Body armor for binaries: preventing buffer overflows without recompilation

Asia Slowinska
Vrije Universiteit Amsterdam

Traian Stancescu
Google, Inc.

Herbert Bos
Vrije Universiteit Amsterdam

Abstract

BinArmor is a novel technique to protect existing C binaries from memory corruption attacks on both control data and non-control data. Without access to source code, non-control data attacks cannot be detected with current techniques. Our approach hardens binaries against both kinds of overflow, without requiring the programs' source or symbol tables. We show that *BinArmor* is able to stop real attacks—including the recent non-control data attack on Exim. Moreover, we did not incur a single false positive in practice. On the downside, the current overhead of *BinArmor* is high—although no worse than competing technologies like taint analysis that do not catch attacks on non-control data. Specifically, we measured an overhead of 70% for `gzip`, 16%-180% for `lighttpd`, and 190% for the `nbench` suite.

1 Introduction

Despite modern security mechanisms like stack protection [16], ASLR [7], and PaX/DEP/W⊕X [33], buffer overflows rank third in the CWE SANS top 25 most dangerous software errors [17]. The reason is that attackers adapt their techniques to circumvent our defenses.

Non-control data attacks, such as the well-known attacks on *exim* mail servers (Section 2), are perhaps most worrying [12, 30]. Attacks on non-control data are hard to stop, because they do not divert the control flow, do not execute code injected by the attacker, and often exhibit program behaviors (e.g., in terms of system call patterns) that may well be legitimate. Worse, for binaries, we do not have the means to detect them *at all*.

Current defenses against non-control data attacks all require access to the source code [20, 3, 4]. In contrast, security measures at the binary level can stop various control-flow diversions [15, 2, 19], but offer no protection against corruption of non-control data.

Even for more traditional control-flow diverting attacks, current binary instrumentation systems detect only the *manifestations* 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 itself, which may have occurred thousands of cycles before. The lag between time-of-attack and time-of-manifestation makes it harder to analyze the attack and find the root cause [27].

In this paper, we describe *BinArmor*, a tool to bolt a layer of protection on C binaries that stops state-of-the-art buffer overflows immediately (as soon as they occur).

High level overview Rather than patching systems after a vulnerability is found, *BinArmor* is proactive and stops buffer (array) overflows in binary software, before we even know it is vulnerable. Whenever it detects an attack, it will raise an alarm and abort the execution. Thus, like most protection schemes, we assume that the system can tolerate rare crashes. Finally, *BinArmor* operates in one of two modes. In *BA-fields mode*, we protect individual fields inside structures. In *BA-objects mode*, we protect at the coarser granularity of full objects.

BinArmor relies on limited information about the program's data structures—specifically the buffers that it should protect from overflowing. If the program's symbol tables are available, *BinArmor* is able to protect the binary against buffer overflows with great precision. Moreover, in BA-objects mode no false positives are possible in this case. While we cannot guarantee this in BA-fields mode, we did not encounter any false positives in practice, and as we will discuss later, they are unlikely.

However, while researchers in security projects frequently assume the availability of symbol tables [19], in practice, software vendors often strip their code of all debug symbols. In that case, we show that we can use automated reverse engineering techniques to extract symbols from stripped binaries, and that this is enough to protect real-world applications against real world-attacks. To our knowledge, we are the first to use data structure

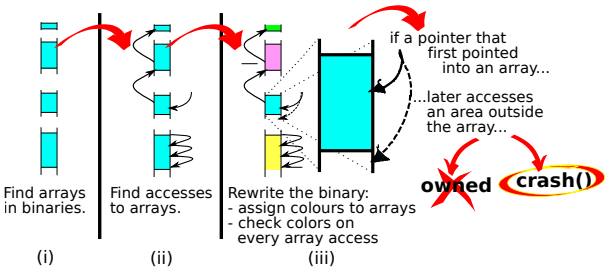

Fig. 1: *BinArmor* overview.

recovery to prevent memory corruption. We believe the approach is promising and may also benefit other systems, like XFI [19] and memory debuggers [24].

BinArmor hardens C binaries in three steps (Fig. 1):

(i) *Data structure discovery:* dynamically extract the data structures (buffers) that need protection.

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

(iii) *Rewrite:* statically rewrite the binary to ensure that a pointer accessing a buffer stays within its bounds.

Data structure discovery is easy when symbol tables are available, but very hard when they are not. In the absence of symbol tables, *BinArmor* uses recent research results [29] to reverse engineer the data structures (and especially the buffers) from the binary itself by analyzing memory access patterns (Fig. 1, step i). Something is a struct, if it is accessed like a struct, and an array, if it is accessed like an array. And so on. Next, given the symbols, *BinArmor* dynamically detects buffer 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 color. Every pointer used to access the buffer for the first time obtains the color of this buffer. *BinArmor* raises an alert whenever, say, a blue pointer accesses a red byte.

Contributions *BinArmor* proactively protects existing C binaries, before we even know whether the code is vulnerable, against attacks on control data *and* non-control data, and it can do so either at object or sub-field granularity. Compared to source-level protection like WIT, *BinArmor* has the advantage that it requires *no access to source code or the original symbol tables*. In addition, in BA-fields mode, by *protecting individual fields* inside a structure rather than aggregates, *BinArmor* is finer-grained than WIT and similar solutions. Also, it prevents overflows on both writes *and* reads, while WIT protects only writes and permits information leakage. Further, we show in Section 9 that points-to analysis (a technique relied on by WIT), is frequently imprecise.

Compared to techniques like taint analysis that also target *binaries*, *BinArmor* detects both control flow and *non-control flow attacks*, whereas taint analysis detects only the former. Also, it detects attacks *immediately* when they occur, rather than sometime later, when a function pointer is used.

The main drawback of *BinArmor* is the very significant slowdown (up to 2.8x for the lighttpd webserver and 1.7x for gzip). While better than most tainting systems (which typically incur 3x-20x), it is much slower than WIT (1.04x for gzip). Realistically, such slowdowns make *BinArmor* in its current form unsuitable for any system that requires high performance. On the other hand, it may be used in application domains where security rather than performance is of prime importance. In addition, because *BinArmor* detects buffer overflows themselves rather than their manifestations, we expect it to be immediately useful for security experts analyzing attacks. Finally, we will show later that we have not explored all opportunities for performance optimization.

Our work builds on dynamic analysis, and thus suffers from the limitations of all dynamic approaches: we can only protect what we execute during the analysis. This work is *not* about code coverage. We rely on existing tools and test suites to cover as much of the binary as possible. Since coverage is never perfect, we may miss buffer accesses and thus incur false negatives. Despite this, *BinArmor* detected all 12 real-world buffer overflow attacks in real-world applications we study (Section 8).

BinArmor takes a conservative approach to prevent false positives (unnecessary program crashes). For instance, no false positives are possible when the protection is limited to structures (BA-objects mode). In BA-fields mode, we can devise scenarios that lead to false positives due to the limited code coverage. However, we did not encounter any in practice, and we will show that they are very unlikely.

Since our dynamic analysis builds on Qemu [6] process emulation which is only available for Linux, we target x86 Linux binaries, generated by `gcc` (albeit of various versions and with different levels of optimization). However, there is nothing fundamental about this and the techniques should apply to other systems also.

2 Some buffer overflows are hard to stop: the Exim attack on non-control data

In December 2010, Sergey Kononenko posted a message on the *exim* developers mailing list about an attack on the *exim* mail server. The news was slashdotted shortly after. The remote root vulnerability in question concerns a heap overflow 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.

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. ASLR, W⊕X, canaries, system call analysis—all fail to stop or even detect the attack.

Both 'classic' buffer overflows [18], and attacks on non-control data [12] are now mainstream. While attackers still actively use the former (circumventing existing measures), there is simply *no* practical defense against the latter in binaries. Thus, researchers single out non-control data attacks as a serious future threat [30]. *BinArmor* protects against both types of overflows.

3 What to Protect: Buffer Accesses

BinArmor protects binaries by instrumenting buffer accesses to make sure they are safe from overflows. Throughout the paper, *a buffer* is an array that can potentially overflow. Fig. 1 illustrates the general idea, which is intuitively simple: once the program has assigned an array to a pointer, it should not use the same pointer to access elements beyond the array bounds. For this purpose, *BinArmor* assigns colors to arrays and pointers and verifies that the colors of memory and pointer match on each access. After statically rewriting the binary, the resulting code runs natively and incurs overhead only for the instructions that access arrays. In this section, we explain how we obtain buffers and accesses to them when symbols are not available, while Sections 5–7 discuss how we use this information to implement fine-grained protection against buffer overflows.

3.1 Extracting Buffers and Data Structures

Ideally, *BinArmor* obtains information about buffers from the symbol tables. Many projects assume the availability of symbol tables [19, 24]. Indeed, if the binary does come with symbols, *BinArmor* offers very accurate protection. However, as symbols are frequently stripped off in real software, it uses automated reverse engineering techniques to extract them from the binary. *BinArmor* uses a dynamic approach, as static approaches are weak at recovering arrays, but, in principle, they work also [26].

Specifically, we recover arrays using Howard [29], which follows the simple intuition that memory access patterns reveal much about the layout of data structures. In this paper, we sketch only the general idea and refer to the original Howard paper for details [29]. Using binary code coverage techniques [13, 9], Howard executes as many of the execution paths through the binary as possible and observes the memory accesses. To detect arrays, it first detects loops and then treats a memory area as an array if (1) the program accesses the area in a loop (either consecutively, or via arbitrary offsets from the array base), and (2) all accesses 'look like' array accesses (e.g., fixed-size elements). Moreover, it takes into account array accesses outside the loop (including 'first' and 'last' elements), and handles a variety of complications and optimizations (like loop unrolling).

Since arrays are detected dynamically, we should not underestimate the size of arrays, lest we incur false positives. If the array is classified as too small, we might detect an overflow when there is none. In Howard, the data structure extraction is deliberately conservative, so that in practice the size of arrays is either classified exactly right, or overestimated (which never leads to false positives). The reason is that it conservatively extends arrays towards the next variable below or above. Howard is very unlikely to underestimate the array size for compiler-generated code and we never encountered it in any of our tests, although there is no hard guarantee that we never will. Size underestimation is possible, but can happen only if the program accesses the array with multiple base pointers, and behaves consistently and radically different in all analysis runs from the production run.

Over a range of applications, Howard never underestimated an array's size and classified well over 80% of all arrays on the executed paths 'exactly right'—down to the last byte. These arrays represent over 90% of all array bytes. All remaining arrays are either not classified at all or overestimated and thus safe with respect to false positives.

We stressed earlier that Howard aims to err on the safe side, by overestimating the size of arrays to prevent false positives. The question is what the costs are of doing so. Specifically, one may expect an increase in false negatives. While true in theory, this is hardly an issue in practice. The reason is that *BinArmor* only misses buffer overflows that (1) overwrite values immediately following the real array (no byte beyond the (over-)estimation of the array 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 overflows that satisfy both conditions are rare. For instance, an overflow of a return value would never qualify, as the program always uses the return address separately. Overall, not a single vulnerability in Linux programs for which we could find an exploit qualified.

One final remark about array extraction and false positives; as mentioned earlier, *BinArmor* does not care which method is used to extract arrays and static extractors may be used just as well. However, this is not entirely true. Not underestimating array sizes is crucial.

We consider the problem of finding correct buffer sizes orthogonal to the binary protection mechanism offered by *BinArmor*. Whenever we discuss false positives in *BinArmor*, *we always assume that the sizes of buffers are not underestimated*.

3.2 Instructions to be Instrumented

When *BinArmor* detects buffers to be protected, it dynamically determines the instructions (array accesses), that need instrumenting. The process is straightforward: for each buffer, it dumps all instructions that access it.

Besides accesses, *BinArmor* also dumps all instructions that initialize or manipulate pointers that access arrays.

4 Code Coverage and Modes of Operation

Since *BinArmor* is based on dynamic analysis, it suffers from coverage issues—we can only analyze what we execute. Even the most advanced code coverage tools [9, 13] cover just a limited part of real programs. Lack of coverage causes *BinArmor* to miss arrays and array accesses and thus incur false negatives. Even so, *BinArmor* proved powerful enough to detect *all* attacks we tried (Section 8). What we really want to avoid are false positives: crashes on benign input.

In *BinArmor*, we instrument only those instructions that we encountered during the analysis phase. However, a program path executed at runtime, p_R, may differ from all paths we have seen during analysis A, $\{p_a\}_{a \in A}$, and yet p_R might share parts with (some of) them. Thus, an arbitrary subset of array accesses and pointer manipulations on p_R is instrumented, and as we instrument exactly those instructions that belong to paths in $\{p_a\}_{a \in A}$, it may well happen that we miss a pointer copy, a pointer initialization, or a pointer dereference instruction.

With that in mind, we should limit the color checks performed by *BinArmor* to program paths which use array pointers in ways also seen during analysis. Intuitively, the more scrupulous and fine-grained the color checking policy, the more tightly we need to constrain protected program paths to the ones *seen before*. To address this tradeoff, we offer two modes of *BinArmor* which impose different requirements for the selection of program paths to be instrumented, and offer protection at different granularities: coarse-grained *BA-objects* mode (Section 5), and fine-grained *BA-fields* mode (Section 6).

5 *BA-objects mode*: Object-level Protection

Just like other popular approaches, e.g., WIT [3] and BBC [4], BA-objects mode provides protection at the level of objects used by a program. To do so, *BinArmor* assigns a color to each buffer[1] on stack, heap, or in global memory. Then it makes sure that a pointer to an object of color X never accesses memory of color Y. This way we detect all buffer overflows that aim to overwrite another object in memory.

5.1 What is Permissible? What is not?

Figs. (2.a-2.b) show a function with some local variables, and Fig. (2.c) shows their memory layout and colors. In BA-objects mode, we permit memory accesses within objects, such as the two tick-marked accesses in Fig. (2.c). In the first case, the program perhaps iterates over the elements in the array (at offsets 4, 12, and 20 in the object), and dereferences a pointer 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` via a pointer to the last element of the array (offset 24). Although the program accesses a field beyond the array, it remains within the local variable `mystruct`, and (like WIT and other projects), we allow such operations in this mode. Such access patterns commonly occur, e.g., when a `memset()`-like function initializes the entire object.

However, *BinArmor* stops the program from accessing the `len` and `p` fields through a pointer into the structure. `len`, `p` and `mystruct` are separate variables on the stack, and one cannot be accessed through a pointer to the other. Thus, *BinArmor* in BA-objects mode stops inter-object buffer overflow attacks, but not intra-object ones.

5.2 Protection by Color Matching

BinArmor uses colors to enforce protection. It assigns colors to each word of a buffer[1], when the program allocates memory for it in global, heap, or stack memory. Each complete object gets one unique color. All memory which we do not protect gets a unique background color.

When the program assigns a buffer of color X to a pointer, *BinArmor* associates the same color with the register containing the pointer. The color 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 pointer is stored to memory, we also store its color to a memory map, to load it later when the pointer is restored.

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

Stale Colors and Measures to Rule out False Positives Due to lack of coverage, a program path at runtime may

[1] Or a struct containing the array as this mode operates on objects

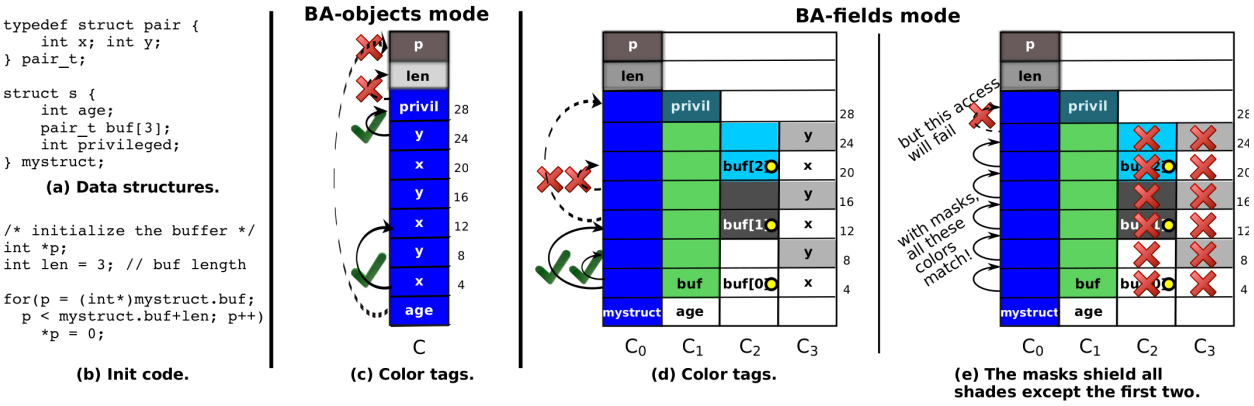

Fig. 2: *BinArmor* colors in BA-objects mode (c) and BA-fields modes (d,e) for sample data structures (a) and code (b).

lack instrumentation on some pointer manipulation instructions. This may lead to the use of a *stale* color.

Consider a function like `memcpy(src,dst)`. Suppose that *BinArmor* misses the `dst` buffer during analysis (it was never used), 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 color of that pointer. However, since the original push was not instrumented, *BinArmor* never set that color! If we are lucky, we simply find no color, and everything works fine. If we are unlucky, we pick up a stale color of whatever was previously on the stack at that position[2]. As soon as `memcpy()` dereferences the pointer, the color check fails and the program crashes.

BinArmor removes all false positives of this nature by adding an additional tag to the colors to indicate to which memory address the color 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 color 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 color corresponds to the memory to which the pointer refers. Specifically, *BinArmor* checks the colors on a dereference of `eax`, **iff** *(tag_{eax}==eax)*. Thus, it ignores stale colors and prevents the false positives.

Pointer Subtraction: What if Code is Color Blind? The colors assigned by *BinArmor* 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 "color blindness", and directly use a pointer to one object to access another object. The `strcat()` and `__strcpy_chk()` functions in current libc implementations on Linux are the best known examples: to copy a source to a destination string, they access both by the same pointer—adding the *distance* between them to access the remote string.

Our current solution is straightforward. When *BinArmor* detects a pointer subtraction, and later spots when the resultant distance is added to the subtrahend to access the buffer associated with the minuend pointer, it resets the color to reflect the remote buffer, and we protect dereferences in the usual way.

If more complex implementations of this phenomenon appear, we can prevent the associated dereferences from being checked at all. To reach the remote buffer, such scenarios have an operation which involves adding a value derived from the distance between two pointers. *BinArmor* would not include it in the set of instructions to be instrumented, so that the tag of the resultant pointer will not match its value, and the color check will not be performed. False positives are ruled out.

Other projects, like WIT [3] and the pointer analysis-based protection in [5], explicitly assume that a pointer to an object can only be derived from a pointer to the same object. In this sense, our approach is more generic.

5.3 Expect the Unexpected Paths

To justify that *BinArmor* effectively rules out false positives, we have to show that all program paths executed at runtime do not exhibit any false alerts. As we discussed in Section 4, a program path at runtime, p_R, may differ from all paths seen during analysis, while sharing parts with (some of) them. Thus, p_R may ac-

[2]There may be stale colors for the stack value, because it is not practical to clean up all colors whenever memory is no longer in use.

cess an array, while some of the instructions associated with these accesses are not instrumented. The question is whether p_R may cause false positives.

Suppose p_R accesses an array. If `arr` is a pointer to this array, 3 generic types of instruction might be missed, and thus not instrumented by *BinArmor*: (1) an `arr` initialization instruction, (2) an `arr` update/manipulation instruction, and (3) an `arr` dereference instruction.

The crucial feature of *BinArmor* which prevents false positives in cases (1) and (2) are the tags introduced in Section 5.2. They check whether the color associated with a pointer corresponds to the right value. In the case of a pointer initialization or a pointer update instruction missing, the pointer tag does not match its value anymore, its color is considered invalid, and it is not checked on dereferences. Finally, if an `arr` dereference instruction is not instrumented, it only means that the color check is not performed. Again, it can only result in false negatives, but never false positives.

6 *BA-fields mode*: a Colorful Armor

BA-objects mode and BA-fields mode differ significantly in the granularity of protection. Where BA-objects mode protects memory at the level of objects, BA-fields mode offers finer-grained protection—at the level of fields in structures. Thus, BinArmor in BA-fields mode stops not only inter-object buffer overflow attacks, but also intra-object ones. We shall see, the extra protection increases the chances of false positives which should be curbed.

6.1 What is Permissible? What is not?

Consider the structure in Fig. (2.a) with a memory layout as shown in Fig. (2.d). Just like in BA-objects mode, *BinArmor* now also permits legitimate memory accesses such as the two tick-marked accesses in Fig. (2.d).

But unlike in BA-objects mode, *BinArmor* in BA-fields mode stops the program from accessing the `privileged` field *via* a pointer into the array. Similarly, it prevents accessing the `x` field in one array element from the `y` field in another. Such accesses that do not normally occur in programs are often symptomatic of attacks[3].

6.2 Shaded Colors

BinArmor uses a *shaded* color scheme to enforce fine-grained protection. Compared to BA-objects mode, the color scheme used here is much richer. In Section 5, the whole object was given a single color, but in BA-fields mode, we add shades of colors to distinguish between individual fields in a structure. First, we sketch how we assign the colors. Next, we explain how they are used.

Since *BinArmor* knows the structure of an object to be protected, it can assign separate colors to each variable and to each field. The colors are hierarchical, much like real colors: lime green is a shade of green, and dark lime green and light lime green, are gradations of lime green, etc. Thus, we identify a byte's color as a sequence of shades: $C_0 : C_1 : .. : C_N$, where we interpret C_{i+1} as a shade of color C_i. Each shade corresponds to a nesting level in the data structure. This is illustrated in Fig. (2.d).

The base color, C_0, corresponds to the complete object, and is just like the color used by *BinArmor* in BA-objects mode. It distinguishes between individually allocated objects. At level 1, the object in Fig. (2.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 color C_i has a type flag indicating whether it is an array element shown in the figure as a dot (a small circle on the right).

We continue the coloring process, until we reach the maximum nesting level (in the figure, this happens at C_3), or exhaust the maximum color depth N. In the latter case, the object has more levels of nesting than *BinArmor* can accommodate in shades, so that some of the levels will collapse into one, 'flattening' the substructure. Collapsed structures reduce *BinArmor*'s granularity, but do not cause problems otherwise. In fact, most existing solutions (like WIT [3] and BBC [4]) operate only at the granularity of the full object.

Protection by Color Matching The main difference between the color schemes implemented in BA-objects mode and BA-fields mode is that colors are more complex now and include multiple shades. We need a new procedure to compare them, and decide what is legal.

The description of the procedure starts in exactly the same way as in BA-objects mode. When a buffer of color X is assigned to a pointer, *BinArmor* associates the same color with the register containing the pointer. The color 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 color to a memory map, to load it later when the pointer is restored to a register.

The difference from the BA-objects mode is in the color update rule: when the program dereferences a register, we update its color so that it now corresponds to

[3]Note: if they *do* occur, either `Howard` classifies the data structures differently, or *BinArmor* detects these accesses in the analysis phase, and applies *masks* (Section 6.2), so they do not cause problems.

the memory location associated with the register. The intuition is that we do not update colors on intermediate pointer arithmetic operations, but that the colors represent pointers used by the program to access memory.

From now on, *BinArmor* vets each dereference of the pointer to see if it is still in bounds. Vetting pointer dereferences is a matter of checking whether the color 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 color. Thus, leaving bases of structures and arrays blank guarantees that a pointer to them can access all internal fields of the object.

Finally, we handle the common case where a pointer to an array element derives from a pointer to another element of the array. Since array elements in Fig. (2c) differ in C_2, such accesses would normally not be allowed, but the dots distinguish array elements from structure fields. Thus we are able to grant these accesses. We now illustrate these mechanisms for our earlier 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 color 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, *BinArmor* checks whether the colors match. C_0 clearly matches, and since the pointer has only the C_1 color of the first array element, its color 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 colors 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.

The Use of Masks: What if Code is Color Blind? Programs do not always access data structures in a way that reflects the structure. They frequently use functions similar to `memset` to initialize (or copy) an entire object, with all subfields and arrays in it. Unfortunately, these functions do not heed the structure at all. Rather, they 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 `memset`-like function:

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

The code is clearly 'color blind', but while it violates the color protection, *BinArmor* should not raise an alert as the accesses are all legitimate. But it should not ignore color blindness either. For instance, the initialization of one object should not trample over *other* objects. Or inside the structure of Fig. (2.b): an initialization of the array should not overwrite the `privileged` field.

One (bad) way to handle such color 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., 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 initialization of (just) the buffer could overflow into the privilege field.

Instead, *BinArmor* exploits the shaded colors of Section 6.2 to implement *masks*. Masks shield code that is color blind from some of the structure's subtler shades. For instance, when the initialization code in Fig. (2.b) is applied to the array, 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 initialization routine writes over the entire object, we filter all shades except C_0, limiting all writes to this object.

Fig. (2.e) illustrates the usage of masks. The code on the left initializes the array in the structure of Fig. 2. By masking all colors beyond C_0 and C_1, all normal initialization code is permitted. If attackers can somehow manipulate the `len` variable, they could try to overflow the buffer and change the `privileged` value. However, in that case the C_1 colors do not match, and *BinArmor* will abort the program.

To determine whether a memory access needs masks (and if so, what sort), *BinArmor*'s dynamic analysis first marks all instructions that trample over multiple data structures as 'color blind' and determines the appropriate mask. For instance, if an instruction accesses the base of the object, *BinArmor* sets the masks to block out all colors except C_0. If an instruction accesses a field at the kth level in the structure, *BinArmor* sets the masks to block out all colors 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. *BinArmor* needs to decide *at runtime* which part of the shaded color to mask. In the above example, if the program initializes the whole structure, *BinArmor* sets the masks to block out all colors except C_0. If the same function is called to initialize the array, however, only C_2 and C_3 are shielded. To do so, *BinArmor*'s dynamic analysis tracks the *source* of the pointer used in the 'color 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. Observe that not all such instructions need masks. For instance, code that zeros all words in the object by adding increasing offsets to the *base* of the object, has no need for masks. After all, be-

cause of the blank shades the base of the object permits access to the entire object even without masks.

BinArmor enforces the masks when rewriting the binary. Rather than checking all shades, it checks only the instructions' *visible* colors for these instructions.

Pointer Subtraction As discussed in Section 5.2, some functions exhibit color blindness, and use a pointer to one object to access another. Both the problem and its solution are exactly the same as for BA-fields mode.

6.3 Why We do Not See False Positives

Given an accurate or conservative estimate of array sizes, the only potential cause of false positives is lack of coverage. As explained in Section 5, we do not address the array size underestimation here–we simply require either symbol tables or a conservative data structure extractor (Section 3). But other coverage issues occur regardless of symbol table availability and must be curbed.

Stale Colors and Tags In Section 5.2, we showed that lack of coverage could lead to the use of stale colors in BA-objects mode. Again, the problem and its solution are the same as for BA-fields mode.

Missed Masks and Context Checks Limited code coverage may also cause *BinArmor* to miss the *need* for masks and, unless prevented, lead to false positives. Consider again the example custom `memset` function of Section 6.2. The code is color 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 initialize 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 same function to access an array of 16-byte structures. Since it did not assign masks to this function originally, *BinArmor* now 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 color checking. When the program calls a function, *BinArmor* checks whether the function call also occurred at the same point during the analysis by examining the call stack. (In practice, we take the top 3 function calls.) Then, it decides whether or not to execute the instrumented version of the function. It performs color checking only for layouts of data structures we *have seen before*, so we do not encounter code that accesses memory in an unexpected way.

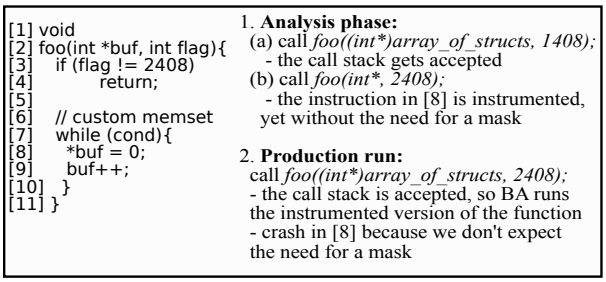

Fig. 3: BA-fields mode: a scenario leading to false positives.

6.4 Are False Positives Still Possible?

While the extra mechanism to prevent false positives based on context checks is effective in practice, it does not give any strong guarantees. The problem is that a call stack does not identify the execution context with absolute precision. Fig. 3 shows a possible problematic scenario. In this case, it should not be 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] initializing the `array_of_structs`, should we allow for an instrumented version of it at runtime. Observe that the scenario is fairly improbable. 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 very particular sets of arguments before it can lead to the awkward situation.

As we did not encounter false positives in *any* of our experiments, and BA-fields mode offers powerful, fine-grained protection, we think that the risk may be acceptable in application domains that can handle rare crashes.

7 Efficient Implementation

Protection by color matching combined with masks for color blindness allows *BinArmor* to protect data structures at a finer granularity than previous approaches. Even so, the mechanisms are sufficiently simple to allow for efficient implementations. *BinArmor* is designed to instrument 32-bit ELF binaries for the Linux/x86 platforms. Like Pebil [21], 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 *BinArmor* modifies the layout of a binary, and next present some details of the instrumentation. (For a full explanation refer to [32].)

7.1 Updated Layout of ELF Binary

To accommodate new code and data required by the instrumentation, *BinArmor* modifies the layout of an ELF binary. 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 6.3), and to assure that the resulting code works correctly—mainly by adjusting jump targets to reflect addresses in the updated code (refer to Section 7.2). To provide protection, *BinArmor* inserts a few additional segments in the binary: BA_Initialized_Data, BA_Uninitialized_Data, BA_Procedures, and BA_Code.

Both data segments–*BA_(Un)Initialized_Data*– store data structures that are internally used by *BinArmor*, e.g., data structures color 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 color of a pointer with the color of a memory location). Finally, the *BA_Code* segment is the pivot of the *BinArmor* protection mechanism—it contains the original program's functions instrumented to perform color checking.

```
_dereference_check_start:
# check whether tag value matches the pointer
cmp %edx, register_tag_edx
jne _dereference_check_end
[save %eax and %ebx used in instrumentation]
lea (%edx, %eax, 4), %ebx
call get_color_of_ebx ; loaded to %bx
mov register_color_edx, %ax
call color_match ; compare colors in %ax and %bx
cmpl $0, %eax    ; check result
je _dereference_ok

_dereference_bad:
["crash"]

_dereference_ok:
[restore %eax and %ebx used in instrumentation]

_dereference_check_end:
movl $0x1234, (%edx, %eax, 4); execute original instr
```

Fig. 4: Instrumentation for an array pointer dereference (with 16b colors and tags). The original instruction is `mov 0x1234,(%edx,%eax,4)`. We replace it by code similar to that presented in the figure (but more efficient).

7.2 Instrumentation Code

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

To provide protection, *BinArmor* reorganizes code at the instruction level. We do not need to know function boundaries, as we instrument instructions which were classified as array accesses, along with pointer move or pointer initialization instructions, during the dynamic analysis phase. We briefly describe the main steps taken by *BinArmor*: (1) inserting trampolines and method selector, (2) code relocation, (3) inserting instrumentation.

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 6.3), and then jump to it. In *BinArmor*, 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. *BinArmor*'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 [31]: they are resolved at runtime, by using arrays maintaining 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 color 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 color of the pointer in a color map, and make sure that the store operation is legal. For an efficient instrumentation, we have developed a large number of carefully tailored snippets.

Colors 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 color tags when needed. The same memory optimization is often used in dynamic taint analysis. In the implementation evaluated in Section 8, we use 32 bit colors with four shades and 16 bit tags.

Fig. 4 shows an example of an array dereference in the binary hardened by *BinArmor*. The code is simplified for brevity, but otherwise correct. We see that each array dereference incurs some extra instructions. If the colors do not match, the system crashes. Otherwise, the dereference executes as intended. We stress that the real implementation is more efficient. For instance, adding two `call` instructions would be extremely expensive. In reality, *BinArmor* uses code snippets tailored to performance

8 Evaluation

We evaluate *BinArmor* on performance and on effectiveness in stopping attacks. As the analysis engine is based on the Qemu processor 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 optimization levels. All results presented in this section are for binaries compiled with `gcc-4.4 -O2` and without symbols, i.e., completely stripped. We reconstruct the symbols using Howard [29].

Performance To evaluate the performance of *BinArmor* operating in BA-fields mode[4], we compare the speed of instrumented (armored) binaries with that of unmodified implementations. Our 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 has 4GB of memory. For our experiments we used an Ubuntu 10.10 install. We ran each test 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 *BinArmor* 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 tests should represent close to worst-case scenarios for *BinArmor*.

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.

Fig. 5 shows that for real I/O-intensive client-side applications like `wget` and `htget` the slowdown is negligible, while `gzip` incurs a slow-down of approximately 1.7x. As `gzip` is a very expensive test for *BinArmor*, the slow-down was less than we expected. 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 16% for large objects. In networking applications I/O dominates the performance and the overhead of *BinArmor* is less important.

Fig. 6 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 optimized *BinArmor*, these results are quite good. Some of the tests incurred a fairly minimal slow-down. 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. They really represent the worst cases for *BinArmor*.

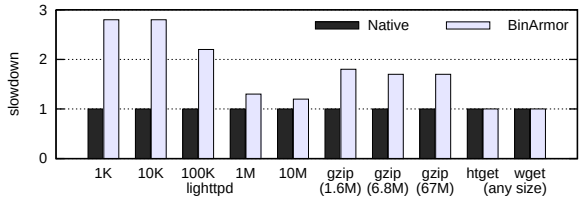

Fig. 5: Performance overhead for real world applications: *lighttpd* – for 5 object sizes (in connections/s as measured by *httperf*), *gzip* – for 3 object sizes, *htget* and *wget*.

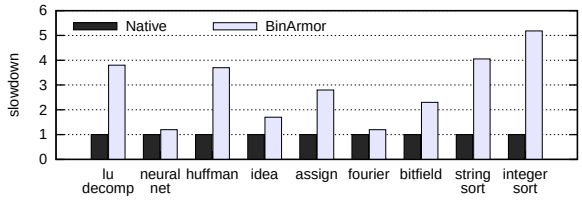

Fig. 6: Performance overhead for the compute-intensive *nbench* benchmark suite.

Effectiveness Table 1 shows the effectiveness of *BinArmor* in detecting attacks on a range of real-life software vulnerabilities. Specifically, these attacks represent *all* vulnerabilities on Linux programs for which we found working exploits. *BinArmor* operating in either mode 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. While control flow diversions would trigger alerts also on taint analysis systems like Argos [25] and Minemu [8], to the best of our knowledge, no other security measure for stripped binaries would be able to detect such attacks. As mentioned earlier, security experts expect non-control data attacks to become even more important attack vectors in the near future [12, 30].

[4]The reason is that BA-fields mode is the most fine-grained, although in practice, the performance of BA-objects mode is very similar.

Application	Vulnerability type	Security advisory
Aeon 0.2a	Stack overflow	CVE-2005-1019
Aspell 0.50.5	Stack overflow	CVE-2004-0548
Htget 0.93 (1)	Stack overflow	CVE-2004-0852
Htget 0.93 (2)	Stack overflow	
Iwconfig v.26	Stack overflow	CVE-2003-0947
Ncompress 4.2.4	Stack overflow	CVE-2001-1413
Proftpd 1.3.3a	Stack overflow	CVE-2010-4221
bc-1.06 (1)	Heap overflow	Bugbench [22]
bc-1.06 (2)	Heap overflow	Bugbench [22]
Exim 4.41	Heap overflow*	CVE-2010-4344
Nullhttpd-0.5.1	Heap overflow†	CVE-2002-1496
Squid-2.3	Heap overflow†	Bugbench [22]

* A non-control-diverting attack. † A reproduced attack.

Table 1: Tested vulnerabilities: *all attacks were stopped by BinArmor*, including the attack on non-control data.

9 Related Work

The easiest way to prevent memory corruptions is to do so at the source level, using a safe language or compiler extension. However, as access to source code or recompilation is often not an option, many binaries are left unprotected. Our work is about protecting binaries. Since it 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. Cyclone [20] and CCured [14] show that similar protection also fits dialects of C—although the overhead is not always negligible. Better still, data flow integrity (DFI) [10], write integrity testing (WIT) [3], and baggy bounds checking (BBC) [4] are powerful protection approaches against memory corruptions for unmodified C.

BinArmor was inspired by the WIT compiler extension—a defense framework that marries immediate (fail-stop) detection of memory corruption to excellent performance. WIT assigns a color to each object in memory and to each write instruction in the program, so that the color of memory always matches the color of an instruction writing it. Thus all buffers which can be potentially accessed by the same instruction share the same color. WIT employs points-to analysis to find the set of objects written by each instruction. If several objects share the same color, WIT might fail to detect attacks that use a pointer to one object to write to the other. To get a grasp of the precision, we implemented points-to analysis ourselves, and applied it to global arrays in `gzip-1.4`. Out of 270 buffers, 124 have a unique color, and there are two big sets of objects that need to share it: containing 64 and 68 elements. (We assume that we provide templates for the `libc` functions. Otherwise, the precision is worse.) To deal with these problems, WIT additionally inserts small guards between objects, which cannot be written by any instruction. They provide an extra protection against sequential buffer overflows. *BinArmor* tracks colors of objects dynamically, so each array is assigned a unique color.

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 [23]. Again, SoftBound requires access to source code.

BinArmor's protection resembles that of WIT, but without requiring source code, debugging information, or even symbol tables. Unlike WIT, it protects at the granularity of subfields in C `structs`. It prevents not just out-of-bounds writes, as WIT does, but also reads. As a drawback, *BinArmor* may be less accurate, since dynamic analysis may not cover the entire program.

Protection of binaries Arguably some of the most popular measures to protect against memory corruption are memory debuggers like Purify and Valgrind [24]. 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, *BinArmor* is much faster and requires neither.

One of the most advanced approaches to binary protection is XFI [19]. Like memory debuggers, XFI requires symbol tables. Unlike memory debuggers, DTA, or *BinArmor*, XFI's main purpose is to protect host software that loads modules (drivers in the kernel, OS processes, or browser modules) and it requires explicit support from the hosting software–to grant the modules access to a slice of the address space. It offers protection by a combination of control flow integrity, stack splitting, and memory access guards. Memory protection is at the granularity of the module, and for some instructions, the function frame. The memory guards will miss most overflows that modify non-control data.

An important class of approaches to detect the *effects* of memory corruption attacks is based on dynamic taint analysis (DTA) [15]. DTA does not detect the memory corruption itself, but may detect malicious control flow transfers. Unfortunately, the control flow transfer occurs at a (often much) later stage. With typical slowdowns of an order of magnitude, DTA in software is also simply *too expensive* for production systems.

Non-control data attacks are much harder to stop [12]. [11] pioneered an interesting form of DTA to detect some of these attacks: pointers become tainted if their values

are influenced by user input, and an alert is raised if a tainted value is dereferenced. However, pointer taintedness for detecting non-control data attacks is shown to be impractical for complex architectures like the x86 and popular operating systems [28]. The problems range from handling table lookups to implicit flows and result in false positives and negatives. Moreover, by definition, pointer taintedness cannot detect attacks that do not dereference a tainted pointer, such as an attack that would overwrite the `privileged` field in Fig. (2a).

10 Discussion

Obviously, *BinArmor* is not flawless. In this section, we discuss some generic limitations.

With a dynamic approach, *BinArmor* 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, the corresponding instructions are not instrumented either. Combined with the tags (Section 5.2), this lack of accuracy can only cause false negatives, but never false positives. In practice, as we have seen in Section 8, *BinArmor* 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 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 cause a false positive. Still, it is unlikely, because to do so, the behavior during analysis should also be significantly different from that during the production run. In our view, the risk is acceptable for software deployments that can tolerate rare crashes.

We have implemented two versions of *BinArmor*: BA-objects mode, and 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 — *BinArmor* protects individual fields within structures — we again think that the risk is acceptable.

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, *BinArmor* would interpret it as normal code behavior 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 such real problems in the testing phase, this can only be beneficial for the quality of software.

11 Future work

BinArmor's two main problems are accuracy (in terms of false negatives and false positives) and performance (in terms of the slowdown of the rewritten binary). In this section, we discuss ways to address these problems.

First, the root cause of *BinArmor*'s false negative and false positive problems is the lack of code coverage. Our next target, therefore, is to extend the paths covered dynamically by means of static analysis. For instance, we can statically analyze the full control flow graphs of all functions called at runtime. Static analysis in general is quite hard, due to indirect calls and jumps, but within a single function indirect jumps are often tractable (they are typically the result of `switch` statements that are relatively easy to handle).

Second, the cause of *BinArmor*'s slowdown is the instrumentation that adds overhead to every access to an array that *BinArmor* discovered. We can decrease the overhead using techniques that are similar to those applied in WIT. For instance, there is no need to perform checks on instructions which calculate the address to be dereferenced in a deterministic way, say at offset 0x10 from a base pointer. Thus, our next step is to analyze the instructions that are candidates for instrumentation and determine whether the instrumentation is strictly needed.

12 Conclusions

We described a novel approach to harden binary software proactively against buffer overflows, without access to source or symbol tables. Besides attacks that divert the control flow, we also detect attacks against non-control data. Further, we demonstrated that our approach 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. On the downside, the overhead of our approach in its current form is quite high—making it unsuitable for many application domains today. However, we also showed that significant performance optimizations may still be possible. It is our view that protection at the binary level is important for dealing with real threats to real and deployed information systems.

Acknowledgements

This work is supported by the European Research Council through project ERC-2010-StG 259108-ROSETTA and the EU FP7 SysSec Network of Excellence. The authors are grateful to David Brumley and his team for pro-

viding us with several local exploits, and to Erik Bosman and Philip Homburg for their work on the Exim exploit.

References

[1] BYTE Magazine nbench benchmark. `http://www.tux.org/~mayer/linux/bmark.html`.

[2] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-Flow Integrity. In *Proceedings of CCS* (2005).

[3] AKRITIDIS, P., CADAR, C., RAICIU, C., COSTA, M., AND CASTRO, M. Preventing memory error exploits with WIT. In *Proc. of the IEEE Symposium on Security and Privacy*, S&P'08.

[4] AKRITIDIS, P., COSTA, M., CASTRO, M., AND HAND, S. Baggy Bounds Checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th Usenix Security Symposium* (2009), USENIX-SS'09.

[5] AVOTS, D., DALTON, M., LIVSHITS, V. B., AND LAM, M. S. Improving software security with a C pointer analysis. In *Proc. of the 27th Intern. Conf. on Software Engineering (ICSE)* (2005).

[6] BELLARD, F. QEMU, a fast and portable dynamic translator. In *USENIX 2005 Annual Technical Conference*, ATEC '05.

[7] BHATKAR, S., DUVARNEY, D. C., AND SEKAR, R. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proc. of USENIX-SS* (2003).

[8] BOSMAN, E., SLOWINSKA, A., AND BOS, H. Minemu: The Worlds Fastest Taint Tracker. In *Proceedings of 14th International Symposium on Recent Advances in Intrusion Detection* (2011), RAID 2011.

[9] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of OSDI* (2008).

[10] CASTRO, M., COSTA, M., AND HARRIS, T. Securing software by enforcing data-flow integrity. In *Proc. of OSDI* (2006).

[11] CHEN, S., XU, J., NAKKA, N., KALBARCZYK, Z., AND IYER, R. K. Defeating memory corruption attacks via pointer taintedness detection. In *Proc. of DSN* (2005).

[12] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data attacks are realistic threats. In *Proc. of 14th USENIX Security Symposium* (2005), SSYM'05.

[13] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2E: A platform for in vivo multi-path analysis of software systems. In *Proc. of ASPLOS* (2011).

[14] CONDIT, J., HARREN, M., MCPEAK, S., NECULA, G. C., AND WEIMER, W. CCured in the real world. In *Proc. of POPL* (2003).

[15] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., AND BARHAM, P. Vigilante: end-to-end containment of internet worms. In *Proc. of SOSP* (2005).

[16] COWAN, C., PU, C., MAIER, D., HINTONY, H., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proc. of USENIX-SS* (1998).

[17] CWE/SANS. TOP 25 Most Dangerous Software Errors. `www.sans.org/top25-software-errors`, 2011.

[18] ELIAS LEVY (ALEPH ONE). Smashing the stack for fun and profit. *Phrack 7*, 49 (1996).

[19] ERLINGSSON, U., VALLEY, S., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: software guards for system address spaces. In *Proc. of OSDI* (2006).

[20] JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CHENEY, J., AND WANG, Y. Cyclone: A safe dialect of C. In *USENIX 2002 Annual Technical Conference*, ATEC '02.

[21] LAURENZANO, M., TIKIR, M. M., CARRINGTON, L., AND SNAVELY, A. PEBIL: Efficient static binary instrumentation for Linux. In *Proc. of ISPASS* (2010).

[22] LU, S., LI, Z., QIN, F., TAN, L., ZHOU, P., AND ZHOU, Y. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools* (2005).

[23] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. SoftBound: highly compatible and complete spatial memory safety for C. In *Proc. of PLDI'09*.

[24] NETHERCOTE, N., AND SEWARD, J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. of the 3rd Intern. Conf. on Virtual Execution Environ.* (2007), VEE.

[25] PORTOKALIDIS, G., SLOWINSKA, A., AND BOS, H. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006* (2006), EuroSys '06.

[26] REPS, T., AND BALAKRISHNAN, G. Improved memory-access analysis for x87 executables. In *Proc. of ETAPS* (2008).

[27] SLOWINSKA, A., AND BOS, H. The Age of Data: Pinpointing Guilty Bytes in Polymorphic Buffer Overflows on Heap or Stack. In *Proc. of ACSAC* (2007).

[28] SLOWINSKA, A., AND BOS, H. Pointless tainting?: evaluating the practicality of pointer tainting. In *Proc. of EuroSys* (2009).

[29] SLOWINSKA, A., STANCESCU, T., AND BOS, H. Howard: a dynamic excavator for reverse engineering data structures. In *Proceedings of NDSS* (2011).

[30] SOTIROV, A. Modern exploitation and memory protection bypasses. USENIX Security invited talk, `www.usenix.org/events/sec09/tech/slides/sotirov.pdf`, August 2009.

[31] SRIDHAR, S., SHAPIRO, J. S., AND NORTHUP, E. HDTrans: An open source, low-level dynamic instrumentation system. In *Proc. of the 2nd Intern. Conf. on Virtual Execution Environ.* (2006).

[32] STANCESCU, T. BodyArmor: Adding Data Protection to Binary Executables. Master's thesis, VU Amsterdam, 2011.

[33] TEAM, P. Design and implementation of PAGEEXEC. `http://pax.grsecurity.net/docs/pageexec.old.txt`, November 2000.