by EBP). Usually, when a new function is called, 8 bytes of the stack are used for the return address and the caller’s EBP, so the callee’s frame starts at \((\text{ESP}-8)\). However, other calling conventions are also possible [187]. This means that we cannot determine where the function frame will start. To deal with this uncertainty, we overestimate the set of possible new base pointers, and mark all of them as possible roots. Thus, Howard does not rely on the actual usage of the EBP register. If, due to optimisations, EBP does not point to the beginning of the frame, nothing bad happens.

6.4.3 Multiple Base Pointers

As a program often accesses a memory location \( A \) through multiple base pointers, we need to pick the most appropriate one. Intuitively, selecting the base pointer that is closest to the location, usually increases the number of hops to the root pointer,
Figure 6.3: Example of a memory area accessed using multiple base pointers. The arrows on top illustrate a function like `memset` that accesses all fields with a stride of 4 bytes, while the ‘real’ access patterns, below, show accesses to the individual fields.

and so provides a more detailed description of a (nested) data structure.

However, as shown in Figure 6.3, functions like `memset` and `memcpy` often process composite data structures. These functions are completely unaware of the actual structure and access the memory in word-size strides. Thus, for 32 bit machines, such functions continuously calculate the next address to dereference by adding 4 to the previous one covering the entire data structure in 4 byte strides. By applying the aforementioned heuristic of choosing the closest base pointer, we could easily build a meaningless recursively nested data structure.

For `structs` the solution is often simple. When the program accesses the memory twice, once with constant stride equal to the word size (e.g., in `memset`) and once in a different manner (when the program accesses the individual fields), we should pick the latter. In arrays, however, multiple loops may access the array. To deal with this problem, we use a similar intuition and detect arrays and structures dynamically with a heuristic preference for non-regular accesses and/or accesses at strides not equal to the word size. For instance, if a program accesses a chunk of memory in two loops with strides 4, and 12, respectively, we will pick as base pointers those addresses that correspond to the latter loop. Intuitively, a stride of 12 is more likely to be specific to a data structure layout than the generic 4.

Our current array detection introduces three categories of loop accesses (see Figure 6.4): (1) accesses with non-constant stride, e.g., an array of strings, (2) accesses with a constant stride not equal to the word-size, e.g., 1 or 12, and (3) accesses with stride equal to the word-size. Our heuristic, then, is as follows. First select the base pointers in the best possible category (lower is better), and next, if needed, pick the base pointer closest to the memory location. Next, we discuss arrays and loops in detail.

### 6.4.4 Array Detection

Array detection is both difficult and important – especially for security. Howard recovers arrays when the program accesses them in loops. Fortunately, this is true for the vast majority of arrays. In the simplest case, the program would access an array in an inner loop, with one array element in each loop iteration. However, a general solution must also handle the following scenarios: (a) multiple loops accessing the same array in sequence, (b) multiple nested loops accessing the same array, (c) loop
1. variable offsets
2. stride not equal to wordsize
3. stride equal to wordsize

Figure 6.4: Different memory access patterns. When Howard observes different access patterns to the same object, it prefers pattern 1 over patterns 2 and 3, and 2 over 3.

unrolling, resulting in multiple array elements accessed in one loop iteration, (d) inner loops and outer loops not iterating over the same array, and (e) boundary array elements handled outside the loop. Howard uses several complementary array detection methods to deal with all these cases. We divide these methods into two major classes depending on the way array elements are accessed by the program.

Loops in real code implement one of two generic schemes for deriving array element addresses: (1) relative to the previous element, realised in instructions like \( \text{elem} = * (\text{prev}++) \), and (2) relative to the base of an array, \( \text{elem} = \text{array}[i] \). As we shall see, Howard handles both cases. We discuss limitations of our array detection method in Section 6.5.

**Accesses relative to previous elements**

To handle loop accesses in a buffer where each element address is relative to a previous one, Howard is set up to track chained sequences of memory locations. For instance, if \( \text{elem} = * (\text{pprev}++) \), then the pointer to \( \text{elem} \) is derived from \( \text{pprev} \).

No loop unrolling. We explain what happens for the non-optimised case first and address loop-unrolling later. Howard identifies each loop with a timestamp-like id \( \text{lid} \) which it assigns to the loop head at runtime when the back edge is taken for the first time. (See Figure 6.5.) At this point this loop head is pushed on \( \text{HStack} \). So, if a loop executes just once and never branches back for a second iteration, it does not get a new \( \text{lid} \). Howard assigns the top \( \text{lid} \) as a tag to each memory location \( A \) the code accesses: \( \text{MLid}(A) := \text{lid} \).

Thus, memory accesses in the first iteration of a loop get the parent \( \text{lid} \). Tags are kept similarly to \( \text{MBase} \), in the emulator. If there are no loops on the call stack, pushed functions are assigned a new \( \text{lid} \). Otherwise, new functions inherit the top loop \( \text{lid} \).

Writing \( B \leftarrow A \) to denote that pointer \( B \) is derived from pointer \( A \), conceptually Howard detects arrays as follows. When pointer \( B \), with \( B \leftarrow A \), is dereferenced in iteration \( i \), while \( A \) was dereferenced in a previous iteration, Howard treats \( A \) as
6.4. HOWARD DESIGN AND IMPLEMENTATION

(a) Control flow graph of a loop that derives array element addresses relative to the previous element. Basic block $T$ is the loop head, and $V \rightarrow T$ the back edge. (b) An example ten-element array accessed by the loop in (a). Three diagrams present the information about the array gathered by Howard at different points of the loop execution. The dashed squares indicate the most recently accessed array element. Arrows represent base pointers as determined by the loop. $S$ and $T$ above the array elements are assigned during the loop execution and indicate loop head ids, MLids. In step 1, Howard checks that $a$, $b$, and $c$ were accessed in the current loop (or just before), and decides (step 2) to store information about a new array $[b]$. It notes that the array should possibly be extended later to contain $a$ and $c$. As more iterations of the loop are executed (step 3), the array contains elements $[b\ldots e]$ with $a$ and $f$ marked as potential extensions.

Figure 6.5: (a) Control flow graph of a loop that derives array element addresses relative to the previous element. Basic block $T$ is the loop head, and $V \rightarrow T$ the back edge. (b) An example ten-element array accessed by the loop in (a). Three diagrams present the information about the array gathered by Howard at different points of the loop execution. The dashed squares indicate the most recently accessed array element. Arrows represent base pointers as determined by the loop. $S$ and $T$ above the array elements are assigned during the loop execution and indicate loop head ids, MLids. In step 1, Howard checks that $a$, $b$, and $c$ were accessed in the current loop (or just before), and decides (step 2) to store information about a new array $[b]$. It notes that the array should possibly be extended later to contain $a$ and $c$. As more iterations of the loop are executed (step 3), the array contains elements $[b\ldots e]$ with $a$ and $f$ marked as potential extensions.

Loop unrolling. The algorithm is simple and intuitive and sketches the main idea fairly accurately, but it is a (slight) simplification of Howard’s real array detection algorithm. In reality, we cannot just look at loop iterations, as common optimisations like loop unrolling force us to look at these patterns within a single iteration also. For completeness, we briefly explain the details.

Assume that the loop id of the current loop is $Lid_T$, while the previous element on the call stack has id $Lid_S$. Also assume that the pointer $C$ is dereferenced for the first time in the loop $T$, where $C \leftarrow B$ and $B \leftarrow A$.

Howard now treats $B$ as a likely array element if the following conditions hold:

1. $MLid(B) \geq Lid_T$, which means that $B$ was accessed in the current loop (regardless of the iteration),

2. $MLid(A) \geq Lid_S$, which means that $A$ was accessed either in the current loop or just before it.

It stores information about the array in $T$, the top loop head on $\text{HStack}$. Whenever a new piece of information is added, Howard tries to extend arrays already discov-

---

2There is a subtle reason why we do not classify $B$ as an array element (yet): if the array consists of structs, $B$ may well point to a field in a struct, rather than an array element.
ered in the top loop head. We will discuss shortly how this works when multiple loops access the same array. First, we look at boundary elements.

**Boundary elements**  Because the first element of an array is often accessed before a new id is assigned to the loop head (remember, a new loop id is assigned only once the back edge is taken), Howard explicitly checks for extending the array. It looks for earlier memory accesses at the base pointers used to recursively derive the first element of the array. Before we continue with the algorithm, see Figure 6.5 for an example array detection scenario.

**Multiple loops**  To handle arrays accessed in multiple loops or functions, arrays that are found in inner loops are passed to the previous element on HStack, whether it be a function or an outer loop node. Howard must decide if the findings of the inner exiting loop ought to be merged with the outer loop results or whether they represent internal arrays of nested structures and should be kept separately. Intuitively, Howard waits to determine whether (1) the array detected in the internal loop will be further extended in the outer loop — hinting at the existence of an iterator which prompts Howard to merge the results, or (2) whether the array detected in the internal loop is not extended further, and kept independent from the outer loop — prompting Howard to classify it as a nested structure.

**Accesses relative to the base**

The above technique helps us detect sequentially accessed arrays where each next address is relative to the previous one. Sometimes, however, accesses are relative to the base pointer. These accesses may or may not be sequential. For instance, randomly accessed arrays like hash tables fall in this category.

For this type of access Howard uses a second method that bears a superficial resemblance to the array detection method in Polyglot [37], but is considerably more powerful. Essentially, Howard tracks all instructions in a loop that access a set of addresses that can be mapped to a linear space \((\text{stride} \times x + \text{offset})\) and that all share the same base pointer (See Figure 6.6.a). If the accesses share a base pointer, they almost certainly belong to the same array, even if the accesses came from different instructions in the loop. Moreover, we can easily extend the array to include boundary elements (e.g., a last element that is accessed outside the loop), because it will share the same base pointer. If the accesses do not share the base pointer (Figure 6.6.b), Howard classifies them as different arrays.

Existing methods like those used in Polyglot [37], and also Rewards [127], only check whether single instructions in a loop access an area that can be mapped to a linear space \((\text{stride} \times x + \text{offset})\). Therefore, they can handle only the simplest of cases. They fail when the program: (1) accesses arrays in multiple loops/functions, (2) accesses boundary elements outside the loop, (3) has multiple instructions that access the same array within one loop (very common with unrolled loops or loops
6.4. HOWARD DESIGN AND IMPLEMENTATION

Figure 6.6: Array accesses from the base pointer. The labels denote whether the accesses are by instruction i1 or i2. In (a) we see a single array accessed by multiple instructions in a single loop, while in (b), the access patterns are similar except that we now have two arrays. Howard distinguishes the two cases by looking for a shared base pointer.

containing conditional statements like if or switch), and (4) allocates the arrays on the stack or statically. These are all common cases.

Are both methods necessary?

The two array detection methods in Howard are complementary. Both are necessary. The first detects accesses relative to the base pointer (like hash tables), while the second detects accesses where each next address is relative to the previous. The combination of our two techniques works quite well in practice. Howard is able to detect nested arrays, hash tables and many other complex cases. We discuss some limitations in Section 6.5.

6.4.5 Final Mapping

Having detected arrays and the most appropriate base pointers, Howard finally maps the analysed memory into meaningful data structures. For a memory chunk, the mapping starts at a root pointer and reaches up to the most distant memory location still based (directly or indirectly) at this root. For static memory, the mapping is performed at the end of the program execution. Memory allocated with malloc is mapped when it is released using free, while local variables and function arguments on the stack are mapped when a function returns.

Mapping a memory region without arrays is straightforward. Essentially, memory locations which share a base pointer form fields of a data structure rooted at this pointer, and on the stack, memory locations rooted at the beginning of a function frame represent local variables and function arguments.

When a potential array is detected, we check if it matches the data structure pattern derived from the base pointers. If not, the array hypothesis is discarded. E.g., if base pointers hint at a structure with variable length fields, while the presumed array has fields of 4 bytes, Howard assumes the accesses are due to functions like memset. The analysis may find multiple interleaving arrays.
6.4.6 Partial Recovery of Semantics

As mentioned earlier, type sinks are functions and system calls with well-known prototypes. Whenever Howard observes a call to one of these functions, it knows the type of the arguments, so it can attach this label to the data structure. Howard propagates these type labels, e.g., when a labelled structure is copied. In addition, it also propagates the type information to the pointers that point into the memory.

In our experience, the recovery works only for a limited set of all data structures, but some of these are important for debugging and forensics. For instance, it may be interesting to see which data structures contain IP addresses. We have implemented type sinks for libc functions and for system calls, as these are used by practically all applications. In general, Howard does not depend on type sinks, but it is capable of using them when they are available. All results presented in Section 6.8 (Evaluation) were obtained without turning on type sinking at all.

6.5 Comments and Limitations

Obviously, Howard is not flawless. In this section, we discuss some generic limitations we have identified.

Limitations in array detection  This section explains the limitations of our array detection techniques.

As we mentioned before, the main goal of Howard is to provide data structures that allow us to retrofit security onto existing binaries. BodyArmour (discussed in Chapter 7) protects binaries by instrumenting array accesses to make sure that they are safe from buffer overflows. The basic idea is that once a vulnerable program has used a pointer to access an array, BodyArmour precludes this pointer from accessing memory beyond the array boundaries. It is crucial that possible array misclassifications do not cause false positives. Thus in this section, we also discuss the impact of the limitations in array detection on BodyArmour.

- **At least four accesses.** To make array detection more accurate, Howard will not recognise an array if fewer than four of its elements are accessed in a loop. In that case, it will be classified as a structure. This misclassification can only cause false negatives, but never false positives. Indeed, if an array remains undetected, BodyArmour does not try to protect instructions accessing it.

- **Merging with unused bytes.** We merge detected arrays with any unused bytes that follow the array. Doing so prevents identification of buffers that are too small (and thus false positives in the case of protection against memory corruption). In general, we designed Howard so that BodyArmour always errs on the safe side. We prefer to overestimate than underestimate the size of a buffer. Observe that even if we would add unused bytes that do not belong
to the same buffer, we still protect the next used field or variable from buffer overflows.

• Incorrect merges. If a structure consists of one field followed by an array, and the field and array are accessed in sequence, it is impossible to classify it correctly solely based on memory access patterns. As Howard always extends arrays to include the first element unless it finds evidence that it does not match (e.g., if because size or type are different), it could lead to overestimation of the array length. Again, this can never cause BodyArmour to raise a false alert.

• Separate last element accesses. It may happen that all but the last elements of an array form a chain, while the last element is always accessed separately, e.g., when a string is first copied to a destination buffer, and then extended with EOL or NULL. Howard misclassifies the memory as a structure containing an array and a separate field. Even though the last element is not attached to the array, this does not cause false positives for our application. Indeed, if the program never exploits the connection between the last element and the remaining part of the array, it is not important for us to join them either. Also, if the size of the array varies across multiple runs of the program, Howard prefers to err on the safe side, merges the elements, and reports accurate results. In general, even if Howard cannot classify an array or structure correctly in one particular loop or function, it may still get it right eventually. Often data structures are accessed in more than one function, yielding multiple loops to analyse the layout.

Other limitations

• Howard cannot recognise nested structs if the inner struct is never accessed separately. In that case, Howard returns a single large structure. As the result is equivalent, we do not consider this a problem.

• Unions might exhibit more than one memory access pattern. As a result, Howard would report a data structure being a merge (or intersection) of the multiple structures included in the union. Howard might report an incorrect (more detailed) interpretation of fields.

• Howard gets confused by certain custom memory allocators. Specifically, it can handle slab-like allocators, but a generic, application-specific memory allocator (such as that of Apache) leads to misclassifications. It stems from the fact that a memory region serving the allocator exhibits mixed access patterns inherited from various different structures/arrays for which it was used. As a result, Howard would classify such buffer as either an array of (perhaps) 4-byte fields or a highly nested structure.
We are currently working on a generic solution to detect custom memory allocators. We perform a dynamic analysis to search for functions that split a buffer into multiple smaller chunks, and pass them to other parts of the program, but do not use this memory for their own computations. While it is too early to claim that the problem is solved, we believe that our analysis will increase the accuracy of Howard.

- Howard does not analyse local variables of functions reached using a jmp rather than a call.

6.6 Code Transformation: Compiler Optimisation and Obfuscation Techniques

In production code, compilers apply optimisations to improve runtime performance. Such optimisations may change the code substantially. Likewise, obfuscation tools change the source code or binary to reduce the ability to understand or reverse engineer the program. In this section, we introduce popular optimisations and obfuscation techniques and discuss how they influence Howard’s data structure detection results. We treat compiler optimisations in Section 6.6.1 and obfuscation techniques in Section 6.6.2. In both cases, we limit ourselves to the techniques that are relevant and that may affect data structure detection.

6.6.1 Compiler Optimisations

Howard detects data structures in gcc-generated x86 C binaries. Even though our techniques were not designed with any specific compiler in mind and should work with other binaries also, we conducted all our experiments on gcc-4.4 binaries. So we focus our discussion on gcc.

Data layout optimisations

Data layout optimisations [98; 97] adapt the layout of a data-structure to its access patterns in order to better utilise the cache by increasing spatial locality. They include structure splitting and field reordering transformations.

In general, Howard detects data structures at runtime, so the analysis results correspond to the optimised code and data—which may be different from what is specified in the source. This is known as WYSINWYX (What You See Is Not What You eXecute) [18] and while it complicates reverse engineering (for instance, some data structures may be reordered or transformed), analysing the code that really executes is of course ‘the right thing’. Without it, we would not be able to protect buffers from overflows, or perform proper forensics.
6.6. OPTIMISATION AND OBFUSCATION TECHNIQUES

Loop code:

\[
\text{for}(i = 0; i < 64; i++) \text{ arr1}[i] = i;
\]

And two possible executions:

\[
\text{addr = arr1;}
\]

\[
\text{for}(i = 0; i < 64; i++)\{
\text{addr = arr1;}
\text{*addr = i;}
\text{addr += 1;}
\}
\]

\[
\text{addr = arr1;}
\]

\[
\text{for}(i = 0; i < 16; i++)\{
\text{addr = arr1;}
\text{*addr = i;}
\text{addr += 1;}
\}
\]

Figure 6.7: An example loop and two possible ways in which the loop can be executed: the non-transformed one on the left hand side, and the unrolled one on the right hand side.

Loop optimisations

Loop transformations [4; 16] reduce loop overhead, increase instruction parallelism, and improve register, data cache or TLB locality. Popular transformations include:

1. **Loop unrolling**, where the body of a loop is replicated multiple times in order to decrease both the number of loop condition tests and the number of jumps,
2. **Loop peeling**, where a loop is peeled, a small number of iterations are removed from the beginning or end of the loop and executed separately to remove dependencies created by the first or last few loop iterations,
3. **Loop blocking**, where a loop is reorganised to iterate over blocks of data sized to fit in the cache.

As described in Section 6.4.4, Howard recovers arrays when the program accesses them in loops. To increase the accuracy of the analysis, Howard’s algorithm allows for arrays accessed in multiple loops, and checks for array elements accessed before or after the loop body. Basically, when a transformed loop accesses an array, all its elements are **classified together**.

However, loop transformations may change not only the layout and number of loops, but also the memory access patterns. As an example, Figure 6.7 presents a simple loop accessing an array, and two possible ways in which this loop can be executed. Refer to Figure 6.8 to observe array `arr1` access patterns for both executions. We can see that depending on the execution, the array is either classified as an array of single fields (as desired) or as an array of 4-field structures. Even though `arr1` is possibly misclassified here, it might be used in other ways somewhere else in the program, and we might eventually get it right. In all similar cases, Howard cannot do anything about not entirely accurate classifications, as its analysis is based solely on memory access patterns.

Optimisations affecting function frame

There are numerous compiler optimisations which affect the way we perceive a function’s frame. First, functions can be inlined, which means that they are integrated into their callers. In this case the small inlined function is not called separately, but
its function frame extends the caller’s one. Second, (some of) the input function arguments might be passed through the registers and not through the stack. Also, gcc might analyse the program to determine when values passed to functions are constants. These are optimised accordingly.

Howard is expected to reflect functions in the way they appear in the binary. In the case of inlined functions we get just one extended function frame. Likewise, since Howard analyses memory access patterns only, it cannot spot function parameters passed through the registers or precomputed by the compiler. As this inaccuracy does not affect our applications, we did not worry about it.

6.6.2 Obfuscation Techniques

Code obfuscation techniques aim to reduce the ability to understand or reverse engineer the program. Data transformations [55] that obscure data structures used in the source application are the most relevant to Howard, and we focus our discussion on them. We also briefly mention anti-dynamic analysis techniques.

Obfuscating arrays

Numerous transformations can be devised for obscuring operations performed on arrays [48; 79]. Popular techniques include: (1) array splitting, where an array is split into several subarrays, (2) array merging, where two or more arrays are merged into one array, (3) array folding and flattening, where the number of dimensions is increased or decreased, respectively.

As before, Howard is expected to reflect arrays in the way they appear and are accessed in the binary. In the case of split, folded or flattened arrays, Howard is supposed to report the new transformed data structures. When two or more arrays are merged to form a new array, the results of the analysis greatly depend on the merging method. For example, one could combine arrays arr1[N1] and arr2[N2] in such a way that the new array arr3 contains N1 elements of arr1 followed by N2 elements of arr2. In this case, it is very probable that Howard reports arr3 as a structure containing two arrays: an N1- and an N2-element long. Another merging
6.6. OPTIMISATION AND OBFUSCATION TECHNIQUES

A technique could interleave \texttt{arr1}'s elements with \texttt{arr2}'s elements. Here, we perhaps expect Howard to report one array of \([N1+N2]\) elements.

Since Howard analyses memory access patterns only, it cannot recognise that certain arrays are used in similar ways by similar functions, and containing similar elements. Thus, it cannot say that some arrays might share a higher level data type. Howard concerns itself with low level data structures only, and limits itself to recognising sizes and basic types of data structures. Understanding contents and usage scenarios is part of our future work.

Obfuscating variables

There is a lot of work on hiding values of sensitive variables from static and dynamic analysis [50; 223]. In the current section, we briefly explain three popular techniques: (1) substitution of static data with code, (2) variables merge, and (3) variable split. A comprehensive overview is offered by Collberg et al. [55].

The simplest technique is to substitute static data with code. A popular approach to obfuscate a static string is to convert it to a procedure that generates its value dynamically. To add to the confusion, this code could also generate other extraneous strings.

Like arrays, variables may be merged or split. For example, two 32-bit integers, \(x\) and \(y\), can be merged into a 64-bit long variable, \(z\). If we use the formula \(z = 2^{32}x + y\), then \(x\) occupies the first half, and \(y\) the second half of the 64-bit variable \(z\). Alternatively, an obfuscation can split a variable into two or more variables. An obvious example is a popular implementation of 64-bit integer variables on 32-bit systems. Two 32-bit variables, \(x\) and \(y\), are used to encode the 64-bit information, \(z\). The encoding applied is again provided by the formula \(z = 2^{32}x + y\).

Even though none of our test applications contain such obfuscations, our current research investigates whether it is possible to detect that variables in a program are obfuscated, and further try to understand their original structure. We discuss it in Section 6.6.3.

Anti-dynamic analysis techniques

Howard observes the behaviour of a program in runtime, and so it is crucial that the program does not refuse to run in the instrumented processor emulator. However, there exist techniques that could be used by programs running in virtualised environments to determine that they are running in a virtual machine rather than on native hardware [165; 89]. When this happens, Howard cannot perform its analysis.

6.6.3 Ongoing Work

A variety of tools exists to perform or assist with C/C++ code obfuscation. The majority of commercial obfuscation solutions work by transforming source code, e.g.,
Stunnix \(^3\), or the framework from Semantic Designs \(^4\). However, software vendors have also a compiler driven obfuscation solution, Morpher \(^5\), at their disposal, or a multi-layer defence approach from Irdeto \(^6\). Since more and more binaries are already or will be obfuscated in the future, it is important to investigate whether it is possible to (1) detect that a given binary obfuscation technique has been applied, and (2) deobfuscate it in order to understand the intended meaning of a data structure.

In our current research, we have been looking into two popular variable obfuscation techniques \([22]\): static to procedural data hiding, and variable split. (Refer also to Section 6.6.2.) Since many commercial obfuscators use opaque predicates to obfuscate the control flow of a program, static analysis might be very hard. For this reason our tool performs dynamic analysis – we run an obfuscated binary in an instrumentation framework, and observe its behaviour. In this section, we highlight the main ideas behind our system. For a detailed description refer to \([22]\).

### Substitution of static data with code

To discover that *static data has been substituted with code*, we search for memory buffers that have a constant value at the moment of usage. For the purpose of this analysis, we say that a variable is a constant if only deterministic values are involved in its computation, i.e., we do not allow for data obtained through system calls (e.g., `read` or `recv`). Additionally, we demand that the value of a variable is consistent across multiple runs of the program.

### Variables split

Before we delve into our algorithm to detect *split variables*, we make an observation which forms a foundation for our method. When a variable \(z\) is split into \(x\) and \(y\), then for each operation (read or write) performed on the original variable \(z\), a semantically equivalent operation must be performed on both variables \(x\) and \(y\). Also, since \(x\) and \(y\) originate in a single variable, their values ought to be combined during some computations, e.g., during an interaction with components of the system that were not obfuscated, such as the operating system.

To detect split variables, we analyse a program memory access trace, and search for variables that are used together, and exchange their data locally, i.e., in a short time interval. Our algorithm employs a model of *reference affinity grouping* proposed by Zhong et al. \([222]\) for cache optimisations. The optimisation was done by restructuring arrays and data structures in order to place elements that are always used together on the same cache line. The model measures how close a group of data are accessed together in a trace, and based on that gives a partition of a program data. In our case, split variables, which are used close together, are grouped

---

\(^3\)www.stunnix.com  
\(^4\)http://www.semdesigns.com/products/obfuscators/CObfuscat.png  
\(^5\)www.morpher.com  
\(^6\)http://irdeto.com/en/application-security.html
together by the optimisation algorithm. Once the algorithm proposes candidates for split variables, we further verify them to minimise false positives. Since the model selects variables by examining only the memory access trace, we additionally check for a dataflow between elements that might form a split.

**Results**

While it is too early to claim that the problem is solved, the results are promising. Our evaluation suggests that the obfuscation techniques are vulnerable to the dynamic analysis we have proposed. The number of false positives is low, making our tool a practical method of detecting that either a static data has been hidden or a variable split obfuscation has been applied.

## 6.7 Applications

The main goal of Howard is to provide data structures that allow us to retrofit security onto existing binaries (see Chapter 7). In this section, we describe two extra applications: binary analysis with reconstructed symbol tables (Section 6.7.1), and reverse engineering high-level data structures (Section 6.7.2). These are not the only possible applications (e.g., we could also use Howard for malware classification [65]), but they are examples of applications that are impossible with any existing technique.

### 6.7.1 Binary Analysis with Reconstructed Symbols

To aid forensics and reverse engineering, Howard automatically generates new debug symbol tables for stripped binaries. In this example, we focus primarily on the new techniques for detection of data structures. However, we also want to show how we complement Rewards’ method of recognising well-known functions (type sinks) and propagating the types of their arguments [127]. For this reason, we also add a minimal set of type sinks.

The symbol tables Howard generates are generic and we can use them with any common UNIX tool. Of course, they are not entirely complete. For instance, we only have exact semantics for the subset of data structures that derive from type sinks. Also, we cannot generate out of thin air the correct names of fields and variables. Nevertheless, the recovered symbol table allows us to analyse the data structures of applications that were practically impossible to analyse otherwise.

Figure 6.9 shows a screenshot of a real `gdb` session with reconstructed symbols. Suppose we have purchased a program that crashes occasionally. We want to analyse the program and perhaps reverse engineer parts of it. Without Howard, common disassemblers (like those of `gdb` or `IDA Pro`) help to analyse the instructions, but not the data structures. In this example, we show that we can now analyse the data also. For demonstration purposes, we use a stripped version of `wget` as our demo binary, and show that we can analyse it with the `gdb` debugger.
To show both the unadorned data structure recovery and recovery of semantics, this example uses a truly minimal set of type sinks. Specifically, our type sinks consist (only) of `inet_addr()` and `gethostbyname()`.

The scenario is that we have witnessed a crash. Thus, we start our analysis with an instruction prior to the crash (0x805adb0) and try to find information about the variables in the scope of the current function (info scope) and then print variables. Where the stripped binary normally does not have any information about the variables, we see that Howard reconstructed most of them (structs, pointers, and strings) and even recovered partial semantics. For instance, we find pointers to a `struct hostent` and an `inetaddr_string`.

We could print out the contents of the `struct hostent`, but really we are interested in the data structures pointed to by the various pointers – for instance the pointer to a `struct identified by pointer_struct_1_0`. Unfortunately, it currently has the value NULL, so there is nothing interesting to see and we have to wait until it changes. We do this by setting a watch point on the pointer.

---

7Names are generated by Howard. Prefixes like `field_`, `pointer_` are for convenience and are not important for this discussion.
Once the pointer changes, and receives its new value 0x80b2678, we can analyse the corresponding memory area. We see that it points to a structure containing an integer with value 3, a pointer to a struct, a one-byte field, and a four-byte field. If we follow the pointer, we see that it points to a struct with two fields, one of four bytes and one of type `in_addr_t`. We can even examine the value of the IP address and see that it corresponds to 74.125.77.147.

Moreover, we see that our struct is in an array of size 3 by dividing the amount of memory allocated (malloc_usable_size) by the size of an element. Thus, we can make an educated guess that the integer value of 3 we found earlier denotes the number of IP addresses in the array.

To complete the admittedly simple example, we also print out these IP addresses. For reasons of space, we stop the scenario here. The purpose of the example is merely to show that we can debug and analyse stripped binaries in the same way as when debugging and analysing binaries with debugging symbols.

We emphasise that by means of type sinks alone (e.g., using Rewards [127]), we would have obtained hardly any data structures as almost all of them are internal to wget. Eventually, the IP addresses are used, so they would appear, but no struct or array would have been found. In addition, connecting the IP addresses to other structures from our entry point into the code (the break point) would have been completely impossible. In contrast, with Howard we can progressively dig deeper and trivially find the connections between the data structures.

### 6.7.2 Recovering High-level Data Structures

In order to gain more information about the binary being analysed, our current research [175] addresses the issue of lifting the low level analysis performed by Howard to higher level data structures. We observe connections between structures, and based on that information reason about pointer structures like linked lists, trees, and balanced trees.

The idea behind our approach is that the shape of a linked data structure reveals the functionality of an object implemented in the binary. For example, if a bunch of heap objects is shaped like an ordinary tree, then the binary might be using a tree data structure. Further, if the shape of the tree is balanced, then the binary might be using a more complex data structure, e.g., a red-black tree.

Even though the current implementation of the system has some limitations, we managed to correctly identify a number of complex data structures in real world programs and libraries. Below we sketch the approach overview, some preliminary results and the main limitations. For more details refer to [175].

### Approach Overview

In a nutshell, the shape recovery procedure is split into the following four steps, all performed at runtime.
1. **Heap graph creation** — we observe objects allocated on the heap, and we track how they get linked together through pointers. The *heap graph*, which can be technically a collection of graphs, records connections between all heap objects.

2. **Heap graph partitioning** — using the information found by Howard, we assign types to heap objects, and we group together linked objects of the same type. These are the candidates for further shape analysis.

3. **Shape detection** — we check whether groups of objects of the same type form one of the modelled shapes.

4. **Shape tracking** — once a structure has been classified as, for example, a tree, we further monitor it to make sure that the shape is preserved throughout the application execution.

**Evaluation**

To verify the accuracy of our approach, we tested it on both a number of small programs using well-known libraries that implement trees and lists, and few real world applications with their own implementations of high-level data structures.

The tested libraries include `libdict`, which contains e.g., AVL and red-black trees, and `glib-2.10` with doubly-linked lists and AVL trees. We also run our analysis on the `sloopy-0.1.1` chess engine, the `ex-050325` line editor for Unix systems, and `tr` and `tsort` from `core-utils-8.9`. In all these cases, we were able to correctly reverse engineer complex data structures, including balanced, and unbalanced trees, and lists.

While most of the high-level data structures used by tested programs were analysed correctly, few were also missed. For example, a balanced tree in `tsort` was misclassified as an unbalanced one. That balanced tree contains a sentinel node at the top of the structure, and our algorithm wrongly evaluated maximum and minimum paths.

**Limitations**

The system is certainly not flawless. We discuss now the main limitations.

**Deviations from model** In our analysis, we observe whether the structure formed by linked objects resembles a known shape. In the current version, we do not allow for serious deviations from the model. In consequence, threaded trees, for example, are identified as plain graphs. Right children pointing to the inorder successor in the tree instead of `null` are considered abnormal. Another example of a scenario we cannot currently handle are sentinel nodes. A common practice in implementing high-level data structures is adding an extra sentinel or dummy node at the top or at
the bottom of a tree or a list. Depending on sentinel nodes’ location, a tree is either classified as a graph, or a balanced tree is classified as an unbalanced one.

**Shape only** Our algorithm tracks the shape of a structure only, so it will miss all subtleties that require an understanding of the contents of an object. For example, we cannot tell a red-black tree from an AVL tree. This particular example could be perhaps handled by analysing operations performed on a tree, but we leave it as a future work.

### 6.8 Evaluation

Howard can analyse any application on the Linux guest on our (QEMU-based) emulator. For instance, we successfully applied Howard to games like glines and gnometris, but also to complex binaries like the Linux loader ld-2.9.so, and huge ones like Apache\(^8\). However, for good results we need code coverage and for code coverage we currently depend on what is supported by KLEE [39] and existing test suites.

Our experimental setup is shown in Figure 6.1. We conduct the code coverage runs offline, using KLEE on LLVM version 2.6 with home-grown modifications to handle networking. All applications run on a Linux-2.6.31-19 kernel. We then use klee-replay to replay the application on top of our Howard analyser with the inputs generated by KLEE. If our coverage is low, we use normal test suites for the applications.

Starting out with small applications (fortune), we worked towards progressively larger applications. In the plots below, we include results for programs of several tens of thousands LoC, including the wget download program and the lighttpd high-performance web server. Table 6.1 shows the applications and the coverage we obtained with KLEE and the test suites. We also applied Howard to utilities in Core-Utils, but these tend to be so small (a few hundred lines, typically) that they are not representative of real applications, and we do not discuss them here.

**Results** To verify Howard’s accuracy, we compare the results to the actual data structures in the programs. This is not entirely trivial. We cannot compare to the original source code since aggressive compiler optimisations may change the binary significantly (“what you see is not what you execute” [18]). Thus, all the results presented in this section were obtained for binaries for which we could also generate symbol tables to compare our results with. This way we were able to get ground truth for real world applications.

We will start with a few quick observations. First, Howard cannot discover variables that always remain unused, but this probably should not count as a ‘missed’

\(^8\) 244,000 lines of code (LoC), as reported by David Wheeler’s sloccount (www.dwheeler.com/sloccount/).
Table 6.1: Applications analysed with Howard. LoC indicates the lines of code according to sloc. Size is the approximate size of the text segment. Func% is the fraction of functions that the tests exercised (KLEE or test suite). Vars% is the fraction of variables used in the tests (KLEE or test suite), and KLEE% is the coverage offered by KLEE (if any).

<table>
<thead>
<tr>
<th>Prog</th>
<th>LoC</th>
<th>Size</th>
<th>Funcs%</th>
<th>Vars%</th>
<th>How tested?</th>
<th>KLEE%</th>
</tr>
</thead>
<tbody>
<tr>
<td>wget</td>
<td>46K</td>
<td>200 KB</td>
<td>298/576 (51%)</td>
<td>1620/2905 (56%)</td>
<td>KLEE + test suite</td>
<td>24%</td>
</tr>
<tr>
<td>fortune</td>
<td>2K</td>
<td>15 KB</td>
<td>20/28 (71%)</td>
<td>87/113 (77%)</td>
<td>test suite</td>
<td>N/A</td>
</tr>
<tr>
<td>grep</td>
<td>24K</td>
<td>100 KB</td>
<td>891/79 (50%)</td>
<td>609/1082 (56%)</td>
<td>KLEE</td>
<td>46%</td>
</tr>
<tr>
<td>gzip</td>
<td>21K</td>
<td>40 KB</td>
<td>74/105 (70%)</td>
<td>352/436 (81%)</td>
<td>KLEE</td>
<td>54%</td>
</tr>
<tr>
<td>lighttpd</td>
<td>21K</td>
<td>130 KB</td>
<td>199/360 (55%)</td>
<td>883/1418 (62%)</td>
<td>test suite</td>
<td>N/A</td>
</tr>
</tbody>
</table>

data structure. Second, all these results were obtained solely with Howard’s ‘core’ data structure excavation techniques. In other words, we turned off all type sinking for these tests.

Figure 6.10 presents our results in terms of accuracy for both the stack and the heap. The accuracy is calculated both for the number of variables, and for the total number of allocated bytes. In the case of stack memory we evaluated the accuracy of Howard’s analysis for all functions used during our experiments. Thus we did not count the stack variables used by functions that were never invoked. In the case of heap, we simply considered all allocated memory regions. The plots show Howard’s results in five categories:

- **OK**: Howard identified the entire data structure correctly (i.e., a correctly identified structure field is not counted separately).

- **Flattened**: fields of a nested structure are counted as a normal field of the outer structure. On the stack these are usually structures accessed via EBP, as explained in Section 6.3.

- **Missed**: Howard misclassified the data structure.

- **Unused**: single fields, variables, or entire structures that were never accessed during our tests.

- **Unused array**: this is a separate category that counts all arrays not recognised because there were insufficient accesses to the array.

The general conclusion is that practically all memory is either detected correctly or unused. Moreover, for practically all arrays and structs the lengths were identified correctly.

The stack has more unused memory than the heap. This is expected. Whenever a function is called, the whole frame is allocated on the stack, regardless of the execution path taken in the function’s code. Heap memory is usually allocated on demand. Much of the unused bytes are due to the limited coverage. As we never ran
6.8. EVALUATION

Figure 6.10: The accuracy of the analysis per application. For each application the first bar denotes the accuracy in number of variables, and the second in number of bytes.

KLEE for longer than a few hours, it may be that by running it longer\(^9\) we obtain better coverage.

The stack also counts more structures that are flattened. Again, this is not surprising. As explained in Section 6.3, structure field addresses may be calculated relative to the beginning of the function frame. In that case, Howard has no means of classifying the region as a structure. On the heap this is less common. Indeed, if an application allocates memory for a structure, it refers to the structure’s fields from the base of the structure, i.e., the beginning of the allocated region. However, Howard may still misclassify it in the case of a nested structure. This happens for instance in fortune, where a 88-byte FILEDESC structure contains a nested 24-byte STRFILE structure.

The main source of missed are data structures occasionally classified as arrays of 4-byte fields. Assume that an application uses a simple structure with 4-byte fields based at the beginning of the structure (i.e., no inner structures). If none of the fields is 1-byte long, nor is a pointer, while a memset function is used for the structure initialisation, we have no means for discarding the array hypothesis.

As the vast majority of structs and arrays reside on the heap, we zoom in on these results for more details. Figure 6.11 breaks down the overall results to show how well Howard discovers individual fields and different types of arrays. The two plots correspond to structures that are separately allocated on the heap and on arrays (possibly containing structures). We label the results as follows:

- **Structures**
  - **OK**: fields correctly identified.
  - **Missed**: fields incorrectly identified.
  - **Unused**: fields missed because the program did not use them.
  - **Flattened**: fields in nested structures that Howard placed at the wrong nesting level.

\(^9\)Or running it better. Driving KLEE down the right execution path is not always trivial.
Figure 6.11: The accuracy of the analysis per application. For each application the first bar denotes the accuracy in number of variables, and the second in number of bytes.

- Arrays
  - **OK**: arrays correctly identified.
  - **Missed**: arrays incorrectly identified.
  - **Missed short**: arrays missed because they were accessed fewer than 4 times.
  - **Unused**: arrays missed by Howard because they were not used at all.
  - **Flattened**: fields of structs in arrays classified as separate elements.

If an array contains structs and any field of any struct is missed, we count it as a missed array. A flattened array means at least one of the structs was flattened. In other words, we analyse structs that were allocated individually separately from structs in arrays. The reason for doing so is that otherwise the arrays would completely dominate the analysis of structs. For instance, if all fields in an array of 1000 structures were classified correctly, this would artificially inflate the number of correctly identified fields.

In the plots in Figure 6.11 we see that Howard mostly detects fields and structures quite accurately, provided they are used at all. Most of the misclassifications are also minor. For instance, the flattened structure in fortune consists of an 88-byte FILEDESC structure that contains a nested 24-byte STRFILE structure, where the inner structure is never used separately. Examples of the arrays missed in wget are a char* array that is reallocated and then not used anymore. Howard classifies it as an int* array. There are also misclassifications when the last element is accessed outside the loop and relative to the base pointer, rather than the previous element. For grep, some arrays of structs were classified as arrays of individual fields. As explained above, we decided to count them in the flattened arrays, rather than in the structures. Also, for grep (and gzip) no structures appear on the heap except in arrays (hence the lack of data in the plot on the left).
6.9. CONCLUSIONS

**Performance.** Most of the overhead in Howard is due to code coverage. It takes several hours to obtain reasonable coverage of real-life applications. Howard is considerably cheaper. Still, since we either re-run the KLEE experiments or the test suites on a heavily instrumented binary, the analysis takes more time than running the applications natively. For instance, programs like `gzip` and `grep` took 5 and 15 minutes to analyse, respectively. Of all the applications we tried, `grep` took the longest. As Howard performs a once-only, offline analysis, these results show that even with the current unoptimised version, we can realistically extract data structures from real applications.

### 6.9 Conclusions

We have described a new technique, known as Howard, for extracting data structures from binaries dynamically without access to source code or symbol tables by observing how program access memory during execution. We have shown that the extracted data structures can be used for analysing and reverse engineering of binaries that previously could not be analysed. As until now data structure extraction for C binaries was not possible, we expect Howard to be valuable for the fields of debugging, reverse engineering, and security.

**Future work** Future research addresses the issue of further lifting the current low level analysis to higher level data structures. In particular, we also intend to monitor operations performed on identified structures, and try to classify, for example, insert, delete, or rotate operations. This would leave the door open for more advanced reverse engineering, such as detecting various sorting algorithms.

Another branch of our current and future research addresses code and data obfuscation techniques. An important question is to what extent we can realise that a particular obfuscation technique was applied, so that we can take it into account during the reverse engineering procedure.

Finally, to increase the accuracy of the analysis, we are currently working on a generic solution to detect custom memory allocators.
Chapter 7

BodyArmour for Binaries

7.1 Introduction

In this chapter, we describe BodyArmour, a novel technique to protect existing C binaries from memory corruption attacks that modify either the control flow of the program, or non-control data. As we have explained already, non-control data modifications are particularly worrying, as they are increasingly common [191], and no existing solutions protect binaries against such attacks.

All reliable defences against non-control data attacks require access to the source code [104; 6; 7]. Existing security measures at the binary level are good at stopping control-flow diversions [62; 149; 3; 113; 85], but powerless against corruption of non-control data.

To address these problems, we have designed and built BodyArmour, a tool chain to bolt a layer of protection on existing C binaries to shield them from state-of-the-art buffer overflow attacks.

High level overview  Rather than patching systems after a vulnerability is found, BodyArmour takes a proactive approach. It stops memory corruption attacks in binary software, before we even know it is vulnerable. Specifically, it mitigates attacks by transforming them from Byzantine to fail-stop.

The worst compromise of a system is Byzantine: a good node goes bad—either because attackers divert the program’s control flow [84], or because they manipulate non-control data [45]. Malicious code can actively evade detection, propagate the infection, and corrupt more data. A posteriori detection is difficult. In contrast, a fail-stop failure (a crash) is much simpler. Since no data is corrupted, it is often sufficient to restart the faulting program. Thus, BodyArmour aims to push Byzantine failures to fail-stop crashes, assuming that a (rare) crash is better than a compromised system.

To protect the binary, BodyArmour relies on limited information about the program’s data structures—specifically the buffers that it should protect from overflow-
It can be set to run in two modes, (1) **BA-fields mode**, when it protects individual fields inside a structure, and (2) **BA-objects mode**, when the protection is slightly more coarse-grained, and limited to complete structures.

If the program’s symbol tables are available, BodyArmour is able to protect the binary against buffer overflows with great precision. In the BA-objects mode, no false positives are possible. While we cannot guarantee no false positives in the BA-fields mode, in practice we did not encounter any, and as we will discuss later, they are highly unlikely.

Assuming the presence of symbol tables is fairly common in the security research community [85; 99; 146; 176]. In practice, however, commercial software vendors often strip their programs of all debug symbols. If this is the case, BodyArmour uses reverse engineering techniques to recover the data structures. To our knowledge, BodyArmour is the first system to use data structure recovery to prevent memory corruption. However, we believe that the approach is quite promising and may benefit other systems, like XFI [85] and memory debuggers [99; 146], also.

BodyArmour hardens C binaries in three steps (see also Figure 7.1):

(i) **Data structure discovery**: track down the sensitive data structures (buffers) that need protection.

(ii) **Array access discovery**: find all potentially unsafe accesses (pointer dereferences) to these buffers.

(iii) **Rewrite**: rewrite the binary to ensure that a pointer accessing a buffer B never strays beyond B’s boundaries.

Data structure discovery is easy when symbol tables are available, but very hard when they are not. In the absence of symbol tables, BodyArmour extracts the data structures from the binary itself by means of static [167; 20; 21] or dynamic analysis [188]. As far as BodyArmour is concerned, the approaches are complementary. However, since current static analysis techniques are weak in array detection, we focus on a dynamic approach in this work. Specifically, BodyArmour discovers buffers (step i) by means of Howard (see Chapter 6), a novel analysis technique called dynamic data excavation, which follows the simple intuition that memory access patterns reveal much about the layout of data structures. Something is a structure field,
if it is accessed like a structure field, and an array, if it is accessed like an array. And so on.

Next, BodyArmour dynamically detects array accesses (step ii). Finally, in the rewrite stage (step iii), it takes the data structures and the accesses to the buffers, and assigns to each buffer a unique colour. Every pointer used to access the buffer obtains the colour of the buffer. BodyArmour raises an alert whenever, say, a blue pointer accesses a red byte.

**Contributions** A high-level contribution of BodyArmour is that it proactively protects existing binaries—before we even know whether the code is vulnerable. A low-level contribution is that we protect the binary at sub-field granularity from attacks on control data and non-control data.

Our method is quite different from existing techniques to protect binaries, so for the technical contributions, it is also useful to compare BodyArmour against the most closely related approaches: compiler enhancements like WIT [6] which require source code, and binary instrumentation techniques like dynamic taint analysis [62; 149], which do not.

Compared to source-level protection like WIT, BodyArmour has the advantage that it requires no access to source code at all. Very often sources and even symbols are not available. In addition, in the BA-fields mode, by protecting individual fields inside a structure rather than aggregates, BodyArmour is significantly finer-grained than WIT and similar solutions. Also, it prevents overflows on both writes and reads (WIT protects only writes and permits information leakage). The drawback of BodyArmour is that it may be less accurate, because dynamic analysis may not cover the entire program.

Compared to techniques like taint analysis that also protect the binary, BodyArmour detects attacks immediately when they occur, rather than sometime later, when a function pointer is used. Also, BodyArmour detects both control flow and non-control flow attacks, whereas taint analysis only detects the former (see Section 3.3). Finally, performance of BodyArmour is much better than that of taint analysis, e.g. the slowdown for gzip is only 1.7x.

In Section 7.8, we compare BodyArmour against pointer taintedness [44] and other approaches like control flow integrity [3; 113], data flow integrity [40], XFI [85], and baggy bounds checking [7].

The focus in this work is on dynamic analysis, and because of this, we also suffer from the limitations of any dynamic approach: we can only protect what we execute during the analysis. This work is not about code coverage techniques. We rely on existing tools and test suites to cover as much of the application as possible. Since coverage is never perfect, we may miss buffer accesses and thus incur false negatives. Even so, BodyArmour detected all real buffer overflow attacks in our evaluation, which demonstrates that BodyArmour is practical. In analogy to real body armour for soldiers: BodyArmour may not cover all parts of the (program)
body, but greatly improves its security in practice. We show in Section 7.7 that it stops all the attacks we have tested.

BodyArmour takes a conservative approach to prevent false positives (unnecessary program crashes), even though our target domain by design consists of systems that can handle rare crashes. No false positives are possible when the protection is limited to structures (the BA-objects mode). As we will see later, due to the limited code coverage, we can theoretically come up with scenarios leading to false positives in the BA-fields mode. However, we did not encounter any, and in Section 7.5.5, we will see that they are very unlikely.

Since our dynamic analysis builds on the Qemu [25] process emulation which is only available for Linux, we target x86 Linux binaries, generated by gcc (various versions and different levels of optimisation). However, there is nothing fundamental about this and the techniques should apply to other operating systems also.

While the idea and the architecture of BodyArmour were designed by me, the implementation of the instrumentation (only) was carried out by Traian Stancescu [194].

Outline In this chapter, we first explain which memory accesses are protected by BodyArmour (Section 7.2), and what sort of problems stemming from limited code coverage ought to be expected and addressed (Section 7.3). Sections 7.4–7.6 implement fine-grained protection. We evaluate BodyArmour in Section 7.7, and highlight the most related approaches in Section 7.8. Finally, we discuss some generic limitation of our work in Section 7.9, and conclude in Section 7.10.

7.2 What to Protect: Arrays and Array Accesses

BodyArmour protects binaries by instrumenting array accesses to make sure they are safe from buffer overflows. Figure 7.1 illustrates the general idea, which is intuitively simple: once the program has used a pointer to access an array, it should not use the same pointer to access elements beyond the array boundaries. For this purpose, BodyArmour assigns colours to arrays and pointers and verifies that the colours of the memory and of the pointer match on each access. After rewriting the binary, the resulting code runs natively and incurs overhead only for the instructions that access arrays. In the remainder of this Section 7.2, we explain how we obtain the arrays and accesses to arrays in case no symbol table is available, while Sections 7.4–7.6 discuss how we use this information to implement fine-grained protection against memory corruption attacks.

7.2.1 Extracting Arrays and Data Structures

Ideally, BodyArmour obtains information about arrays from the binary’s symbol tables. The availability of symbol tables is a fairly common assumption in security research [85; 176]. If the binary comes with symbols, BodyArmour is able to offer
very accurate protection. As such information is often stripped off in commercial software, BodyArmour is also capable of extracting it from the binary itself, by means of automated reverse engineering.

To find the arrays, BodyArmour can use either static analysis (e.g., IDA Pro’s Hexrays plug-in [75; 19]), or dynamic analysis (e.g., the Howard dynamic data excavator discussed in Chapter 6). BodyArmour is indifferent to which method is used to extract the data structures, but in practice accurate static analysis is still not fully solved. Moreover, current static techniques are weak at extracting arrays. Dynamic approaches find more buffers and the overall results for BodyArmour are better. For brevity, therefore, we limit our description to dynamic analysis in the remainder of this chapter.

BodyArmour recovers data structures using Howard, discussed in detail in Chapter 6. Since arrays are detected dynamically, we must make every effort not to underestimate the size of arrays, which could lead to false positives.

In Howard, the data structure extraction is deliberately conservative, in the sense that the size of arrays is either classified exactly right, or overestimated (which will never lead to false positives). The reason is that we conservatively extend arrays towards the next variable below or above. It is very unlikely that Howard underestimates the array size for compiler generated code and we have never encountered it in practice (although there is no guarantee that we never will). It can happen only if the program accesses the array with multiple base pointers, and behaves consistently and radically different in all analysis runs than in the production run. It never occurred in practice. The details are complex and tedious and beyond the scope of this chapter. Instead, we discussed our experiences with real software in Section 6.8. Of course, doing so serves only as anecdotal ‘evidence’, rather than hard proof.

We stress again that we designed the array extractor in Howard to err on the safe side. Overestimating the size of arrays prevents false positives—classifying innocuous code that fills an array as a buffer overflow and crashing. One may argue that the drawback of over-estimation will be an increase in false negatives. While true in theory, this is hardly an issue in practice. The reason is that Howard only misses buffer overflows that (1) overwrite values immediately following the real array size (no byte beyond the overestimated array size is vulnerable), and (2) that overwrite a value that the program did not use separately during the dynamic analysis of the program (otherwise, we would not have classified it as part of the array). Exploitable buffer overflows that satisfy both conditions are rare.

We conclude this section with a remark on the issue of false positives in BodyArmour caused by a possible underestimation of the size of a buffer. Since BodyArmour is indifferent to which method is used to extract data structures, the problem of correct buffer sizes, while of primary importance, is orthogonal to the binary protection mechanism offered by BodyArmour. For that reason, whenever we discuss the issue of false positives in BodyArmour, we always assume that the sizes of buffers are not underestimated.
7.2.2 Array Accesses and Pointer Manipulations

When Howard detects arrays to be protected, it figures out which instructions, i.e., array accesses, need to be instrumented. Discovery of array accesses also occurs dynamically. The process is straightforward. Once Howard has identified an array $A$, it dumps all instructions that access $A$.

Alongside array accesses, Howard also dumps all instructions that initialise or manipulate pointers that access arrays. As we have mentioned already, the general idea behind BodyArmour is that it makes sure that once a pointer has been assigned to an array, it never accesses memory beyond that array’s boundaries. It means that BodyArmour needs to not only monitor array accesses, but it must also propagate information about the memory object associated with a given pointer on, for example, pointer copy operations.

Knowing that an instruction accesses an array is important, because it means that BodyArmour should instrument that instruction at runtime. Knowing which array it accesses is also important, for two reasons. First, if the access is always at a constant offset from the base of the array and within the array boundaries, there is no need to instrument it after all. The current implementation of BodyArmour does not yet include this optimisation.

7.3 Code-coverage and Problems to be Expected

Since BodyArmour is based on dynamic analysis, it also suffers from coverage issues—we can only analyse what we execute. Even the most advanced code coverage tools \cite{39; 47} cover just a limited part of real applications. Lack of coverage may cause BodyArmour to miss arrays and array accesses and thus lead to false negatives. While unfortunate, BodyArmour is powerful enough to detect all the real attacks we tried (see Section 7.7). What we really want to avoid are false positives: crashes on benign input for no good reason.

In general terms, BodyArmour prevents buffer overflows by assigning a colour to each array, and making sure that a pointer to an array of colour $X$ never accesses memory of a colour $Y$. To achieve this, we first assign the colour of an array to the colour of a pointer on pointer initialisation instructions. Later, we monitor array accesses to make sure that the colour of an array being accessed matches the colour of a dereferenced pointer. Also, as we mentioned in Section 7.2.2, we propagate the colour of a pointer on, for example, pointer copy instructions.

In BodyArmour, we instrument only those instructions which were seen by Howard during the analysis. It means that a program path executed at runtime, $p_R$, may differ from all paths seen before by Howard, \{$p_h\}_{h \in H}$, yet $p_R$ might share parts with (some of) them. In principle, we must anticipate that an arbitrary subset of array accesses and pointer manipulations on $p_R$ is instrumented. Indeed, we instrument exactly those instructions that belong to a path in the \{$p_h\}_{h \in H}$ set, so it can
happen that we miss a pointer copy, a pointer initialisation or a pointer dereference instruction.

With that in mind, ideally we would like to limit the colour checks performed by BodyArmour to program paths which manipulate and use array pointers in ways which were seen before. Since the architecture should also incur as little false negatives as possible, and be efficient in terms of performance, we propose two different modes of operation. Depending on the target application requirements, either the BA-objects or the BA-fields mode can be chosen. The BA-objects mode protects objects at a coarser granularity than the BA-fields mode, but false positives are ruled out. As for the BA-fields mode, we will see in Section 7.5.6 that complex scenarios where false positives can occur are theoretically possible. We have never encountered them in practice though. In the following two sections (Sections 7.4 and 7.5), we describe both modes in detail.

7.4 **BA-objects mode**: an Object-level Protection

Just like other popular approaches, e.g., WIT [6] and BBC [7], the BA-objects mode provides protection at the level of objects used by a program in C. To achieve that, BodyArmour assigns a colour to each local function variable, each global variable and each object allocated on the heap, and makes sure that a pointer to an object of colour X never accesses memory of colour Y. Figure (7.2.a) presents an example function with a few local variables, their memory layout (7.2.b), and colours (7.2.c).

7.4.1 What is Permissible? What is not?

BodyArmour permits memory accesses such as the two tick-marked accesses in Figure (7.2.b). In the first case, the program perhaps iterates over the elements in the

![Figure 7.2: BA-objects mode: variable colours.](image-url)
array (at offsets 4, 12, 20, and 28 in the object), and generates a pointer dereference to the second element (offset 12) by adding sizeof(pair_t) to the array’s base pointer at offset 4. In the second case, it accesses the privileged field of mystruct through a pointer to the last element of the array (offset 32). Even though the program accesses a field beyond the array’s boundaries, it remains within a local variable on the stack (mystruct), and we allow for this operation. A custom implementation of a memory initialisation routine could iterate over the elements of the whole structure, and generate such an access pattern. Refer to Figure (7.2.a).

However, BodyArmour stops the program from accessing the len and i fields through a pointer into the structure. len, i and mystruct are separate local variables on the stack, and one cannot be accessed through a pointer to the other.

7.4.2 Colours

BodyArmour uses a colour scheme to enforce protection. First, we sketch how it assigns the colours. In the next section, we explain how the colours are used.

BodyArmour assigns colours to each word of a variable when the program allocates memory for it in global, heap, or stack memory. A unique colour corresponds to the complete object. Refer to Figure (7.2.c). Thus, just like more existing solutions (e.g., WIT [6] and BBC [7]), in the BA-objects mode, BodyArmour operates at the granularity of full objects.

7.4.3 Protection by Colour Matching

When a buffer of colour X is assigned to a pointer, BodyArmour associates the same colour with the register containing the pointer. The colour does not change when the pointer value is manipulated (e.g., when the program adds an offset to the pointer), but it is copied when the pointer is copied to a new register. When the program stores a pointer to memory, we also store its colour to a memory map, to load it later when the pointer is restored to a register.

From now on, BodyArmour vets each dereference of the pointer to see if it is still in bounds. Vetting pointer dereferences is a matter of checking whether the colour of the pointer matches that of the memory it points to.

7.4.4 Pointer Subtraction: What if the Code is Colour Blind?

The colours assigned by BodyArmour prevent a binary from accessing object X though a pointer to object Y. Even though programs in C are not expected to do so, some functions exhibit “colour blindness”, and directly use a pointer to one object to access another object. The strcat() function in current libc implementations on Linux is the best known example: to append a source to a destination string, strcat() accesses both by the same pointer—adding the distance between them to access the remote string. While the details are tedious, the solution is conceptually
straightforward: (1) for code sequences that exploit pointer distances, detect the use of the distance to reach across objects, and (2) protect the dereferences in the usual way.

### 7.4.5 Stale Colours and Extra Measures to Rule out False Positives

As we discussed in Section 7.3, due to lack of coverage, we must anticipate that a program path executed at runtime has some pointer copy, pointer initialisation, or pointer dereference instructions not instrumented even though it should have been. This might lead to the use of a stale colour.

Consider a function like `memcpy(dst,src)`. Suppose that BodyArmour misses the `dst` buffer during analysis, so that it (erroneously) does not instrument the instructions manipulating the `dst` pointer prior to calling `memcpy()`—say, the instruction that pushes `dst` on the stack. Also suppose that `memcpy()` itself is instrumented, so the load of the `dst` pointer into a register obtains the colour of that pointer. However, since the original push was not instrumented, BodyArmour never set that colour! If we are lucky, we simply find no colour, and everything works fine. If we are unlucky, we pick up a stale colour of whatever was previously on the stack at that position\(^1\). As soon as `memcpy()` dereferences the pointer, the colour check fails and the program crashes.

BodyArmour removes all false positives of this nature by adding an additional tag to the colours to indicate to which memory address the colour corresponds. The tag functions not unlike a tag in a hardware cache entry: to check whether the value we find really corresponds to the address we look for. For instance, if `eax` points to `dst`, the tag contains the address `dst`. If the program copies `eax` to `ebx`, it also copies the colour and the tag. When the program manipulates the register (e.g., `eax++`), the tag incurs the same manipulation (e.g., `tag_{eax++}`). Finally, when the program dereferences the pointer, we check whether the colour in the tag corresponds to the memory it points to. Specifically, BodyArmour checks the colours on a dereference of `eax`, \(\text{iff}\ (tag_{eax} == eax)\). Thus, it ignores stale colours and prevents the false positives.

### 7.4.6 Why False Positives are not Possible

To justify that BodyArmour can effectively rule out false positives, we have to answer two questions: (1) why we do not expect problems in the dynamic analysis phase, when Howard runs a binary, and dumps instructions to be instrumented, and (2) why a program path executed at runtime, which has not been seen before during the analysis, does not exhibit any false alerts.

In this section, we do not address the problem of possible false positives caused by underestimated buffer sizes. It can not occur if we have symbol tables. Even

---
\(^1\)The reason there may be stale colours for the stack value, is that it is probably not possible and certainly not practical to clean up all colours whenever memory is no longer in use.
without them, Section 7.2.1 showed that underestimation is rare in practice.

**Soundness of the BodyArmour policy**

To ascertain that the protection policy implemented by BodyArmour is sound, observe that BodyArmour aims to prohibit only an access to object X through a pointer to object Y. Such behaviour should not occur in practice, and the real world programs we tested do not exploit the relative distance between objects to access memory. Other projects, e.g., WIT [6] or Avots et al. [15], explicitly assume that a pointer to an object can only be derived from a pointer to the same object.

The only exceptions we came across are the `strcat()` and `__strcpy_chk()` functions in current libc implementations on Linux. They access both source and destination buffers by the same pointer — adding the distance between them to calculate an address in the remote string. As explained in Section 7.4.4, our current solution to deal with this problem is very straightforward. Basically, Howard detects a pointer subtraction operation, and later spots when the resultant distance is added to the subtrahend to access the buffer associated with the minuend pointer. At this point, we reset the colour to reflect the remote buffer, and we protect dereferences in the usual way. False positives are ruled out.

If more elaborate implementations of this phenomenon appear, we can always prevent the associated dereferences from being checked at all. To reach the remote buffer, such scenarios usually have an operation which involves adding a value derived from the distance between two pointers. If Howard simply does not include it in the set of instructions to be instrumented, the tag of the resultant pointer will not match its value (refer to Section 7.4.5), and the colour check will not be performed. More complicated cases are unlikely.

**Expect the Unexpected Paths**

The other possible reason for false positives stems from limited code coverage. As we discussed in Section 7.3, we must anticipate a program path executed at runtime, \( p_R \), which differs from all paths seen before by Howard, but shares parts with (some of) them. It can happen that \( p_R \) accesses an array, but some of the instructions associated with these accesses are not instrumented by BodyArmour. We need to ascertain that such path does not cause any unexpected problems.

Say that the program executing \( p_R \) accesses an array. If \( arr \) is a pointer to this array, three generic types of instructions might be missed, and not instrumented by BodyArmour: (1) an \( arr \) initialisation instruction, (2) an \( arr \) update/manipulation instruction, and (3) an \( arr \) dereference instruction.

The crucial feature of BodyArmour which prevents false positives in cases (1) and (2) are the tags introduced in Section 7.4.5. They check whether the colour associated with a pointer corresponds to the value we look for. In the case of a pointer initialisation or a pointer update instruction missing, the pointer tag does not match
its value anymore, its colour is considered invalid, and it is not checked on dereferences. Finally, if an arr dereference instruction is not instrumented, it only means that the colour check is not performed. Again, it can only result in false negatives, but never false positives.

7.5 **BA-fields mode: a Colourful Armour**

In general terms, the main difference between the BA-objects mode discussed in Section 7.4 and the BA-fields mode is the granularity of protection. When operating in the BA-objects mode, BodyArmour protects memory at the level of objects, i.e., local variables allocated on the stack, global variables or objects allocated on the heap. In this section, we describe a version of BodyArmour which offers a more fine-grained protection—at the level of fields in data structures. As we will see later, with the extra protection, come also more chances for false positives, which need to be curbed.

7.5.1 **What is Permissible? What is not?**

To make the things more concrete consider the structure in Figure (7.3.a) with a memory layout as shown in Figure (7.3.c). BodyArmour permits legitimate memory accesses such as the two tick-marked accesses in Figure (7.3.c). In the first case, exactly as in Figure (7.2.b), the program perhaps iterates over the elements in the array (at offsets 4, 12, 20, and 28 in the object), and generates a pointer dereference to the second element (offset 12) by adding sizeof(pair_t) to the array’s base pointer at offset 4. In the second case, it accesses the y field at offset 8 via the struct’s base pointer at offset 4. All normal behaviour.

However, unlike in Section 7.4, BodyArmour in the BA-fields mode stops the program from accessing the privileged field via a pointer into the array. Similarly, it prevents accessing the x field of array element i from the y field in element \((i - 1)\). These accesses do not normally occur in programs (and if they did occur in the program, the data structure extractor would have classified the data structures differently)\(^2\).

Even these simple examples require very precise protection. For example, a naive, and broken, solution would be to colour at the granularity of top-level field—say the entire array buf (including subfields) green, the age field blue, and the privileged field red. Doing so allows us to prevent all accesses to red or blue memory by means of green pointers, but this is not sufficient. Even within the array, we want to prevent certain accesses, as discussed previously.

\(^2\)In the example code, we cannot actually exploit the access of \(x\) by means of a pointer to the \(y\) field of the previous element, but we would have been able to, if the \(y\) field had been an array.
typedef struct pair {
    int x; int y;
} pair_t;

struct s {
    int age;
    pair_t buf[4];
    int privileged;
} mystruct;

/* initialize the buffer */
int *p;
int len = 4; //buf length
for(p = mystruct.buf;
p < mystruct.buf+len;
p++)
    *p = 0;

Figure 7.3: BA-fields mode: shades and nesting levels.

7.5.2 Shaded Colours

BodyArmour uses a shaded colour scheme to enforce fine-grained protection. Compared to the BA-objects mode, the colour scheme used here is much more rich. In Section 7.4, the whole object was given a single colour, while in the BA-fields mode we add shades to colours to distinguish between fields in a structure. First, we sketch how we assign the colours. In the next section, we explain how the colours are used.

BodyArmour assigns colours to each word of an object when the program allocates memory for it in global, heap, or stack memory. Since it knows the structure of the object, it can assign separate colours to each variable and to each field. Colours in BodyArmour are hierarchical, much like real colours are hierarchical. Army green, lime green, neon green, and emerald are all shades of green, and dark lime green and light lime green, are gradations of lime green. And so on. Thus, we identify a byte’s colour as a sequence of shades:

\[ C_0 : C_1 : \ldots : C_N \]

where we interpret \( C_{i+1} \) as a shade of colour \( C_i \). Each shade corresponds to a nesting level in the data structure. This is illustrated in Figure (7.3.d).

The base colour, \( C_0 \), corresponds to the complete object, and is very similar to colours used by BodyArmour in the BA-objects mode. For simplicity, we did not show it as an actual colour in the figure; the base colour is the same for all fields in the allocated object. In other words, it distinguishes between individually allocated objects. At level 1, the object in Figure (7.3.d) has three fields, each of which gets a unique shade \( C_1 \). The two integer fields do not have any further nesting, but the array field has two more levels: array elements and fields within the array elements. Again, we assign a unique shade to each array element and, within each array element, to each field. The only exceptions are the base of the array and the base of the structs—they remain blank for reasons we explain shortly. Finally, each
7.5. BA-FIELDS MODE: A COLOURFUL ARMOUR

colour $C_i$ ($0 < i \leq N$) has a type flag indicating whether it is an array element. In the figure, the array elements are indicated with a yellow dot.

We continue the colouring process, until we reach the maximum nesting level (in the figure, this happens at $C_3$), or exhaust the maximum colour depth $N$. In the latter case, the object has more levels of nesting than BodyArmour can accommodate in shades, so that some of the levels will collapse into one, ‘flattening’ the substructure. Collapsed structures reduce BodyArmour’s granularity, but do not cause problems otherwise. In fact, most existing solutions (like WIT [6] and BBC [7]) operate only at the granularity of the full object.

7.5.3 Protection by Colour Matching

There are two main differences between the colour schemes implemented in the BA-objects mode and the BA-fields mode which we must take into account. First, the colours are more complex now, including multiple shades — we need a new procedure to compare them, and decide what is legal, and what is not. Next, it is not enough anymore to set the colour of a pointer just once, on a pointer initialisation instruction. For example, in order to protect the inner array in `mystruct` in Figure 7.3, the colour of a pointer needs to be updated to reflect the array, and not the whole structure. In this section, we describe the steps taken by BodyArmour.

The description of the procedure starts in exactly the same way as in the BA-objects mode. When a pointer is assigned to a buffer of colour X, BodyArmour associates the same colour with the register containing the pointer. The colour does not change when the pointer value is manipulated (e.g., when the program adds an offset to the pointer), but it is copied when the pointer is copied to a new register. When the program stores a pointer to memory, we also store its colour to a memory map, to load it later when the pointer is restored to a register.

What is different from the BA-objects mode, is the colour update rule: when the program dereferences a register, we update its colour so that it now corresponds to the memory location associated with the register again. The intuition is that we do not update colours on intermediate pointer arithmetic operations, but that the colours represent pointers used by the program to access memory.

From now on, BodyArmour vets each dereference of the pointer to see if it is still in bounds. Vetting pointer dereferences is a matter of checking whether the colour of the pointer matches that of the memory it points to—in all the shades, from left to right. Blank shades serve as wild cards and match any colour. Let us illustrate this by returning to our previous examples.

Suppose the program has already accessed the first array element by means of a pointer to the base of the array at offset 4 in the object. In that case, the pointer’s initial colour is set to $C_1$ of the array’s base. Next, the program adds `sizeof(pair_t)` to the array’s base pointer and dereferences the result to access the second array element. At that point, BodyArmour checks whether the colours match. $C_0$ clearly matches, and since the pointer has only the $C_1$ colour of the first array element, its
colour and that of the second array element match. Our second example, accessing the y field from the base of the array, matches for the same reason.

However, an attacker cannot use this base pointer to access the privileged field, because the \( C_1 \) colours do not match. Similarly, going from the y field in the second array element to the x field in the third element will fail, because the \( C_2 \) shades differ.

### 7.5.4 What if the Code is Colour Blind?

If a program always accessed its data structures in a way that reflects the structure, all would be well. However, we have already seen that this is not the case. Programs frequently use functions similar to \texttt{memset} to initialise (or copy) an entire object, with all subfields and arrays in it. Unfortunately, these functions do not heed the structure at all. Rather, they simply trample over the entire data structure in, say, word-size strides. Here is an example. Suppose \( p \) is a pointer to an integer and we have a custom \texttt{memset}-like function:

```c
for (p=objptr, p<sizeof(*objptr); p++) *p = 0;
```

The code is clearly ‘colour blind’, but while it violates the colour protection, BodyArmour should not raise an alert as the accesses are all legitimate. It should not ignore colour blindness either. For instance, the initialisation of one object should not trample over other objects. Or inside the structure of Figure (7.3.b): an initialisation of the array should not overwrite the privileged field.

One (bad) way to handle such colour blindness is to white-list the code. For instance, we could ignore all accesses from white listed functions. While this helps against some false alerts, it is not a good solution for two reasons. First, it does not scale; it helps only against a few well-known functions (e.g., \texttt{libc} functions), but not against applications that use custom functions to achieve the same. Second, as it ignores these functions altogether, it would miss attacks that use this code. For instance, the initialisation of (just) the buffer could overflow into the privileged field.

**Masks** Instead, BodyArmour exploits the shaded colours of Section 7.5.2 to implement masks. Masks shield code that is colour blind from some of the structure’s subtler shades. For instance, when the initialisation code is applied to the array in Figure (7.3.b), we filter out all shades beyond \( C_1 \): the code is then free to write over all the records in the array, but cannot write beyond the array. Similarly, if an initialisation routine writes over the entire object, we filter all shades except \( C_0 \), limiting all writes to this object.

Figure (7.3.e) illustrates the usage of masks. The code on the left initialises the array in the structure of Figure 7.3. By masking all colours beyond \( C_0 \) and \( C_1 \) all normal initialisation code is permitted. Supposing attackers can somehow manipulate the \texttt{len} variable, they could try to overflow the buffer and change the privileged
field. However, in that case the $C_1$ colours do not match, and BodyArmour will abort the program.

To determine whether a memory access needs masks (and if so, what sort), BodyArmour’s dynamic analysis first marks all instructions that trample over multiple data structures as ‘colour blind’ and determines the appropriate mask. For instance, if an instruction accesses the base of the object and covers it entirely, BodyArmour sets the masks to block out all colours except $C_0$. If an instruction accesses a field at the $k^{th}$ level in the structure, BodyArmour sets the masks to block out all colours except $C_0$...$C_k$. And so on.

Finding the right masks to apply and the right places to do so, requires fairly subtle analysis:

- BodyArmour needs to decide at runtime which part of the shaded colour to mask. In the above example, if the program initialises the whole structure, BodyArmour sets the masks to block out all colours except $C_0$. If the same function is called to initialise the array, however, only $C_2$ and $C_3$ are shielded. To do so, BodyArmour’s dynamic analysis tracks the source of the pointer used in the ‘blind’ instruction, i.e., the base of the structure or array. The instrumentation then allows for accesses to all fields included in the structure (or substructure) rooted at this source.

- However, not all such instructions need masks. For instance, code that zeroes all words in the object by adding increasing offsets to the base of the object, has no need for masks. After all, because of its blank shades the base of the object permits access to the entire object even without masks.

BodyArmour enforces the masks when rewriting the binary. Rather than checking all shades, it checks only the instructions’ visible colours for these instructions.

**Colour blindness and pointer subtraction**  The above explanation easily covers the vast majority of cases, but there are some rare code sequences that need special treatment. Specifically, some functions exhibit colour blindness of a different nature, where a pointer to one object is used directly to access another object. The situation does not differ from the one discussed in Section 7.4.4, and we apply exactly the same solution.

### 7.5.5 Why We do Not See False Positives in Practice

The only potential cause for false positives is lack of coverage. For instance, if we do not have symbol tables, we may underestimate the size of arrays. Or we may execute the instrumented code with very different data than during the analysis, with false positives caused by incorrect masking. Similarly as we argued in Section 7.4, array size underestimation cannot occur if we have the symbol tables. Even without them, Section 7.2.1 showed that underestimation is rare in practice. But other coverage issues occur regardless of symbol tables. They must be curbed.
Stale Colours and Additional Tags

In Section 7.4.5, we discussed the issue of lack of coverage leading to the use of stale colours. The problem certainly remains when BodyArmour is run in the BA-fields mode, and the solution applied is exactly the same. We implement tags, which check whether the colour associated with a pointer corresponds to the value we look for. If not, the colour is considered invalid, and is not checked on dereferences.

Missed Masks and Context Checks

Limited code coverage may also cause BodyArmour to miss the need for masks and, unless prevented, lead to false positives. Consider again the example custom memset function in Section 7.5.4. The code is colour blind, unaware of the underlying data structure, and accesses the memory according to its own pattern. To prevent false positives, we introduced masks that filter out some shades to allow for benign memory accesses.

Suppose that during analysis the custom memset function is invoked only once, to initialise an array of 4-byte fields. No masks are necessary. Later, in a production run, the program takes a previously unknown code path, and uses the function to access an array of 16-byte structures. Since it did not assign masks to this function originally, BodyArmour raises a (false) alarm.

To prevent the false alarm, we keep two versions of each function in a program: a vanilla one, and an instrumented one which performs colour checking. When the program calls a function, BodyArmour checks whether the function call also occurred at the same point during the analysis by examining the call stack. Depending on the outcome, it decides whether or not to execute the instrumented version of the function. It performs colour checking only for layouts of data structures we have seen before, so we do not encounter code that accesses memory in an unexpected way.

7.5.6 Are False Positives Still Possible?

As lack of false positives is a desired feature of each protection mechanism, we have applied powerful measures to decrease the odds against false alarms. Even though they prove enough in practice — we have not encountered any false positives during our experiments, we cannot guarantee that they are ruled out. In this section, we discuss theoretically possible scenarios that can still lead to false positives.

Possible false positives stem from lack of code coverage. The principal reason is a failure to notice the need for masks. As we explained in Section 7.5.5, we try to limit colour checks to program paths and data structures which were seen before, during the dynamic analysis carried out by Howard. On each function call we perform a context check, and we execute an instrumented version of a function only if the current call stack was seen before. Otherwise, a vanilla version is run, which does not check colours on pointer dereferences.
While the solution we have implemented is effective in practice, it does not give any strong guarantees. The problem is that a call stack does not identify the context of a sequence of instructions with an absolute precision. Figure 7.4 shows a possible problematic scenario. In this case, it should be not the call stack, but a node in the program control flow graph which identifies the context. Only if we saw the loop in lines [6-9] initialising the infamous array_of_structs, should we allow for an instrumented version of it at runtime. However, a solution tracking the control flow graph of a program would be not only leading to numerous false negatives, but it would be also prohibitively expensive.

Observe that the scenario presented in Figure 7.4 is fairly imaginary. First, the offensive function must exhibit the need for masks, that is, it must access subsequent memory locations through a pointer to a previous field. Second, it needs to be called twice with particular sets of arguments leading to the awkward situation.

Because we did not encounter any problematic scenarios during all our experiments, and the BA-fields mode offers an attractive fine-grained protection, we think that the risk is acceptable provided an application can afford rare crashes.

7.6 Efficient Implementation

Protection by colour matching combined with masks for colour blindness allows BodyArmour to protect data structures at a finer granularity than previous approaches. Even so, the mechanisms are sufficiently simple to allow for efficient implementations. BodyArmour is designed to instrument 32-bit ELF binaries for the Linux/x86 platforms. Like Pebil [121], it performs static instrumentation, i.e., it inserts additional code and data into an executable, and generates a new binary with permanent modifications. We first describe how BodyArmour modifies the layout of an ELF binary, and next present some details about the instrumentation.
7.6.1 Updated Layout of ELF Binary

To accommodate new code and data required by the instrumentation, BodyArmour modifies the layout of an ELF binary. First, the original data segment stays unaltered, while we modify the text segment only in a minor way—just to allow for the selection of the appropriate version of a function (Section 7.5.5), and to assure that the resulting code works correctly—mainly by adjusting jump targets to reflect addresses in the updated code (we discuss this further in Section 7.6.2). Next, to provide protection, BodyArmour inserts a few additional segments in the binary: BA_Initilised_Data, BA_Unitialised_Data, BA_Procedures, and BA_Code.

Both data segments—BA_(Un)Initialised_Data—store data structures that are internally used by BodyArmour, e.g., data structure colour maps, or arrays mapping addresses in the new version of a binary to the original ones. The BA_Procedures code segment contains various chunks of machine code used by instrumentation snippets (e.g., a procedure that compares the colour of a pointer with the colour of a memory location). Finally, the BA_Code segment is the pivot of the BodyArmour protection mechanism—it contains the original program’s functions instrumented to perform colour checking.

7.6.2 Instrumentation Code

To harden the binary, we rewrite it to add instrumentation to those instructions that dereference the array pointers. We use multi-shade colours only if the data structures are nested. Often, we can tell that a variable is a string, or some other non-nested array. In those cases, we switch to a simpler, single-level colour check.

To provide protection, BodyArmour reorganises code at the instruction level. We do not need to know function boundaries, as we instrument instructions which were classified as array accesses during the dynamic analysis phase. The whole procedure also requires instrumenting, for example, pointer move or pointer initialisation instructions. All are classified dynamically. We briefly describe the main steps taken by BodyArmour during instrumentation: (1) inserting trampolines and method selector, (2) code relocation, (3) inserting instrumentation. A full explanation of the instrumentation procedure can be found in [194].

Inserting trampolines and method selector. The role of a method selector is to decide whether a vanilla or an instrumented function should be executed (see Section 7.5.5), and then jump to it. In BodyArmour, we place a trampoline at the beginning of each (dynamically detected) function in the original text segment, which jumps to the method selector. The selector picks the right code to continue execution, as discussed previously.

Code relocation. BodyArmour’s instrumentation framework must be able to add an arbitrary amount of extra code between any two instructions. In turn, targets of
all jump and call instructions in a binary need to be adjusted to reflect new values of the corresponding addresses. As far as direct/relative jumps are concerned, we simply calculate new target addresses, and modify the instructions. Our solution to indirect jumps is similar to implementations found in multiple dynamic instrumentation frameworks [31; 193]: indirect jumps are resolved at runtime, by employing arrays which maintain a mapping between old and new addresses.

**Inserting instrumentation.** Snippets come in many shapes. For instance, snippets to handle pointer dereferences, to handle pointer move instructions, or to colour memory returned by malloc. Some instructions require that a snippet performs more than one action. For example, an instruction which stores a pointer into an array, needs to both store the colour of the pointer in a colour map, and make sure that the store operation is legal, i.e., within the array boundaries. For an efficient instrumentation, we have developed a large number of carefully tailored snippets.

Colours map naturally on a sequence of small numbers. For instance, each byte in a 32-bit or 64-bit word may represent a shade, for a maximum nesting level of 4 or 8, respectively. Doing so naively, incurs a substantial overhead in memory space, but just like in paging, we need only allocate memory for colour tags when needed. The same memory optimisation is often used in dynamic taint analysis. In the IA32 implementation evaluated in Section 7.7, we use 32 bit colours with four shades and 16 bit tags.

Figure 7.5 shows an example of an array dereference in the hardened binary—after rewriting by BodyArmour. The code is a little simplified for brevity, but otherwise correct. We see that each array dereference incurs some extra instructions. If the colours do not match, the system crashes. Otherwise, the dereference executes as intended.

### 7.7 Evaluation

We evaluate BodyArmour on performance and on effectiveness in stopping attacks. To do so, we analysed and rewrote a variety of applications. As the analysis engine is based on the Qemu process emulation which is currently only available on Linux, all examples are Linux-based. However, the approach is not specific to any operating system.

We have performed our analysis for binaries compiled with two compiler versions, gcc-3.4 and gcc-4.4, and with different optimisation levels. All results presented in this section are for binaries compiled with gcc-4.4 -O2 and without symbols. All experiments are performed for BodyArmour operating in the BA-fields mode.

**Performance** To evaluate BodyArmour’s performance, we compare the speed of instrumented (armoured) binaries with that of unmodified implementations. Our
Say the original dereference looked as follows:

```c
mov $0x2210078, (tdata, tbase, 4)
```

It is an access in an array, so it should be instrumented. As a result, the access above is replaced with the following code:

```c
_dereference_check_start:
    # check whether the tag value corresponds to the pointer
cmp tdata, TagRegister_tag_word
je _dereference_check_end

... save eax and ebx registers used in instrumentation ...

les (tdata, tbase, 4); tbase ; address to dereference -> tbase
call get_color_of_tag ; color loaded in tbase -> tbase
call _color_match ; compare colors in tbase and the tag
        call color_match ; compare colors in tbase and the tag
je _dereference_ok

_dereference_check_end:
```

Figure 7.5: Instrumentation for an array pointer dereference (with 16 bit colours and tags). For the sake of clarity this code is highly inefficient (e.g., it uses two `call` instructions). In reality, BodyArmour uses snippets tailored to performance.

The test platform is a Linux 2.6 system with an Intel(R) Core(TM)2 Duo CPU clocked at 2.4GHz with 3072KB L2 cache. The system holds 4GB of memory. For our performance tests we used an Ubuntu 10.10 install. We ran each experiment multiple times and present the median. Across all experiments, the 90th percentiles were typically within 10% and never more than 20% off the mean.

We evaluate the performance of BodyArmour with a variety of applications—all of the well-known `nbench` integer benchmarks [1], and a range of real-world programs. We picked the `nbench` benchmark suite, because it is compute intensive and several of the individual tests represent close to worst-case scenarios for BodyArmour, as they revolve around string and array manipulation solely.

For the real-world applications, we chose a variety of very different programs: a network server (`lighttpd`), several network clients (`wget, htget`), and a more compute-intensive task (`gzip`). `Lighttpd` is a high-performance web server used by such popular sites as YouTube, SourceForge, Wikimedia, Meebo, and ThePirateBay. `Wget` and `htget` are well-known command-line web clients. `Gzip` implements the DEFLATE algorithm which includes many array and integer operations. We evaluate both `lighttpd` and `gzip` with different object sizes.

Figure 7.6 shows that for real client-side applications like `wget` and `htget` the slowdown for I/O intensive applications is negligible, while `gzip` incurs a slowdown of approximately 1.7x. As `gzip` is a very expensive test for BodyArmour, the slowdown was much less than we expected.

As illustrated in Figure 7.7, the overhead for a production-grade web server like `lighttpd` is also low: less than 2.8x for all object sizes, and as little as 1.16x for large objects. Clearly, in networking applications I/O dominates the performance.
7.8. RELATED WORK

and the overhead of BodyArmour is less important.

Figure 7.8 shows the results for the very compute-intensive nbench test suite. The overall slowdown for nbench is 2.9x. Since this benchmark suite was chosen as worst-case scenario and we have not yet fully optimised BodyArmour’s instrumentation, we were surprised the overhead was not higher. Some of the tests incurred a fairly minimal slowdown. Presumably, these benchmarks are dominated by operations other than array accesses. String sort and integer sort, on the other hand, manipulate strings and arrays constantly and thus incur much higher slowdowns.

Effectiveness Table 7.1 shows the effectiveness of BodyArmour in detecting attacks on a range of real-life software vulnerabilities. BodyArmour detected all attacks we tried and did not generate any false positives during any of our experiments. The attacks detected vary in nature and include overflows on both heap and stack, local and remote, and of both control and non-control data. The detection of attacks on non-control data is especially encouraging.

7.8 Related Work

The easiest way to prevent memory corruption attacks is to do so at the source level, using a safe language or compiler extension. Unfortunately, doing so leaves many binaries at the mercy of attackers. Protection at the binary level is much harder. In this chapter, we explained our solution for protecting binaries, but since our work was inspired by the WIT compiler extension, we briefly look at compile time solutions also.

Protection at compile time Managed languages like Java and C# are safe from buffer overflows by design. As we discussed already in Section 2.1, efforts like
Chapter 7. Body Armour for Binaries

Figure 7.7: Performance of Body Armour for lighttpd (in connections/s as measured by httpperf).

Cyclone [104] and CCured [57] show that similar protection also fits dialects of C—although the overhead is not always negligible. Better still, recent work on data flow integrity (DFI) [40], write integrity testing (WIT) [6], and baggy bounds checking (BBC) [7] prove that powerful protection against memory corruption is possible even for unmodified C.

Body Armour was inspired by the WIT compiler extension—an attractive defence framework that marries immediate (fail-stop) detection of memory corruption to excellent performance. For each instruction, WIT finds the approximate set of objects written by it and dynamically prevents writes to objects not in the set. To do so, WIT relies on the precision of points-to analysis applied at compile time on the source code.

Unfortunately, access to source code or recompilation is often not possible in practice. Most vendors do not share the source, or even the symbol tables with their customers. In all probability, many programs in use today will never be recompiled at all. To protect such software, we need a solution that works for binaries.

Also, WIT and BBC protect at the granularity of memory allocations. If a program allocates a structure that contains an array as well as other fields, overflows within the structure go unnoticed. As a result, the attack surface for memory attacks is still huge. SoftBound is one of the first tools to protect subfields in C structures [140]. Again, SoftBound requires access to source code, which is not too helpful for legacy software.

Body Armour’s protection resembles that of WIT, but without requiring source code, debugging information, or even original symbol tables. Unlike WIT, it protects at the granularity of subfields in C structures. Less importantly, it prevents not just out-of-bounds writes, as WIT does, but also reads³. As a drawback, Body Armour may be less accurate, since dynamic analysis may not cover the entire program.

³We do not consider this an important issue, as it can probably be incorporated in WIT also (with a bit more overhead).
7.9. DISCUSSION

Figure 7.8: Performance overhead for the compute-intensive nbench benchmark suite.

Protection of binaries  Arguably some of the most popular measures to protect against memory corruption are memory debuggers like Purify [99] and Valgrind [145]. These powerful testing tools are capable of finding many memory errors without source code. However, they incur overheads of an order of magnitude or more. Moreover, their accuracy depends largely on the presence of debug information and symbol tables. In contrast, BodyArmour is much faster and requires neither.

As we explained in detail in Section 3.2, an important class of approaches to detect the effects of memory corruption attacks is based on dynamic taint analysis. DTA does not detect the memory corruption itself, but may detect malicious control flow transfers. Unfortunately, the control flow transfer occurs at a (sometimes much) later stage. In Section 3.3, we discussed that the technique poses serious problems when one tries to extend it to cater to the non-control-diverting attacks — it is prone to both false positives and false negatives. With typical slowdowns of an order of magnitude, DTA in software is also simply too expensive for production systems.

7.9 Discussion

Obviously, BodyArmour is not flawless. In this section, we discuss some generic limitations.

With a dynamic approach, BodyArmour protects only arrays detected by Howard. If the attackers overflow other arrays, we will miss the attacks. Also, if particular array accesses are not exercised in the analysis phase at all, the corresponding instructions are not instrumented either. Combined with the tags introduced in Section 7.4.5, this lack of accuracy can only cause false negatives, but never false positives. In practice, as we have seen in Section 7.7, BodyArmour was able to protect all vulnerable programs we tried.

Howard itself is designed to err on the safe side. In case of doubt, it overestimates the size of an array. Again, this can lead to false negatives, but not false
<table>
<thead>
<tr>
<th>Application</th>
<th>Type of vulnerability</th>
<th>Security advisory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aeon 0.2a</td>
<td>Stack overflow</td>
<td>CVE-2005-1019</td>
</tr>
<tr>
<td>Aspell 0.50.5</td>
<td>Stack overflow</td>
<td>CVE-2004-0548</td>
</tr>
<tr>
<td>Htget 0.93 (1)</td>
<td>Stack overflow</td>
<td>CVE-2004-0852</td>
</tr>
<tr>
<td>Htget 0.93 (2)</td>
<td>Stack overflow</td>
<td></td>
</tr>
<tr>
<td>Iwconfig v.26</td>
<td>Stack overflow</td>
<td>CVE-2003-0947</td>
</tr>
<tr>
<td>Ncompress 4.2.4</td>
<td>Stack overflow</td>
<td>CVE-2001-1413</td>
</tr>
<tr>
<td>Proftpd 1.3.3a</td>
<td>Stack overflow</td>
<td>CVE-2010-4221</td>
</tr>
<tr>
<td>bc-1.06 (1)</td>
<td>Heap overflow</td>
<td>Bugbench [128]</td>
</tr>
<tr>
<td>bc-1.06 (2)</td>
<td>Heap overflow</td>
<td>Bugbench [128]</td>
</tr>
<tr>
<td>Exim 4.41</td>
<td>Heap overflow, non-control data</td>
<td>CVE-2010-4344</td>
</tr>
<tr>
<td>Nullhttpd-0.5.1</td>
<td>Heap overflow, reproduced</td>
<td>CVE-2002-1496</td>
</tr>
<tr>
<td>Squid-2.3</td>
<td>Heap overflow, reproduced</td>
<td>Bugbench [128]</td>
</tr>
</tbody>
</table>

Table 7.1: Tested vulnerabilities: all exploits were stopped by BodyArmour, including the attack on non-control data.

positives. However, if the code is strongly obfuscated or deliberately designed to confuse Howard, we do not guarantee that it will never misclassify a data structure in such a way that it will not cause a false positive. Still, it is unlikely, because to do so, the behaviour during analysis should also be significantly different from that during the production run. In our view, the risk is acceptable given our application domain: software deployments that can tolerate crashes, as long as they are rare.

We have implemented two versions of BodyArmour: the BA-objects mode, and the BA-fields mode. While the latter protects memory at a fine-grained granularity, there exist theoretical situations that can lead to false alerts. However, in practice we did not encounter any problems. Since the protection offered is very attractive — BodyArmour protects individual fields within structures — we again think that the risk is acceptable provided that an application can afford rare crashes.

Code coverage is a limitation of all dynamic analysis techniques and we do not claim any contribution to this field. Interestingly, code coverage can also be ‘too good’. For instance, if we were to trigger a buffer overflow during the analysis run, BodyArmour would interpret it as normal code behaviour and not prevent similar overruns during production. Since coverage techniques to handle complex applications are currently still fledgling, this is mostly an academic problem. At any rate, if binary code coverage techniques are so good as to find real problems in the testing phase, this can only be beneficial for the quality of software.
7.10 Conclusions

We described a novel approach to harden binary software without access to source code or even symbol tables. Using our approach, we can protect binaries against buffer overflows pro-actively, before we know they are vulnerable. Besides attacks that divert the control flow of a program, we also detect and stop attacks against non-control data. Even without much optimisation, the overhead on benchmarks and real applications is fairly low. Further, we demonstrated that it stops a variety of real exploits. Finally, as long as we are conservative in classifying data structures in the binaries, our method will not have false positives. We think the approach is a good starting point for further research in binary protection. Specifically, we plan to explore new ways to increase data structure coverage as well as code coverage to increase our accuracy. It is our view that protection at the binary level is important for dealing with real threats to real and deployed information systems. This chapter describes a promising new approach to provide such protection.
Chapter 8

Conclusions

The main goal of this thesis has been to investigate the ways of protecting legacy C binaries against memory corruption attacks. The research community has long recognised the problem, and has proposed multiple solutions. However, the existing proposals are either inapplicable to legacy software, or incur a high performance overhead.

Our focus has been on techniques that can be applied to existing binaries, such as dynamic information flow tracking. At the same time, we attempted to design solutions that would benefit from the wealth of information available at runtime, but that would still be applicable at an acceptable cost. Another primary concern of ours has been to provide mechanisms that would also cater to non-control-diverting attacks. Since these attacks do not lead to unusual code execution, but “silently” modify important data, they are exceedingly hard to detect. In fact, no previous solutions protect binaries against them.

This chapter summarises the results, and offers some future directions.

Results

We can summarise the results of the thesis in the following points:

1. We analyse and evaluate pointer tainting, an incarnation of dynamic taint analysis used to detect keyloggers and memory corruption attacks on non-control data (Chapter 3)
   - We carry out an in-depth analysis of the problems of pointer tainting on real systems, which shows that the method does not work against malware spying on users’ behaviour, and is problematic in other forms also. It is an open challenge to use an incarnation of pointer tainting to detect non-control-diverting attacks on the most popular PC architecture (x86) and the most popular OS (Windows).
• We present an analysis and evaluation of all known fixes to the problems. We argue that they all have serious shortcomings.

2. Prospector (Chapter 4)

• Prospector is a secure emulator that is able to identify all bytes contributing to a buffer overflow attack. The identification is performed without the need for replaying attacks, and is sufficiently fast to be used in honeypots. The information gathered can be used for signature generation, as well as assist human security experts in an attack analysis.

• We improve the accuracy of the existing vulnerability-based signatures based on the length of a protocol field. We show that by taking into account too little or too few protocol fields, both false positives and false negatives are possible. Prospector remedies this weakness.

3. Hassle (Chapter 5)

• Hassle is a honeypot that is capable of generating signatures for attacks over both encrypted and non-encrypted channels. Since the techniques we describe work by interposing encryption routines, they are applicable to most types of encryption and require no modification of the applications that need protection.

4. Howard (Chapter 6)

• Howard is a framework which extracts data structures from a stripped binary. The analysis is performed by dynamically observing the program’s memory access patterns.

• The main goal of Howard is to provide data structures that allow BodyArmour (Chapter 7) to retrofit security onto existing binaries. We demonstrate two other applications of data gathered by Howard: (1) we use Howard to furnish existing disassemblers and debuggers with information about data structures and types to ease reverse engineering, (2) we show how Howard can be used to further reverse engineer a binary, and recover high-level pointer structures, e.g., singly- and doubly-linked lists or trees.

5. BodyArmour (Chapter 7)

• We implemented BodyArmour, an approach to harden a legacy binary without access to source code or even the original symbol tables. Using our approach, we can protect binaries against buffer overflows proactively, before we know they are vulnerable. Besides attacks that divert the control flow of a program, we also detect and stop attacks against non-control data.
• BodyArmour can operate in two modes: the BA-fields mode and the BA-objects mode. In the BA-fields mode, by protecting individual fields inside a structure rather than aggregates, BodyArmour is significantly finer-grained than other solutions. Even though we cannot guarantee no false positives in the BA-fields mode, they are very unlikely, and we never encountered them. No false positives are possible when the protection is limited to structures (the BA-objects mode).

• Our solution stops a variety of exploits, and works in a timely fashion, e.g., the slowdown for gzip is only 1.7x.

Limitations and Future Work

In Chapter 4 we described Prospector, an emulator capable of tracking which bytes contribute to a buffer overflow attack on heap or stack. Whenever we recognise the protocol governing the malicious network message, we also generate a signature capable of identifying polymorphic versions of the attack. A signature consists of so called critical and value fields. While critical fields are determined automatically, Prospector relies on a protocol module that identifies value fields - fields which may be considered important, and should therefore be additionally added to the signature. Currently, we have built the protocol module manually. It would be interesting to investigate whether value fields could be determined automatically. A possible solution could employ path slicing, such as Bouncer [61].

The framework to extract data structures from a stripped binary we have described in Chapter 6 operates by dynamically observing memory access patterns. However, a variety of tools exists to perform or assist with C code obfuscation. They certainly aim to reduce the ability to understand or reverse engineer a program. For that reason, it is important to examine to what extent they affect the algorithm implemented by Howard. Additionally, one could investigate whether it is possible to (1) detect that a given binary obfuscation technique has been applied, and (2) deobfuscate it in order to understand the intended meaning of the code.

In Chapter 7 we introduced BodyArmour, a tool chain to bolt a layer of protection on existing C binaries to shield them from memory corruption attacks. Since it is based on dynamic analysis, it also suffers from coverage issues - we can only analyse what we execute. Lack of coverage may cause BodyArmour to miss arrays and array accesses and thus lead to false negatives. Another popular, yet very different, approach to analyse a binary is static analysis. Even though the method is less accurate than dynamic analysis, it offers a full code coverage. Consequently, it might be interesting to explore a hybrid solution, which would marry BodyArmour to static protection approaches, such as WIT [6].
References


REFERENCES


REFERENCES


[37] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and communica-


[48] Seongje Cho, Hyeyoung Chang, and Yookun Cho. Implementation of an ob-


REFERENCES


REFERENCES

on Applications of Cryptography, 1996.


[120] Benjamin A. Kuperman and Eugene Spafford. Generation of Application
REFERENCES

Level Data via Library Interposition. Technical Report CERIAS TR 1999-11, COAST Laboratory, Purdue University, 1999.


REFERENCES

windows/cse/bit_projects.mspx.


REFERENCES


ware & Vulnerability Assessment, DIMVA’06, 2006.


[199] David Thomas and Andrew Hunt. Programming Ruby: the pragmatic pro-


