CERIAS Tech Report 2013-3

BISTRO: Binary Component Extraction and Embedding for Software Security Applications

by Zhui Deng, Xiangyu Zhang, Dongyan Xu
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

BISTRO: Binary Component Extraction and Embedding for Software Security Applications

Zhui Deng, Xiangyu Zhang, and Dongyan Xu

Department of Computer Science, Purdue University, West Lafayette, IN 47907, USA
{deng14,xyzhang,dxu}@cs.purdue.edu

Abstract. In software security and malware analysis, researchers often need to directly manipulate binary program – benign or malicious – without source code. A useful pair of binary manipulation primitives are binary functional component *extraction* and *embedding*, for extracting a functional component from a binary program and for embedding a functional component in a binary program, respectively. Such primitives are applicable to a wide range of security scenarios such as legacy program hardening, binary semantic patching, and malware function analysis. Unfortunately, existing binary rewriting techniques are inadequate to support binary function carving and embedding. In this paper, we present BISTRO, a system that supports these primitives *without* symbolic information, relocation information, or compiler support. BISTRO preserves functional correctness of both the extracted functional component and the stretched binary program (with the component embedded) by properly patching them using – interestingly – the same technique and algorithm. We have implemented an IDA Pro-based prototype of BISTRO and evaluated it using real-world Windows software. Our results show that BISTRO performs these primitives efficiently; Each stretched binary program only incurs small time and space overhead. Furthermore, we demonstrate BISTRO's capabilities in various security applications.

1 Introduction

In software security and malware analysis, researchers often need to manipulate binary code – benign or malicious – without source code or symbolic information. One pair of complementary binary manipulation primitives is to (1) extract a re-usable functional component from a binary program and (2) embed a value-added functional component in an existing binary program. We call the binary manipulation primitives described above binary component *extraction* and *embedding*. These primitives are useful in a wide range of software security and malware analysis scenarios. In *security hardening of legacy binaries*, binary component embedding enables the retrofitting of legacy or close-source software with a third-party functional component that performs a value-added security function such as access control policy enforcement. In *binary semantic patching*, binary programs from different vendors may leverage the same functional component. Suppose one vendor identifies a vulnerability in such a component and releases a patched version for its own program; whereas other vendors are not aware of the vulnerability or have not patched their products. We can apply binary component extraction to carve out the patched component from a patched program and replace the

vulnerable version of the same component in an un-patched program using binary component embedding. In *malware analysis*, binary component extraction and embedding supports "plug and play" of malicious functions extracted from malware captured in the wild. One can even "stitch" multiple extracted malware functions to compose a new piece of malware – a capability that may help enable *strategic defence* in cyber warfare.

Enabling binary component extraction and embedding poses significant challenges. Brute force extraction and insertion of binary functions will most likely fail. Instead, both the extracted component and the target binary program need to be carefully transformed. For example, instructions in the target binary need to be shifted to create space for the embedded function; when a function is extracted from its origin binary, the instructions in it need to be re-positioned and re-packaged; accesses to global variables need to be re-positioned; function pointers need to be properly handled; and indirect jumps/calls need to have their target addresses recalculated. These problems are especially challenging when the binary component or the target binary program is *not relocatable*, which is often the case when dealing with legacy or malware binaries.

Despite advances in binary instrumentation and rewriting, existing techniques are inadequate to address the binary component extraction and embedding challenges. Dynamic binary instrumentation tools such as PIN [1], Valgrind [2], DynamoRIO [3] and QEMU [4] perform instrumentation only when a binary program is executed on their infrastructures. They do not generate an instrumented, stand-alone version of the binary for production runs. Static binary rewriting tools such as Diablo [5], Alto [6], Vulcan [31], and Atom [8] can generate instrumented, stand-alone binaries. However, they require symbolic information or that the binaries be generated by special compilers.

More lightweight techniques exist that do not require symbolic information or special compilers [9–14]. Among these techniques, some create *trampolines* at the end of the target binary program in which instrumentation is placed and then use control flow *detours* to access the trampolines [9–11]. The others duplicate the body of the target binary program in its virtual memory space and only the replica is instrumented. The original binary body is retained in its original position to provide a kind of control flow forwarding mechanism [12–14]. However, none of these techniques supports extraction of binary component or implanting an extracted component to another binary. Many of them cause substantial space/performance overhead. To the best of our knowledge, none of them has been successfully applied to large-scale Windows applications or kernel code. A more detailed comparison is presented in Section 3.

Recently, researchers proposed approaches that focus on identification, extraction and reuse of components from binaries. Inspector Gadget [30] performs dynamic slicing to identify and extract components from malware. The extracted component might have incomplete code path coverage due to the limitation of dynamic analysis. BCR [17] adopts a combination of static and dynamic approach to extract a function from a binary. However, it uses labels to represent jump/call targets, thus does not preserve the semantic of indirect jumps/calls. ROC [24] uses dynamic slicing to identify reusable functional components in a binary but does not extract them. None of them supports reusing extracted components to enhance legacy binaries. Moreover, they could not extract components from non-executable binaries (e.g., malware corpse) due to the use of dynamic analysis.

In this paper, we present BISTRO, a systematic approach to binary functional component extraction and embedding. BISTRO automatically performs the following: (1) extracting a functional component, with its instructions and data section entries non-contiguously located in the virtual address space, from an original binary program and (2) embedding a binary component of any size at any user-specified location in a target binary program, without requiring symbolic information, relocation information, or compiler support. For both extraction and embedding, BISTRO preserves the functionalities of the target binary program and the extracted component by accurately patching them – using the same approach and technique. BISTRO performs extraction and embedding operations efficiently and the "stretched" target binary program after embedding only incurs small time and space overhead.

We have developed a prototype of BISTRO as a IDA-Pro [22] plugin. We have conducted extensive evaluation and case studies using real-world Windows-based applications (including large-scale software such as Firefox and Adobe Reader), kernel-level device drivers, and malware. Our evaluation (Section 7) indicates BISTRO's efficiency and precision in patching the extracted components and target binary programs. Moreover, the stretched target binary program incurs small performance overhead (1.9% on average) and space overhead (10.9% on average). We have applied BISTRO to the following usage cases: (1) We carve out patched components from a binary and use them to replace their un-patched versions in other application binaries, achieving binary semantic patching (Section 7.2); (2) We stitch malicious functions from an un-executable Conficker worm [15] sample and compose a new, executable malware (Section 7.3); and (3) We demonstrate the realistic threat of *trojan-ed* device drivers with malicious rootkit functions embedded in benign driver – using real-world drivers and rootkits (Section 7.4).

2 Overview and Assumptions

An overview of BISTRO is shown in Figure 1. BISTRO has two key components: *binary extractor* and *binary stretcher*.

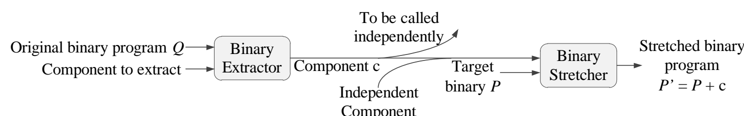

Fig. 1. Overview of BISTRO.

- The binary extractor is responsible for extracting a designated functional component c from an original binary program Q. c includes both the code and data of the functional component. The extractor does so by removing the unwanted code and data from Q and then collapsing the remaining data and code into a re-usable component c that occupies a contiguous virtual address region. More importantly, the instructions in c are properly patched for repositioning. We note that c can either be called as a library function or be embedded directly in another binary program.
- The binary stretcher is responsible for stretching the target binary program P to make "room" (holes in its address space) to embed a function component. As shown in Figure 1, the stretcher takes the target binary P and the to-be-embedded component c as input; stretches P, and patches the code in P to allow the embedding of

c. The output of the stretcher is a "stretched" binary program $P' = P + c$ that is ready for execution.

Summary of Enabling Techniques. Both the binary extractor and stretcher are based on the same binary stretching algorithm (Section 4). The overarching idea is to shift instructions for creating space (by stretcher) or squeezing out unwanted space (by extractor). The algorithm focuses on patching the control transfer and global data reference instructions by precisely computing the offsets they need to be adjusted. For instance, if a component with size $|c| = n$ is inserted, all the original instructions following the insertion point will be shifted by n bytes, and control transfers to any of the shifted instructions need to be incremented by n.

To address the challenge of handling indirect calls and call back functions invoked by external libraries, we develop another algorithm (Section 5.1) that stretches a subject binary at the original entries of functions that are potential targets of indirect calls, creating small holes (usually a few bytes) to hold a long jump instruction to forward any calls to those functions to their shifted locations. These holes must not be shifted by any stretching/shrinking operations. They always stay in their original positions and we thus call them "*anchors*". Our algorithm precisely takes into account these anchors when performing stretching/shrinking. To handle indirect jumps, we leverage an efficient perfect hashing scheme to translate jump targets dynamically. We use these approaches to patch indirect jumps/calls in both the component and the target binary.

Assumptions. We make the following assumptions (and hence stating the *non-goals* of BISTRO): (1) The user, not BISTRO, will predetermine the semantic appropriateness of embedding functional component c in target program P. Furthermore, he/she will decide the specific location to insert the component. This can be practically done by performing reverse engineering on P. For example, to harden P with some security policy enforcement mechanism based on control flow [7], the user can reconstruct the control flow graph of P, collect its dominance and post-dominance information, and decide proper locations to insert c. (2) The identification of component c in the original program Q, including its code and data, is done a priori by the user through manual or automated techniques, such as Inspector Gadget [30], binary slicing [16], binary differencing [32], and BCR [17]. While we will present our experience with functional component identification in our case studies (Section 7), the identification technique itself is *outside* the scope of this paper. (3) Binaries can be properly disassembled (e.g., by IDA-Pro) *before* being passed to BISTRO. This assumption is supported by the large number of real-world, off-the-shelf binaries in our experiments. Although we currently do not handle obfuscated or self-modifying binaries, we note that, in addition to IDA-Pro, other conservative disassembling [14, 36] and unpacking [35] tools can also be used as the pre-processor of BISTRO to handle more sophisticated binaries.

3 Problems with Existing Techniques

Before presenting BISTRO, we take an in-depth look at the existing binary rewriting techniques and explain their limitations for binary component extraction and implanting. Our discussion only focuses on existing techniques that work on stripped binaries

without debugging symbols or relocation information, which we classify into two categories: *detour-based rewriting* and *duplication-based rewriting*.

Detour-Based Rewriting. Detour-based binary rewriters [9–11] create control flow detours from the original code to the instrumentation. More specifically, to instrument an instruction, the rewriter replaces the instruction with a detour to a *trampoline*, where the instrumentation code is located. At the end of the instrumentation, the control flow jumps back to the original code. For example, as shown in Figure 2, suppose we need to count the number of times function *Func_B* gets called. The instrumentation involves replacing the three instructions at the entry of *Func_B* with a jump instruction, which detours the control flow to the trampoline code *Tramp_B* placed at the end of the binary. *Tramp_B* will increment the counter, execute the three instructions that were replaced, and jump back to the instruction right after the instrumentation point. Detour-based rewriting works well when the number of instrumentation points is small and the instrumentation code is simple.

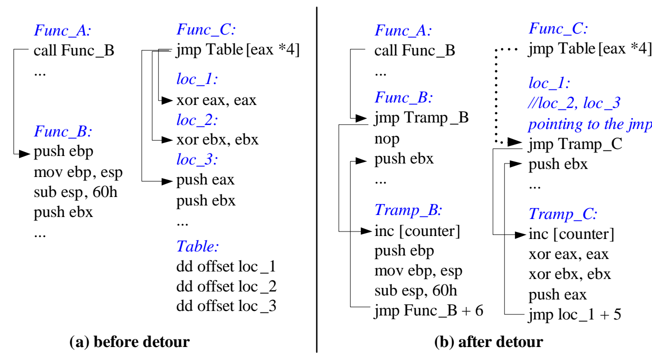

Fig. 2. Examples of detour using trampoline. Arrows show the direction of control-flow. The dashed line shows ill-formed control-flow.

The main problem with detour-based rewriting is that it requires the instructions to be replaced at the instrumentation point be *relocatable*. To understand this problem, let us look at the case where we try to perform the same instrumentation at *loc_1* in function *Func_C*. There is an indirect jump in *Func_C* using a jump table *Table*, with potential targets *loc_1*, *loc_2*, and *loc_3* – such structure is often generated by the compiler to represent a `switch` statement. We replace the instructions at *loc_1* with the unconditional jump instruction, which will take 5 bytes, and the instructions being replaced will be relocated to the trampoline code in *Tramp_C*. However, now *loc_2* and *loc_3* will point to the middle of the jump instruction, causing ill-formed control flow. While it seems that one can patch the jump table in this example, the problem becomes much more difficult to fix if an overwritten instruction is the computed target of some indirect jump/call, as the target may be stored in some data structure fields or generated dynamically via complex computation. One might also consider using software breakpoint (a special one-byte instruction) instead of jump instruction to detour the control flow. However, software breakpoints incur significant performance overhead.

Duplication-Based Rewriting. Recently, duplication-based binary rewriters [12–14] are proposed. These techniques make a copy of the original code sections and then instrument the copy. The instrumented copy is executed in cooperation with the original

code. In particular, to preserve control flow correctness, branch targets in the instrumented copy need to be patched. To handle indirect calls and jumps, jump/call targets *in the original code sections* are replaced with redirection to their new targets in the instrumented copy. The original data sections are also reused by the new copy.

The first problem with duplication-based techniques is their excessive space requirement. For the code sections, the space has to be almost *doubled.* Second, it is difficult to precisely determine all the possible targets of each indirect jump/call before instrumentation [13, 14] (for the sake of inserting redirection). Using a conservative analysis may result in large sets of potential targets, leading to runtime inefficiency [13].

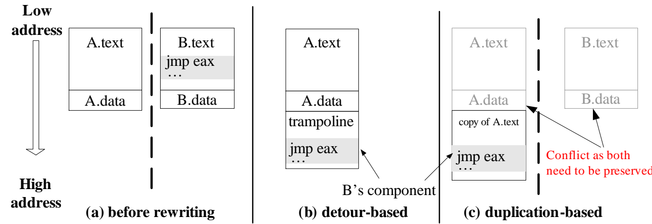

Fig. 3. Difficulties when transplanting a component (shaded) from binary B into binary A.

Most importantly, neither duplication-based nor detour-based rewriting supports binary component transplanting – the main application scenario of BISTRO. Consider the example in Figure 3. Suppose we wish to extract a component from B and insert it into A. The component is shaded in (a) and contains an indirect jump instruction. In (b), a detour-based technique is applied and the component is inserted in the trampoline at the end of A's body. However, the indirect jump in the component will not work properly, jumping to some irrelevant location in A instead of to the correct target as if in B. In (c), a duplication-based techniques is applied. The text section of A is duplicated and the component is inserted to the replica. However, to ensure correctness of the indirect jump in A, it is necessary to preserve B's original text section *at the same location as in B* and insert redirection at the original possible targets of the indirect jump. Unfortunately, the position of B's text section conflicts with that of A's in the virtual address space. Such conflict is highly likely to happen in practice: by default, common compilers choose to select the same base loading address when generating executable binaries: 0x400000 for Windows PE binaries and 0x8048000 for Linux ELF binaries. This means that most binaries will overlap from the very beginning when loaded into memory.

4 Basic Algorithm for Binary Extraction/Stretching

In this section, we present the basic algorithm (Algorithm 1) executed by both the binary extractor and stretcher of BISTRO (Section 2). For the time being, we assume (1) there is no indirect control transfer and (2) global data is directly referenced in an instruction using its address.

The algorithm takes the subject binary and a list of virtual address intervals called *snippets* representing (1) the holes to be created in the binary in the case of stretching *or* (2) the unwanted instruction/data blocks in the case of shrinking (extraction). First, for each byte in the binary, the algorithm computes a mapping between its original index

in the binary and its corresponding index after the snippets are inserted/removed. After that, the algorithm patches address operands in control transfer and global data reference instructions, and copy each byte to its mapped location according to the mapping.

Practical Challenges. To make BISTRO work for real-world large-scale software, we still need to overcome a number of practical challenges *not* addressed by Algorithm 1.

- The target of an indirect control transfer instruction (e.g., `call eax`) is computed during execution and takes different values depending on the execution path. Such an instruction cannot be patched by Algorithm 1.

Algorithm 1 Basic binary stretching/shrinking algorithm

Input: P – the subject binary; it has $size$ and $base_addr$ fields to represent its size when loaded into memory and base loading address, respectively.

M – a list of address intervals represent code/data to be inserted/removed, sorted increasingly by their location; each interval has $addr$, len and $type$ fields, denoting the location, size and type respectively. Type "INSERT" means inserting right before $addr$; "REMOVE" means the block starting at $addr$ is to be removed.

Output: P' – the stretched/shrunk binary.

```
1: function BASICSTRETCHING(P, M)
2:   map ← ComputeMapping(P, M)
3:   P' ← PatchTarget(P, map)
4: end function

5: function COMPUTEMAPPING(P, M)
6:   offset ← 0
7:   m ← M.begin()
8:   for i ← 0 to P.size do
9:     if m.addr == P.base_addr + i then
10:      if m.type == INSERT then
11:        offset ← offset + m.len
12:      else if m.type == REMOVE then
13:        offset ← offset − m.len
14:        i ← i + m.len
15:      end if
16:      m ← M.next()
17:    end if
18:    map[i] ← i + offset
19:  end for
20:  return map
21: end function

22: function PATCHTARGET(P, map)
23:   P' ← {nop, nop, ..., nop}
24:   for i ← 0 to P.size do
25:     if map[i] ≠ ⊥ then
26:       if P[i] is instruction then
27:         ins ← P[i]
28:         for each data address operand op in ins do
29:           target ← op.addr − P.base_addr
30:           off ← map[target] − target
31:           op.addr ← op.addr + off
32:         end for
33:         if ins is near call/jump then
34:           target ← i + ins.len + ins.target
35:           off ← map[target] − target
36:           off' ← map[i + ins.len] − (i + ins.len)
37:           ins.target ← ins.target + off − off'
38:         else if ins is far call/jump then
39:           target ← ins.target − P.base_addr
40:           off ← map[target] − target
41:           ins.target ← ins.target + off
42:         end if
43:         P'[map[i]] ← ins
44:       else if P[i] is data then
45:         P'[map[i]] ← P[i]
46:       end if
47:     end if
48:   end for
49:   return P'
50: end function
```

- Function pointers may be present in data or in an instruction as an immediate operand. These function pointers might be passed as parameters to external libraries as callback functions. If a function is relocated due to stretching, the external library will call back to a wrong address. All these have to be properly handled to ensure correctness of binary stretching/shrinking.
- Accesses to global data may be via data pointers (e.g., `mov ebx, ptr_data; mov eax, [ebx+4]`). The addresses of data are not known until runtime. These instructions cannot be patched using Algorithm 1 either.

We will present our solutions to these challenges in the following sections.

5 Handling Indirect Control Transfer

Handling indirect jumps and calls is one of the key challenges in the design of BISTRO. The difficulty is that the jump/call target cannot be known statically and thus is hard to patch. To understand the challenge, consider the example in Figure 4. On the left, there are three objects that are connected via pointers, with two of type `B` and one of type `A`. On the right, part of function `foo()` is presented. The function takes two parameters stored in `eax` and `ebx` denoting pointer values. These two pointers may be aliased to each other. If so, `ecx` at 0x4302B2 gets the value 0x400340 defined at 0x4302A0, and then eventually the call instruction at 0x4302BD acquires the function pointer 0x444142. However, if the two pointer parameters are not aliased, the call instruction may get a completely different target, making statically patching it difficult.

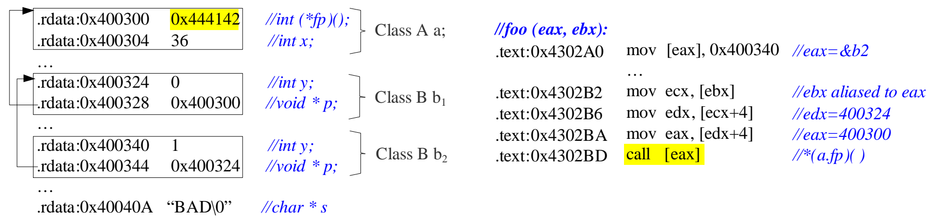

Fig. 4. An example showing indirect call handling in binary stretching/extraction.

A naive solution is to identify and patch any constant value in the binary that appears to be a jump/call target. But this is not safe as such values may not be jump/call targets. Notice in the example, there is a null-terminated string "BAD" at address 0x40040A. With the little endian representation in x86, this string has the same binary value as the function pointer at 0x400300. Without type information, it is impossible to know whether the value is a string or function pointer. Failure to identify and patch a function pointer leads to broken control-flow, changing the semantics of the target binary. Misclassifying a string as a function pointer leads to undesirable changes to data. While it is plausible to leverage recent advances in binary type inference to type constants in a binary [18–21], the involvement of aliasing as in the example makes such analysis very difficult. In fact, IDA-Pro [22] failed to recognize the function pointer for this case.

If a binary has a relocation table and it does not perform any address space layout self-management such as through a packer, the relocation table will provide the positions of all constant values that are jump/call targets for BISTRO to patch them, thus lead to a sound and complete solution to binary stretching/shrinking. However, relocation table may be absent or contain bogus entries in legacy and malware binaries. Hence, for the rest of the paper, *we do not assume the presence of relocation tables in our design and evaluation.* Next, we describe how to handle indirect calls in Section 5.1 and indirect jumps in Section 5.2.

5.1 Handling Indirect Calls

Indirect calls are very common in modern binaries to leverage the flexibility of function pointers. We have discussed the difficulty of handling function pointers at the beginning of Section 5. In fact, there is a more challenging situation, in which a binary may pass its function addresses to external library functions which call back the provided functions (e.g., a user function *cmp()* is provided as a parameter to an external library function

qsort()). In this case, if a function entry has changed due to stretching or shrinking, its invocation sites are outside the body of the binary and thus beyond our control. It is difficult to patch call back function pointer parameters before they are passed on to libraries for two reasons. First, a function pointer might not directly appear as a parameter. It could be a member of a structure passed to an external library. It may even require several layers of pointer indirection to access its value. Patching that is challenging. Second, for many external library functions, we cannot assume the availability of their prototype definitions, it is hence difficult to know their parameter types.

To handle indirect calls including call back functions, we propose to stretch the target binary to make small holes at the entry point of each function that may be an indirect call target. These holes are called *anchors*; they should not be moved during stretching/shrinking. Inside an anchor, we place a jump instruction that jumps to its mapped new address in the stretched/shrunk binary, which is the new entry of the function. As such, we do not need to identify or patch any function pointers in the binary.

Since an anchor must be placed at a fixed address in the stretched binary, it could coincide with instructions that get shifted to the address. To ensure correctness, we put a jump right before an anchor to jump over it. We call the jump the *prefix* of an anchor.

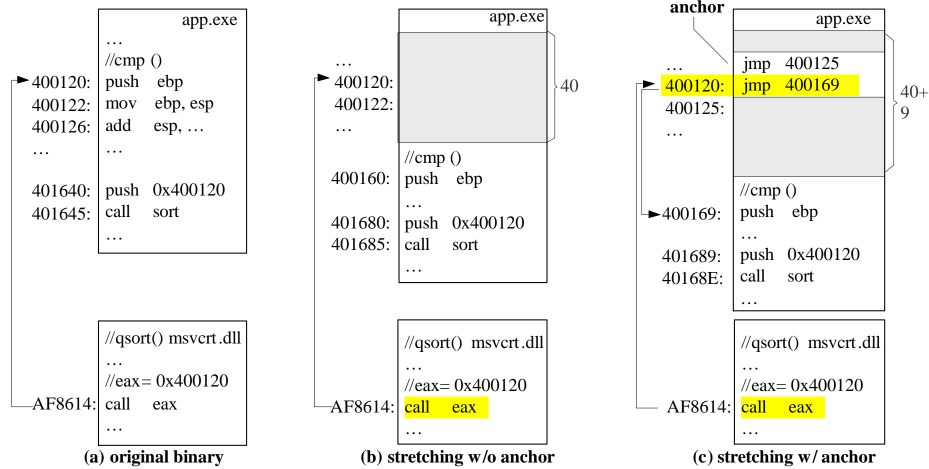

Fig. 5. Stretching with Anchors. The shaded area in (b) is the 40-byte snippet inserted.

Consider the example in Figure 5 (a), in which the call-back function *cmp()* is invoked inside *qsort()*. The entry address of function *cmp()* in the original binary is 0x400120. When we stretch without anchors as shown in (b), in function *qsort()*, the indirect call to *cmp()* at 0xAF8614 will incorrectly go to 0x400120 in the shaded area. When we stretch with anchors as shown in Figure 5(c), an anchor containing the jump instruction will be placed at 0x400120. Any indirect call that goes to the original entry address of *cmp()*, 0x400120, will be redirected to the actual function body at the new entry address. The jump instruction preceding 0x400120 is its prefix.

Anchor-based Algorithm. With the presence of anchors, fixing control flow transfer instructions becomes more challenging than that in Algorithm 1. We hence devise a new algorithm (Algorithm 2). The idea is to divide the stretching/shrinking operation into two phases. In phase one, the subject binary program is stretched/shrunk us-

ing Algorithm 1 to create space for the inserted snippets or removed blocks. Then the stretched/shrunk binary is further stretched to insert anchors using a similar procedure. Separating the two phases substantially simplifies the interference from anchors.

Algorithm 2 Anchor-based stretching algorithm.

Input: P – the subject binary; it has $size$ and $base_addr$ fields to represent its size when loaded into memory and base loading address, respectively.
M – a list of code/data snippets to be inserted/removed, sorted increasingly by their location; each snippet has $addr$, len and $type$ fields, denoting the location, size and type respectively.
A – a list of anchors to be placed, sorted increasingly by their location; each anchor has $addr$ and len fields, denoting the location and the content size, respectively.

Output: $anchor_map$ – the mapping between the indices after placing snippets and their corresponding indices after anchors are placed.
$prefixlen[a]$ – the prefix length of an anchor a.

```
1: function STRECHINGWITHANCHOR(P, M, A)
2:   map ← ComputeMapping(P, M)
3:   P_t ← PatchTarget(P, map)
4:   anchor_map ← ComputeAcMapping(P_t, A)
5:   P' ← PatchTarget(P_t, anchor_map)
6: end function

7: function COMPUTEACMAPPING(P, A)
8:   offset ← 0
9:   ac ← A.begin()
10:  i ← 0
11:  while i < P.size do
12:    curaddr ← P.base_addr + i + offset
13:    if ac.addr == curaddr then
14:      prefix ← i− SIZEOF(JMP)
15:      if P[prefix] is not the start of an instruction then
16:        prefix ← start of instruction before prefix
17:      end if
18:      prefixlen[ac] ← i − prefix
19:      i ← prefix
20:      offset ← offset + ac.len + prefixlen[ac]
21:      ac ← A.next()
22:    else
23:      anchor_map[i] ← i + offset
24:      i ← i + 1
25:    end if
26:  end while
27:  return anchor_map
28: end function
```

Pruning Anchors. Potentially, we can create anchors for all function entries to guarantee that we will never miss any necessary function call forwarding. However, this is not efficient. In fact, we only need to create anchors for the subset of functions that could be the possible target of some indirect call. Assuming a 32-bit machine, we construct the subset with the following criterion: *Any four-byte data value or any four-byte immediate operand in an instruction is considered a possible indirect call target, if it is equal to one of the function entries*. We obtain this subset by sequentially scanning data and code sections. Our pruning heuristic is very effective in practice. For example, the code section size of *gcc* in SPEC CPU 2000 benchmark suite is over 1MB, with over 2000 functions; after pruning, there are only 271 functions left that need anchors.

Embedding a Component with Anchors. If an extracted component contains a function that may be invoked by an indirect call in the component, BISTRO will create an anchor *in the target binary* at exactly the same address of the function entry in *the component's original binary* to allow proper forwarding. If the anchor conflicts with some existing anchor in the target binary, BISTRO will integrate the two overlapping anchors into an arbitration function and redirect control flow to the function instead. The function further determines which real target it should forward the call to. The calls from the target binary and those from the to-be-embedded component are distinguished by setting a flag. The arbitration function uses the flag to decide the real forwarding target.

In some rare cases, the space between two function entries might not be enough to hold the anchors. In such cases, instead of using the jump instruction for redirection, we use a software interrupt instruction, which takes only one byte. When an indirect call

reaches the old function entry, a software exception will be generated and intercepted by our exception handler, which will redirect the control flow to the new function entry.

5.2 Handling Indirect Jumps

Indirect jumps are different from indirect calls as the jump targets may not be function entries, but rather anywhere in the binary. If we adopt the anchor approach, there would be too many anchors needed. One might leverage some heuristics such as that indirect jumps usually receive their targets from jump tables and thus simply patch the jump table entries. However, this is unsafe because of the difficulty of determining jump table boundaries. A jump table may not be distinguishable from regular data. Hence, we propose a different approach. Specifically, we insert a code snippet right before each indirect jump to translate the jump target to its mapped address in the stretched binary at runtime, as shown in the example below.

```
jmp eax  ⟶  mov eax, mapping[eax - old_base]
            add eax, new_base
            jmp eax
```

Note that the example is just for illustration. In our implementation, we use perfect hashing for address lookup, which will be explained later, and preserve the flag register during translation. Since a complete byte-to-byte mapping is computed in Algorithm 1, any indirect jump target could be properly translated and handled by this method. Observe that additional instructions need to be added to perform translation. We can easily handle this by stretching the subject binary to accommodate these instructions.

Branch Target Set Pruning. Although the translation using a complete mapping guarantees safety, it also introduces significant memory overhead. Each byte in the original binary requires 4 bytes to represent its mapped address. In fact, we only need a subset of the mapping: the stretched/shrunk binary will be safe as long as the mapping contains translation for every possible indirect jump target.

We construct the set with the following criterion: *any four-byte data value or any four-byte immediate operand in an instruction is considered a possible indirect jump target, if the value falls in the range of some code section*. We further prune the set by removing the values that point to the middle of an existing instruction. Note that the strategy is safe for long/set jumps as their jump targets are acquired at runtime. This pruning strategy is very effective in practice. For example, the code section size of Adobe Reader X (AcroRd32.exe) is over 800KB, with over 260K instructions; after pruning, there are only 3635 possible branch targets left.

Perfect Hash Translator. The remaining challenge is to achieve fast translation. Note that after pruning, the jump target set becomes a sparse set in the address space. As a compromise between memory consumption and runtime overhead, we choose to use perfect hashing for translation. A perfect hash function maps a set of keys to another set of integer values without any collision. It guarantees $O(1)$ translation time. We use gperf [23] to generate the perfect hash function for the jump target set and compile it into a linkable .obj file that can be embedded in the target binary through BISTRO.

A perfect hash function may require more space than the N keys to achieve $O(1)$ translation time. In practice, we find the size of generated perfect hash functions acceptable. For example, for the 3635 branch targets of Adobe Reader, the generated hash function is about 152KB, which is about 11% of the size of the Adobe Reader binary.

6 Handling Data References

Binary extraction/stretching may cause relocation of data entries, so we need to ensure the correctness of instructions referencing those data. We discuss how to address this problem from the perspectives of the target binary and the component to be embedded.

Compared to the component, the target binary is usually more complex and involves a lot of global data references. To handle this problem efficiently, we group data in the binary as continuous data blocks. If a data block might be indirectly accessed, we will make sure the block is not re-located to avoid patching data accesses, by wrapping the block in an anchor. Note that the number of data access instructions is much larger than the number of indirect jumps/calls. Otherwise, if the data block is only directly accessed, we allow it to be relocated (by Algorithm 1). We use the following criterion: *if the value of any four-byte data, or any four-byte immediate operand (in an instruction) that is not directly used as an address falls in the range of a data block, then this block might be indirectly accessed using data pointers, and hence should not be re-located.*

In contrast, data entries extracted as part of the to-be-embedded component are most likely to be relocated. For example, if they are sparsely distributed in the address space, the BISTRO extractor (Section 2) will collapse them into a contiguous block, causing relocation. We adopt a method similar to the dynamic jump target translation scheme to translate data reference addresses. We add a comparison before translation to avoid translating stack or heap accesses. According to our experience, only 2% of dynamic memory references need to be translated. We further use offline static peephole scanning to identify references that surely access stack and avoid instrumenting them completely.

7 Evaluation

We have implemented BISTRO for Win32 PE binaries as an IDA-Pro plug-in. We have addressed a variety of engineering challenges such as virtual space layout re-arrangement with a large embedded component, patching PE header, import and export tables, and re-generating relocation table. We omit the details due to space limitation.

7.1 Performance: Efficiency and Overhead

We first evaluate the performance of BISTRO by stretching (1) real-world Windows-based applications and (2) SPEC CPU 2000 binaries. Our experiments are done on a Dell Inspiron 15R laptop with Intel(R) Core(TM) i5-2410M 2.30GHz CPU and 4GB memory, running Windows 7 SP1. For the SPEC CPU 2000 benchmark suite, we use the "win32-x86-vc7" config file which includes all integer benchmark binaries and four floating-point benchmark binaries. We compile the benchmark suite using Visual Studio 2010, with full optimizations. To test BISTRO on non-relocatable binaries, we set "/DYNAMICBASE:NO" switch for the compiler to prevent it from generating relocatable binaries. The application binaries are readily available and we do not know about their compilers. Although the binaries of Adobe Reader and Chrome web browser carry relocation tables, we ignore them for testing our solutions for non-relocatable binaries.

We measure the following performance metrics: (1) space overhead – for both binary file and initial memory image – of a stretched binary compared with its original

Table 1. Performance results of stretching Windows software and SPEC CPU 2000 binaries.

Binary	Instr. Count	Indirect Jumps	Indirect Calls	Call/Jump Targets: Anchors(%)	Data Blocks: Data Anchors(%)	File Size (KB)		Initial Mem. Image Size (KB)		Run Time (s)		Stretching Time (s)
						Orig: Stch'ed	growth(%)	Orig: Stch'ed	growth(%)	Orig: Stch'ed	overhead(%)	
SPEC CPU 2000 benchmarks												
164.gzip	19825	19	103	98: 23 (23.47%)	163: 1 (0.61%)	86.5: 98.5	13.87%	424: 440	3.77%	83.2: 84.6	1.68%	0.752
175.vpr	54595	53	106	229: 31 (13.54%)	404: 1 (0.25%)	232: 248.5	7.11%	248: 268	8.06%	64.5: 64.6	0.16%	0.755
176.gcc	337033	456	260	3855: 271 (7.03%)	2580: 14 (0.54%)	1264: 1393	10.21%	1348: 1480	9.79%	33.3: 33.9	1.8%	1.420
181.mcf	20566	36	103	144: 25 (17.36%)	100: 2 (2.00%)	76.5: 85.5	11.76%	100: 108	8%	40.2: 40.4	0.5%	0.685
186.crafty	65375	56	130	312: 29 (9.29%)	247: 1 (0.40%)	283: 298.5	5.48%	1344: 1360	1.19%	38.2: 38.9	1.83%	0.935
197.parser	44554	36	112	155: 27 (17.42%)	463: 1 (0.22%)	164: 173.5	5.79%	352: 360	2.27%	83.1: 83.5	0.48%	0.754
252.eon	114249	50	441	1659: 1253 (75.53%)	1455: 1 (0.07%)	499: 575	15.23%	592: 668	12.84%	42.7: 44.7	4.68%	0.950
253.perlbmk	164093	148	211	2166: 499 (23.04%)	1293: 6 (0.46%)	626: 743	18.69%	648: 764	17.9%	63.3: 67.9	7.27%	1.118
254.gap	129464	35	1357	816: 625 (76.59%)	1142: 1 (0.09%)	452.5: 492	8.73%	896: 936	4.46%	35.4: 37.2	5.08%	1.001
255.vortex	132034	66	145	446: 71 (15.92%)	738: 1 (0.14%)	561: 585	4.28%	588: 612	4.08%	50.6: 51.1	0.99%	1.050
256.bzip2	21360	36	101	145: 25 (17.24%)	150: 1 (0.67%)	87.5: 99	13.14%	172: 184	6.98%	73.4: 74.6	1.63%	0.714
300.twolf	64669	41	106	193: 30 (15.54%)	391: 2 (0.51%)	253: 263	3.95%	296: 304	2.7%	93.2: 93.6	0.43%	0.809
177.mesa	143679	211	552	2675: 473 (17.68%)	942: 5 (0.53%)	549.5: 652.5	18.74%	568: 672	18.31%	64.9: 65.6	1.08%	0.990
179.art	23353	38	103	149: 26 (17.45%)	103: 2 (1.94%)	85.5: 94.5	10.53%	104: 112	7.69%	32: 32.3	0.94%	0.690
183.equake	21824	38	101	146: 27 (18.49%)	116: 1 (0.86%)	88.5: 97	9.6%	104: 112	7.69%	26.1: 26.1	0%	0.720
188.ammp	61214	39	128	224: 70 (31.25%)	279: 1 (0.36%)	235.5: 245.5	4.25%	252: 264	4.76%	88.7: 88.3	1.92%	0.780
Average	-	-	-	- (24.80%)	- (0.60%)	-	10.09%	-	7.53%	-	1.90%	-
Real-world Windows-based Software												
putty	107220	57	662	942: 291 (30.89%)	93: 1 (1.08%)	444: 496	11.71%	472: 524	11.02%	-	-	0.865
gvim	561626	294	5111	3893: 1004 (25.79%)	5081: 22 (0.43%)	1950.5: 2150	10.23%	2008: 2212	10.16%	-	-	2.121
notepad++	272434	159	4302	4897: 2695 (55.03%)	3394: 7 (0.21%)	1584: 1864	17.68%	1660: 1940	16.87%	-	-	1.480
Adobe Reader	273710	146	2543	3635: 2160 (59.42%)	3037: 11 (0.36%)	1445.9: 1702.4	17.74%	1472: 1728	17.39%	-	-	1.556
Chrome	230234	82	1280	1842: 933 (50.65%)	930: 6 (0.65%)	1211: 1338	10.49%	1240: 1368	10.32%	-	-	1.391
Average	-	-	-	- (44.36%)	- (0.55%)	-	13.57%	-	13.15%	-	-	-

version, (2) runtime overhead of the stretched binary, and (3) time for BISTRO to stretch the binary. In particular, we are interested in the overhead incurred *by* BISTRO *itself*, not by the execution of the embedded components. As such, we embed a minimal component (a one-byte snippet) into each subject binary in our experiments. To create a "worst-case" scenario, we insert it at the beginning of each binary so that every byte in the binary gets shifted, which entails *all* indirect control transfer targets in the binary to be redirected. The measured overhead is hence the upper bound of overhead.

For each SPEC 2000 binary, we run both its original and stretched versions, and compare their execution time and file/initial image size. We do not measure the execution time of the Windows applications because they are all interactive. We experience no perceivable overhead when using their stretched versions.

The results are shown in Table 1. From the *Indirect Jumps* and *Indirect Calls* [1] columns, we observe that indirect calls are very common in application binaries, indicating that they might be C++ programs. Further investigation confirms our speculation, indicating BISTRO's effectiveness for binaries compiled from C++ programs. Moreover, there are much less indirect jumps than indirect calls, indicating they are likely to have less impact on runtime overhead. Note that a small number of indirect jumps does not imply an equally small number of potential indirect jump targets. In fact, due to the difficulty of identifying jump table boundaries, we conservatively consider any constant in a binary that appears to be an instruction address as a potential jump target. The large number of potential jump targets and the low impact on performance justify our design choice of using the slightly more expensive but more flexible dynamic target translation scheme (Section 5.2), compared to the anchor scheme (Section 5.1).

The *Call/Jump Targets: Anchors* column shows the number of potential indirect call/jump targets, the number of anchors generated, and their comparison. Observe that the number of anchors created is small, compared to the size of the potential set. For binaries from C++ programs, due to the heavy use of virtual methods, it is not a surprise to see many anchors created. The *Data Blocks: Data Anchors* column shows that only less

[1] We exclude indirect calls to external library functions through import address table (IAT), as these external targets are not handled by our redirection mechanisms.

than 1% of all data blocks need to be preserved at their original locations using anchors. From the *File Size* columns, we can see BISTRO only increases the file size by 10.1% on average for SPEC programs, and 13.6% for application binaries. The overhead is dominated by the perfect hash tables. The *Initial Mem. Image Size* columns show the initial memory consumption when the binary is loaded into memory, which increases by only 7.5% on average for SPEC programs and 13.2% for application programs. Note that BISTRO does not cause any additional memory overhead during execution. The *Run Time* columns present the runtime overhead, which is only 1.9% on average. Except *eon*, *perlbmk* and *gap*, all SPEC binaries have less than 2% overhead. The last column *Stretching Time* shows the stretching time of BISTRO. The time is consistently short, implying that BISTRO can stretch a binary at runtime when it is loaded.

7.2 Case Study I: Binary-level Semantic Patching Using BISTRO

Code reuse is a common practice in software development. One popular approach is to directly compile and statically link a piece of re-usable code with the target software – either directly in the executable or in some private library – to make the software self-contained, avoid compatibility problems, and improve performance. Indeed, developers of many popular programs (e.g., *chrome* and *firefox*) reuse code this way. The consequence is that programs reusing the same code may have the code placed at different locations in their address spaces. The reused code may not even have the same instructions if compiled by different compilers.

Table 2. Results of binary semantic patching using BISTRO

Vulnerability	Patch Extracted From	Vulnerable Application Patched	Original File Size (KB)	Patched File Size (KB) w. / w.o. Reloc	Semantic Patch Available	Vendor Patch Available
CVE-2010-1205	libpng 1.2.43 → 1.2.44 (rpng2-win.exe)	Firefox 3.6.6 (xul.dll)	11747.5	12371.5 / 13005	6/25/2010	7/20/2010
CVE-2011-3026	libpng 1.4.8 → 1.4.9 (rpng2-win.exe)	Zoner Photo Studio 15 (Zxl.dll)	8225.1	8502.1 / 9181.6	2/18/2012	N/A
SA47322 / CVE-2012-0025	IrfanView 4.30 → 4.32 (Fpx.dll)	XnView 1.99.5 (Xfpx.dll)	356	368 / 400	12/20/2011	N/A
		LeadTools 17.5 (ltkdku.dll)	138.5	143 / 151	12/20/2011	N/A
SA47388	XnView 1.98.5 → 1.98.8 (Xfpx.dll)	IrfanView 4.35 (Fpx.dll)	432	448 / 508	3/12/2012	N/A
		LeadTools 17.5 (ltkdku.dll)	372.5	428.5 / 493.5	3/12/2012	N/A
SA48772 / CVE-2012-0278	IrfanView 4.33 → 4.34 (Fpx.dll)	XnView 1.99.5 (Xfpx.dll)	356	368 / 400	4/13/2012	N/A
		LeadTools 17.5 (ltkdku.dll)	138.5	142.5 / 150.5	4/13/2012	N/A
SA49091	XnView 1.98.8 → 1.99 (Xfpx.dll)	LeadTools 17.5 (ltkdku.dll)	372.5	428.5 / 488.5	6/15/2012	N/A

However, code reuse via static linking introduces a security liability: When a piece of re-usable code contains a vulnerability, all programs that reuse the code will suffer from the same vulnerability. If these programs have been shipped in binary forms, the only way to fix the vulnerability is to release multiple binary patches – one for each program and by the corresponding vendor. However, not all vendors react to a vulnerability with equal timeliness and some may not even be aware of the vulnerability not in their own code. Thus it may be desirable for customers, who do *not* have source code access, to patch these programs without vendors' involvement. Binary *syntactic patching*, which directly applies a patch for software A to software B sharing the same (vulnerable) code, will hardly work, because of the different locations of the code and the syntactic differences between the two code copies (due to different compilers used or different call/jump targets inside the copies).

In our first case study, we show that BISTRO can enable *binary semantic patching*. Assume that software A and B share a function f and the vendor of A has released a binary patch of f for a vulnerability. Let the patched program and the patched function be A' and f', respectively. We will use BISTRO to extract f' from A' and embed it to B

to replace the vulnerable version. Note that BISTRO is critical in ensuring the extracted f' is properly patched and the target binary B is properly stretched to contain f'.

We acquire a group of application binaries that leverage the same vulnerable component using public, vendor-provided information (e.g., which libraries are used in the software) or by finding similar binary snippets using the binary comparison tool *bindiff* [32]. Suppose at least one binary in the group, say A, has a patched version A'. Our goal is to extract a *semantic* patch out of A' and transplant it to patch the other vulnerable binaries $\{B_1, ..., B_n\}$.

We collect 6 real-world vulnerabilities, with their CVE or Secunia IDs shown in Column 1 of Table 2. For each vulnerability, the vulnerable program(s) that has been patched by its vendor is shown in Column 2. The file names in braces represent the files that are patched. Column 3 shows a list of other un-patched programs with the same vulnerabilities. Column 6 shows the patch release date for the application in Column 2, i.e. the earliest date we can extract the semantic patch. Column 7 shows the date when the vendors for the software in Column 3 release their patches (N/A means no vendor patch is available yet). Most of the applications used in this case study are close-source (except *libpng* and *firefox*). Observe that most of the applications in Column 3 do not have vendor patches so far. For *firefox*, the new version (3.6.7) which patched the vulnerability was released – but with a one-month latency. With BISTRO, we can fix all these vulnerable applications as soon as one vendor releases the corresponding patch.

Failure of Syntactic Patching. We first verify that simple syntactic patching does not work – that is, using an existing binary differencing tool that generates and applies patches (e.g., *xdelta*, *bsdiff*, *bspatch*, etc.) will not properly patch $B_{1...n}$. For each vulnerability in Table 2, we use *bsdiff* to extract the syntactic difference between the pair of shared functions (f and f') in the versions in Column 2 as a patch, and use *bspatch* to apply it to the corresponding vulnerable applications in Column 3. None of the resultant binaries works. Further inspection shows that syntactic patches cannot properly fix the call/jump targets that are different among copies of the same reused code.

Function Identification. To extract the semantic patch for a specific vulnerability, we need to identify the functions in A and A' that are related to the vulnerability. To illustrate, we denote the set of functions in a binary A by F_A. First, we notice that the related functions must exist in all the vulnerable binaries. We take A and the other vulnerable binaries $\{B_1, ..., B_n\}$ and use *bindiff* [32] to identify the set of common functions F among them (Equation (1) below). However, some functions in F are not related to the vulnerability (e.g., other pieces of reused code.) To pinpoint the relevant functions in F, we leverage the observation that they have been patched in A'. Particularly, we utilize the partial matching feature of *bindiff* to identify the relevant patched functions as shown in Equation (2). The generated M is a mapping that maps a function $f \in F$ to its patched version $f' \in F_{A'}$. Two functions are said to be partially matched when they share similar characteristics (e.g., common basic blocks, similar CFG) but are not exactly the same. By performing partial matching between F and $F_{A'}$, we also exclude patches in A' that are not related to the target vulnerability. Note that a vendor may

patch a few (unrelated) problems in a single release.

$$F = F_A \cap F_{B_1} \cap F_{B_2} ... \cap F_{B_n} \quad (1)$$

$$M = \mathit{BinDiff_Partial_Matching}(F, F_{A'}) \quad (2)$$

Patch Transplanting. We have developed a binary semantic patching tool based on BISTRO and *bindiff*. The extraction and application of the patch is guided by mapping function M. For each mapping under M: $(f, f' \in F_{A'})$, we use BISTRO to extract f' from A' as the semantic patch for f. For each vulnerable binary B, we use *bindiff* to find f. We use BISTRO to cut out f and then stretch the resulting binary to implant f' at the same starting address of f. BISTRO ensures the correctness of both f' and the patched binary B' by properly stretching and patching control transfer instructions and data references. Our patching tool tries to avoid extracting dependent functions or global data entries of f' (i.e., functions being called and global data accessed by f') as much as possible by redirecting them to their counterparts in the target binary B. Since f' is a patched version of f, they likely share the same dependencies. For example, for each function invocation to function g' inside f', if *bindiff* is able to identify the matching function g in B, our tool will automatically redirect the invocation in the extracted patch to g, without extracting g'. To be conservative, g and g' must be fully matched. Otherwise, g' will be extracted as part of the semantic patch.

We evaluate our patching tool on the subjects in Table 2. We apply our tool in two different ways to stress-test the robustness of BISTRO: first, we use the relocation information when it is present in the binary; second, we do not use relocation information at all. The patches are not large, each consisting of tens to hundreds of instructions. However, it is not straightforward to generate them because of the nature of the vulnerabilities being patched. In both runs, the patching is successful: the patched applications work well and no longer suffer from the corresponding vulnerabilities. Columns 4 and 5 show the file size changes.

The first two vulnerabilities are in *libpng*, which is widely used in various software to read, write and render PNG images. The two vulnerable applications in Column 3 have *libpng* statically linked in their private DLLs (*xul.dll* and *Zxl.dll*). To patch these DLLs, we extract the semantic patch from *rpng2-win.exe*, a sample application in the *libpng* package. The remaining four vulnerabilities lie in *libfpx*, a library to handle the Flashpix (.fpx) image format. For the four vulnerabilities, only the first one was patched by the maintainer of *libfpx*; the other three were patched by individual developers who use *libfpx*. However, as shown in the table, individual developers only care about patching the *libfpx* code in their *own* applications. Using our binary semantic patching tool, users of the un-patched applications can transplant the patches and eliminate the vulnerabilities without the help of application developers.

7.3 Case Study II: Malware Stitching Using BISTRO

In the second case study, we demonstrate how BISTRO helps in the study of cyber attacks and counter-attacks. Specifically, we use BISTRO to compose a new, *executable* malware by stitching 3 separate functional components extracted from a *non-executable*

sample of the Conficker worm [15]. It is an *unpacked* version without relocation information. Based on the published technical report of Conficker [15] and manual code inspection, we identify the code and data associated with the following 3 components:

– **DNS API hijacking.** This component prevent DNS query of the web sites in a blacklist by hijacking the functions *Query_Main*, *DNSQuery_A*, *DNSQuery_W* and *DNSQuery_UTF8* in *dnsapi.dll*. The result is these web sites will not be accessible using their domain names.
– **Code injection.** To hijack the functions in *dnsapi.dll* used by a process (e.g., Internet Explorer), the malware must inject itself into the address space of the process. This component performs the injection. It takes the process identifier (PID) of the target process and the path of the malware as parameters.
– **Process identification.** This component gets a process' PID using its process name and provide the PID to the code injection component.

The identification process takes us about 60 minutes. After that we use BISTRO to extract the three components from the Conficker sample. We then create a dummy DLL to serve as the container of these components. Next, we use BISTRO to embed the 3 components into the empty DLL, right before the *DllMain()* function. After that, we add instructions to the *DllMain()* function to invoke the inserted components. The invocation code first checks if the current process is the target process. If so, it will invoke the DNS API hijacking component to hijack the DNS query. If not, it will call the process identification component to find the PID of the target process, and then call the DLL injection component to inject itself into the target process for DNS API hijacking. The whole composition process takes us about 30 minutes.

To verify the functionality of the newly composed malware, we select two applications as our targets (in two experiment runs): Internet Explorer and FlashFXP (an FTP client). After being loaded, the malware injects itself into the target processes. Then, in the target application, we try to access web site *avast.com*, which is blacklisted by Conficker [15]. Interestingly, the access was not blocked at first (namely, the malware did not succeed). After debugging, we found that it was due to a bug in Conficker's original code: the hijacked *DNSQuery_W()* has one unnecessary instruction which sets a wrong return value. We point out that *we would not have spotted the problem, had we not made these components executable and observed their runtime behavior.* After removing this instruction using BISTRO, both IE and FlashFXP are successfully compromised: they can no longer access *avast.com* due to a DNS query error.

7.4 Case Study III: Trojan-ing Kernel Drivers

In the third case study, we demonstrate the use of BISTRO in transplanting malicious modules from existing kernel rootkits to existing kernel-level device drivers. The trojaned kernel drivers will execute the rootkit modules while performing their original functionalities. The goal of this case study is two-fold: (1) to evaluate the effectiveness of BISTRO for kernel-level binaries, (2) to show the possibility and ease of composing – instead of implementing from scratch – device drivers with hidden and possibly malicious logic. Such trojan-ed device drivers are more difficult to detect and clean up, compared with traditional rootkits that come as stand-alone kernel modules. On the flip

Table 3. Trojan-ed Device Drivers (Two per Row).

Original Driver	File Name	File Size(KB)	w/ proc_hider embedded File size(KB)	w/ proc_hider embedded Work?	w/ keylogger embedded File size(KB)	w/ keylogger embedded Work?
Beep	beep.sys	4.1	7.9	√	12.4	√
FAT File System	ftdisk.sys	122.1	135.1	√	137.5	√
NT File System	ntfs.sys	561.1	595.3	√	598	√
Intel E1000 Network Adapter	e1000325.sys	167.1	175.4	√	180.6	√
Logitech C500 Webcam	LVPr2Mon.sys	25	31.4	√	33.8	√

side, trojan-ed kernel drivers can also be leveraged in defensive missions, such as honeypot deployment, to achieve better stealthiness in attack monitoring and containment. For example, malware may try to aggressively detect and disable any monitoring kernel module (e.g., Sebek). With BISTRO, one could transplant stealthy monitoring/logging functions into a general-purpose device driver, making them more difficult to detect and disable.

In this case study, the two Windows-based kernel rootkits tested are captured variants of *HookSSDTMDL* and *Klog* which were originally packed in the wild, without relocation information for the rootkit code. The packers use their own algorithms to perform rootkit code relocation, and such relocation information is lost after the rootkits are unpacked. The samples we obtained are the *unpacked* version. We wish to extract two modules from the samples (one from each): (1) *proc_hider* for hiding processes and (2) *keylogger* for logging keystrokes. To show the generality of device driver trojan-ing, we transplant these two rootkit modules into 5 different Windows-based kernel drivers, resulting in a total of 10 trojan-ed kernel drivers.

First, we use an approach similar to [16, 24] to identify the modules to extract. We then use BISTRO to shrink each of the two kernel rootkits so that only the code of the two modules and the data they access remain (as snippets). The size of the extracted snippets is 2.3KB for *proc_hider* and 7KB for *keylogger*, respectively. The size of the data in the extracted snippets is 169 bytes and 514 bytes, respectively. After preparing the snippets, we use BISTRO to insert each of them into each of the following five drivers: *beep.sys, ftdisk.sys, ntfs.sys, e1000325.sys* and *LVPr2Mon.sys*. The OS is Windows XP (SP2)[2]. Table 3 lists the 10 resultant trojan-ed drivers (two per row). For each of the 10 drivers, we install it and confirm the proper working of (1) the original driver functionalities and (2) the malicious rootkit module.

When determining where to insert a rootkit module, we choose to insert it right before a randomly chosen function in the driver. To invoke the rootkit module when the driver is loaded, we insert a call to the rootkit module in the `DriverEntry ()` function of the driver, which is a mandatory function exported by any driver and can be located in the code section by reading the export table. Interestingly, we are able to use BISTRO to implement a "timebomb"-style invocation of the rootkit module: Instead of activating the module upon driver loading, we wish to invoke it only under a certain condition. Specifically, when we trojan the NT file system driver (`ntfs.sys`) with the keylogger module, we want to activate the keylogger only when a file with the word

[2] We use Windows XP because the real-world rootkits we obtained do not work with newer versions.

"secret" in its name is opened. This is done by calling a file-name-matching function before activating the keylogger. We write this function using C, compile it into a binary snippet, and use BISTRO to insert it into `ntfs.sys` (just like inserting the rootkit module), particularly, inside `NtfsFsdCreate ()`, a function that is called every time a file is opened. Here, we leverage IDA-Pro to spot this function, which is the IRP dispatch routine to handle `IRP_MJ_CREATE` IRPs. This can be easily done by finding the initialization of the IRP dispatch table in `DriverEntry ()`. We verify that the timebomb-controlled trojan-ed driver works as expected.

We observe that, for a "native" driver developed by the OS vendor (e.g., *beep.sys, ftdisk.sys* and *ntfs.sys*), the installation of the trojan-ed version does raise an alert to the user, thanks to the built-in integrity check mechanism in the OS. Unfortunately, if the user chooses to ignore the alert, the installation will proceed and the system will never complain again. For third-party drivers (e.g., *e1000325.sys* and *LVPr2Mon.sys*), the detection of maliciously trojan-ed version is much more difficult because these drivers may be widely distributed and frequently updated without a centralized authority. Even if such an authority exists and performs digital signing for its drivers, authors of trojan-ed drivers may still evade detection by stealing certificates from the authority to sign their trojan-ed drivers, as was done in the crafting of Stuxnet [33]. In our study, the installations of the trojan-ed third-party drivers did not trigger any warnings.

8 Discussion

BISTRO cannot work on self-modifying, self-checking or obfuscated binaries. Self-modifying binaries generate instructions dynamically during runtime, which could not be statically patched using BISTRO. Self-checking binaries use checksum or other integrity checks to detect changes made to their code by BISTRO, thus may refuse to run properly. Obfuscated binaries in many cases cannot be properly disassembled. For instance, the attacker can craft a conditional jump, with one branch never taken but pointing to a data entry. A disassembler will have trouble handling such binaries as it does not know statically that one of the branches cannot be taken. However, we note that all other static binary rewriting/instrumentation techniques face the same challenge.

Our anchor and branch target set pruning criteria assume the constants in a binary represent a superset of all possible indirect control transfer targets. This assumption should hold for binaries generated by common compilers. One exception is position independent code (PIC), which obtains addresses at runtime and use them to compute indirect control transfer targets. All PIC we encountered has the form of making a call and then obtain the return address from the stack (e.g., *call $+5; pop eax*), which is the address of the instruction right after the call. We identify all such instructions and insert snippets to adjust the addresses to their mapped addresses. Also, special compilers or hand-written binaries might violate our assumption. For example, in the instruction sequence *mov eax, Target; add eax, 5; jmp eax*, the actual target is *Target* $+5$ instead of the constant *Target*; our pruning heuristic will miss the actual target. For such binaries, we can choose not to prune the anchor set or the branch target set, which will consume more memory but guarantee correctness.

Currently, BISTRO only supports Win32 PE binaries. However, the design is general, without relying on specific features of Win32 PE. We plan to extend BISTRO to support other formats on x86, especially ELF, which is similar to Win32 PE.

9 Related Work

The most related work is discussed in Section 1 (with details in Section 3.) In this section, we discuss other related work in the general area of binary manipulation. They fall into three categories: (1) static binary rewriting (2) dynamic binary rewriting and (3) binary component identification, extraction and reuse.

Static Binary Rewriting. Static binary rewriting is widely applied in many scenarios, such as in-lined reference monitors [34], software fault isolation [25, 26, 7, 27], binary instrumentation [11, 10, 12, 6, 8, 5], binary obfuscation [37, 38] and retrofitting security in legacy binaries [28, 13]. Most of these rewriters require the binary to be compiled by specific compilers, or contains symbolic information.

PEBIL [12], REINS [34], STIR [14] and SecondWrite [13] are recently developed rewriters targeting stripped binaries. However, they all aim at rewriting a single binary, so they all keep the original code and data sections in place. In contrast, BISTRO supports "transplanting" binary components from one or more binaries to a target binary, which requires rewriting and combining multiple binaries. Keeping original code and data sections in place may result in address space conflicts and hence is not an option for BISTRO. Detour-based techniques [11, 9, 10] are lightweight and can work on stripped binaries. However, they cannot patch non-trivial jumps/calls that are repositioned.

Dynamic Binary Rewriting. Dynamic binary rewriters [3, 4, 1, 29] are generally more robust as they do not require specific compilers or symbolic information. It is possible to apply them to conduct binary stretching and transplanting. However, we choose to use a static approach mainly because of the following two reasons: (1) Dynamic binary rewriters usually have much higher run time overhead than static ones. (2) It is more difficult to deploy a instrumented binary using dynamic approaches, as the rewriter itself must be deployed along with the binary.

Binary Component Identification, Extraction and Reuse. Recently, researchers proposed to identify, extract and reuse components from binaries for security scenarios [30, 17, 24]. Kolbitsch et al. proposed Inspector Gadget [30], which performs dynamic slicing on a malware binary to identify and extract the slice pertinent to a specific malicious functionality, and wrap the slice into a stand-alone binary that could be reused later to execute the malicious functionality. Inspector Gadget is able to extract component from self-modifying code, which is not supported by BISTRO due to the limitation of static binary manipulation. Using dynamic slicing, Inspector Gadget also avoids the problem of handling indirect calls/jumps in BISTRO as all call/jump targets are directly known in the slice. However, the slice may not cover all possible code paths, which could result in incorrect execution when the user provides an input that would lead to a code path which is not included in the slice. Compared to Inspector Gadget, BISTRO statically extracts the component from the binary, which involves handling of indirect calls/jumps but provides better code path coverage.

Caballero et al. proposed BCR [17] to identify and extract a function from a binary using a combination of static and dynamic analysis. The extracted function, in the format of disassembly, is wrapped in a C file to be reused. BCR statically disassembles the designated function starting at its entry point; when encountering indirect call/jumps, BCR utilizes dynamic execution trace to the find the call/jump targets. During the extraction, BCR rewrites all calls/jumps to use labels. Using labels implies that indirect call/jump can only have one target, which may not always hold in practice. Although BCR specially handles indirect jumps that use jump tables, there are other forms of multiple-target indirect calls/jumps such as function pointers and vtables. Compared to BCR, BISTRO preserves the original semantic of indirect calls/jumps when performing the extraction, hence does not suffer from this problem.

Neither Inspector Gadget nor BCR could extract components from non-executable binaries (as in Section 7.3) because they are based on dynamic analysis. In such case, BISTRO can still perform the extraction statically. Moreover, neither Inspector Gadget nor BCR supports reusing extracted components to enhance legacy binaries (as in Section 7.2), as they lack the capability of embedding instructions that invoke the components into the target binary. BISTRO is able to handle such a scenario by performing both binary component extraction and embedding.

Lin et al. proposed ROC [24] which uses dynamic slicing to identify reusable functional components in a binary for attack purposes. However, compared to BISTRO, ROC only reuses the components in the same binary; it does not support extraction or reusing components in a different program.

10 Conclusions

We have developed a new pair of binary program manipulation primitives called BISTRO, which provides two complementary capabilities: extracting and re-packaging a functional component from a binary program; and embedding a functional component in a target binary program. We have overcome the challenges of patching control transfer instructions and data references to preserve the semantics of both the extracted component and the stretched binary program, especially indirect calls and jumps. BISTRO incurs low runtime overhead (1.9% on average) and small space overhead (11% on average). The extraction and embedding operations are highly efficient, with less than 1.5s for most cases. We have applied BISTRO to two security application scenarios, demonstrating BISTRO's efficiency, precision, and versatility.

Acknowledgements. This research has been supported by DARPA under Contract 12011593. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of DARPA.

References

1. C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *SIGPLAN Notices*, 2005.

2. N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” in *PLDI*’07.
3. D. Bruening, “Efficient, transparent, and comprehensive runtime code manipulation,” Ph.D. dissertation, MIT, 2004.
4. F. Bellard, “Qemu, a fast and portable dynamic translator,” in *USENIX ATC*’05 .
5. B. De Sutter, B. De Bus, and K. De Bosschere, “Link-time binary rewriting techniques for program compaction,” in *TOPLAS*’05.
6. R. Muth, S. Debray, S. Watterson, and K. De Bosschere, “Alto: a link-time optimizer for the compaq alpha,” in *SPE*’01.
7. M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-flow integrity principles, implementations, and applications,” in *TISSEC*’09.
8. A. Eustace and A. Srivastava, “Atom: A flexible interface for building high performance program analysis tools,” in *USENIX ATC*’95
9. B. Buck and J. K. Hollingsworth, “An api for runtime code patching,” in *IJHPCA*’00.
10. T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen, “Instrumentation and optimization of win32/intel executables using etch,” in *USENIX Windows NT Workshop*, 1997.
11. G. Hunt and D. Brubacher, “Detours: Binary interception of win32 functions,” in *USENIX Windows NT Symposium*’99.
12. M. Laurenzano, M. Tikir, L. Carrington, and A. Snavely, “Pebil: Efficient static binary instrumentation for linux,” in *ISPASS*’10.
13. P. OSullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. Keromytis, “Retrofitting security in cots software with binary rewriting,” *IFIP SEC*’11.
14. R. Wartell, V. Mohan, K. Hamlen, and Z. Lin, “Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code,” in *CCS*’12.
15. P. Porras, H. Saidi, and V. Yegneswaran, “Conficker c analysis,” *SRI International*, 2009.
16. N. Johnson, J. Caballero, K. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song, “Differential slicing: Identifying causal execution differences for security applications,” in *IEEE S&P*’11.
17. J. Caballero, N. Johnson, S. Mccamant, and D. Song, “Binary code extraction and interface identification for security applications,” in *NDSS*’10.
18. G. Balakrishnan and T. Reps, “Analyzing memory accesses in x86 executables,” in *CC*’04.
19. A. Slowinska, T. Stancescu, and H. Bos, “Howard: A dynamic excavator for reverse engineering data structures,” in *NDSS*’11.
20. J. Lee, T. Avgerinos, and D. Brumley, “Tie: Principled reverse engineering of types in binary programs,” in *NDSS*’11.
21. Z. Lin, X. Zhang, and D. Xu, “Automatic reverse engineering of data structures from binary execution,” in *NDSS*’10.
22. Hex-Rays, “Ida pro disassembler.” http://www.hex-rays.com/products/ida/index.shtml
23. D. Schmidt, “Gperf: a perfect hash function generator,” *More C++ gems*, 2000.
24. Z. Lin, X. Zhang, and D. Xu, “Reuse-oriented camouflaging trojan: Vulnerability detection and attack construction,” in *DSN*’10.
25. R. Wahbe, S. Lucco, T. Anderson, and S. Graham, “Efficient software-based fault isolation,” in *OS Review*’94.
26. S. McCamant and G. Morrisett, “Evaluating sfi for a cisc architecture,” in *USENIX Security*’06.
27. Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula, “Xfi: Software guards for system address spaces,” in *OSDI*’06.
28. M. Prasad and T. Chiueh, “A binary rewriting defense against stack based buffer overflow attacks,” in *USENIX ATC*’03.

29. K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. Soffa, “Retargetable and reconfigurable software dynamic translation,” in *CGO*’03.
30. C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda, “Inspector gadget: Automated extraction of proprietary gadgets from malware binaries,” in *IEEE S&P*’10.
31. A. Srivastava, A. Edwards, and H. Vo, “Vulcan: Binary transformation in a distributed environment,” *Tech. Rep.*, Microsoft Research, 2001.
32. H. Flake, “Structural comparison of executable objects,” in *DIMVA*’04.
33. N. Falliere, L. Murchu, and E. Chien, “W32. stuxnet dossier,” *White paper, Symantec Corp., Security Response*, 2011
34. R. Wartell, V. Mohan, K. Hamlen, and Z. Lin, “Securing untrusted code via compiler-agnostic binary rewriting,” in *ACSAC*’12.
35. A. Dinaburg, P. Royal, M. Sharif, and W. Lee, “Ether: malware analysis via hardware virtualization extensions,” in *CCS*’08.
36. S. Nanda, W. Li, L. Lam, and T. Chiueh, “BIRD: binary interpretation using runtime disassembly,” in *CGO*’06.
37. A. Moser, C. Kruegel, and E. Kirda, “Limits of static analysis for malware detection,” in *ACSAC*’07.
38. I. Popov, S. Debray, and G. Andrews, “Binary obfuscation using signals,” in *USENIX Security*’07.