Binary Rewriting without Control Flow Recovery[*]

Gregory J. Duck
Department of Computer Science
National University of Singapore
Singapore
gregory@comp.nus.edu.sg

Xiang Gao
Department of Computer Science
National University of Singapore
Singapore
gaoxiang@comp.nus.edu.sg

Abhik Roychoudhury
Department of Computer Science
National University of Singapore
Singapore
abhik@comp.nus.edu.sg

Abstract

Static binary rewriting has many important applications in software security and systems, such as hardening, repair, patching, instrumentation, and debugging. While many different static binary rewriting tools have been proposed, most rely on recovering control flow information from the input binary. The recovery step is necessary since the rewriting process may move instructions, meaning that the set of jump targets in the rewritten binary needs to be adjusted accordingly. Since the static recovery of control flow information is a hard problem in general, most tools rely on a set of simplifying heuristics or assumptions, such as specific compilers, specific source languages, or binary file meta information. However, the reliance on assumptions or heuristics tends to scale poorly in practice, and most state-of-the-art static binary rewriting tools cannot handle very large/complex programs such as web browsers.

In this paper we present E9Patch, a tool that can statically rewrite `x86_64` binaries without any knowledge of control flow information. To do so, E9Patch develops a suite of binary rewriting methodologies—such as instruction punning, padding, and eviction—that can insert jumps to trampolines without the need to move other instructions. Since our approach preserves the set of jump targets, the need for control flow recovery and related heuristics is eliminated. As such, E9Patch is robust by design, and can scale to very large (>100*MB*) stripped binaries including the Google Chrome and FireFox web browsers. We also evaluate the effectiveness of E9Patch against realistic applications such as binary instrumentation, hardening and repair.

[*]This work was partially supported by the National Satellite of Excellence in Trustworthy Software Systems, funded by National Research Foundation (NRF) Singapore under National Cybersecurity R&D (NCR) programme.

CCS Concepts: • **Software and its engineering**;

Keywords: Static binary rewriting, binary instrumentation, binary patching, binary repair, instruction punning, instruction eviction, memory management

ACM Reference Format:
Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary Rewriting without Control Flow Recovery. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3385412.3385972

1 Introduction

Static binary rewriting has many important applications in software security and systems, such as program hardening [8, 19, 41, 43], automated repair [20, 33], instrumentation [10, 29], optimization [14, 36], and debugging [5, 31]. The advantage of binary rewriting is that it can be applied even when the source code of the software is unavailable, as is often the case with *Commercial Off-The-Self* (COTS) software. The importance and usefulness of static binary rewriting has led to the development of multiple tools spanning many years [2, 6, 9, 10, 21, 25, 27, 28, 30, 32, 34, 35, 37–39, 41, 42]. Most existing tools use a pipeline consisting of (1) a disassembler *frontend* that parses machine code instructions from the input binary, (2) the *recovery* of (some form of) control flow information such as jump targets, etc., (3) a *transformation* that inserts, deletes, replaces, or relocates binary code, and (4) a *backend* that emits the modified binary file. Since the binary rewriting process may move instructions, some form of control flow recovery is necessary in order to adjust the set of jump targets in the rewritten binary.

However, recovering control flow information from binary code is notoriously difficult [26]. One approach is to exploit binary file meta information such as debug symbols or relocations. However, such information is not always available (e.g., *stripped binaries* or non-PIC). Another approach is to use static binary analysis for control flow recovery. However, this is undecidable in the general case [15]. To compensate, most analysis-based rewriting tools make simplifying assumptions about the input binary code, such as assuming that indirect jumps follow a specific pattern (e.g., *jump tables* for C-style `switch` statements), etc. However, this tends to scale poorly, as the underlying heuristics/assumptions will break for large enough binaries. For example, a "99%

accurate" static analysis will have an effective accuracy of ~37% for binaries 100× larger, and a near zero accuracy for binaries 1000× larger. Such scales do exist in practice. For example, the Google Chrome [17] binary is approximately 1200× larger than `ls` binary from `coreutils`. The reliance on assumptions/heuristics is recognised as a major problem for binary rewriting systems [2].

In this paper we introduce E9Patch, a tool for static binary rewriting without the need for control flow recovery and associated assumptions/heuristics. The key idea behind E9Patch is to exclusively use binary rewriting methodologies that are *control flow agnostic*, meaning that the set of jump targets from the input binary need not be known and will be preserved. For example, one such promising methodology is baseline *instruction punning*—an idea previously used to implement dynamic instrumentation [7]. Here, given a set of *patch location instructions* P, instruction punning attempts to substitute each $I \in P$ with a *jump instruction* J that redirects control flow to a *trampoline* that implements some intended binary patch/instrumentation before returning control flow back to the main program. However, some instructions are smaller than jumps (five bytes for the `x86_64`) and cannot be substituted directly. To handle this case, baseline instruction punning will specially engineer a "punned" jump whose byte representation is the same as that of any overlapping instruction. This "punned" jump can therefore safely substitute I without modifying or moving any other instruction. Crucially, the set of jump targets is also preserved, meaning that instruction punning is control flow agnostic.

Although promising, the applicability of baseline instruction punning is highly dependent on the byte values of overlapping instructions. The resulting punned jump will sometimes target an invalid memory location that cannot be used. This may result in poor *coverage* where only a subset of P can be patched. As such, boosting patching coverage is one of the key technical challenges for E9Patch. To do so, we develop a suite of patching "tactics" that can be applied to cases where instruction punning fails. For example, one key idea is *instruction eviction*, which changes the byte representation of overlapping instructions without changing the execution semantics. This may allow instruction punning to find new valid punned jumps where previously none were available. We show that our tactics can boost patching coverage to at or near 100% for realistic applications.

Another problem with instruction punning is that suitable trampoline locations are typically *constrained*. This means that trampoline memory cannot necessarily be packed contiguously, possibly leading to high fragmentation and output file size bloat. To address the issue, we introduce a new space optimization—*physical page grouping*— that can significantly reduce physical memory usage (RAM, file size), sometimes by orders of magnitude. Furthermore, physical page grouping uses file-backed mappings for executable code, allowing for physical memory resources to be shared by several instances of the same program.

In summary, the main contributions of this paper are:

- We adapt baseline instruction punning to a static binary rewriting setting. However, instruction punning by itself does not provide sufficient coverage for most applications. For this, we develop several new instruction patching tactics, such as instruction *padding* and *eviction*, that are designed to boost coverage to at or near 100%.
- We present an optimization in the form of *physical page grouping*—a method for reducing physical memory usage while preserving file-backed executable code.
- We present E9Patch, a powerful static binary rewriting tool designed to scale to very large binaries. To do so, E9Patch only uses binary rewriting methodologies that preserve the set of jump targets, thereby eliminating the need for control flow recovery and associated heuristics.
- We evaluate E9Patch against the SPEC2006 benchmark suite [18] and several large binaries. To demonstrate scalability, we also evaluate E9Patch against web browsers such as Google Chrome [17] and FireFox [16], each with a binary size exceeding 100*MB*. We also consider two realistic applications in the form of binary repair and binary heap write hardening.

Open Source Release

https://github.com/GJDuck/e9patch

2 Overview and Background

Our aim is to statically rewrite (or "patch") large binaries (executables and libraries) while preserving correctness and reasonable performance. Although many static binary rewriting tools exist [40], many work by relocating code and updating the control flow (e.g., jump targets) in the modified binary—an approach that scales poorly [2]. Instead, our approach is to design static binary rewriting methodologies that are control flow agnostic, meaning that the set of jump targets need not be known in order to correctly rewrite the binary. The key idea is to treat all instructions (I) as potential jump targets (whether they really are or not), and to preserve the program semantics should control flow happen to jump to I at runtime. To achieve this, we use a minimally-invasive design that ensures all instructions are either:

1. preserved;
2. replaced by an operationally equivalent instruction; or
3. replaced by an instruction that implements some desired modification (e.g., repair, instrumentation, etc.).

We modify binaries strictly at the instruction level—i.e., a patch operation may replace/substitute individual instructions, but must not move nor change the semantics of other (non-patched) instructions. Our approach must also reasonably balance performance, coverage and scalability.

2.1 Background

We briefly review existing x86_64 instruction patching methods (*B0*/*B1*/*B2*) that are also control flow agnostic.

2.1.1 Baseline *B0*: Signal Handlers.

One old idea is to replace each patch location instruction with a single-byte x86_64 int3 instruction. When executed, the int3 instruction raises an interrupt which manifests as a SIGTRAP signal that is sent to the program. Next, a signal handler implements the patch. This approach is traditionally used by debuggers to implement *break points*. Although jump targets are preserved, the use of interrupts and signal handlers requires kernel/user mode context switching, and suffers from poor performance (sometimes by orders of magnitude).

2.1.2 Baseline *B1*: Jumps.

Another old idea is to replace each patch location instruction with a *jump instruction* that redirects control flow to a *trampoline* that implements the patch. The patch trampoline can also execute (or emulate) the displaced instruction (if necessary) before returning control back to the main program. This approach is much faster than signal handlers, and is used by many different binary rewriting tools.

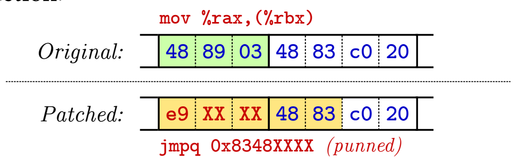
For the x86_64, this approach can be implemented using the *relative near jump* (jmpq *rel32*) instruction. Here *rel32* is a 32bit signed integer that is added to the *program counter* (%rip) in order to implement the jump. The relative near jump instruction is five bytes long, including one byte for the *opcode* (0xe9) and four bytes for the *rel32* value. A patch location instruction that is greater-than-or-equal-to five bytes can be directly replaced, but complications arise when the patch location instruction is smaller than five bytes. One idea is to replace *more than one* instruction with a jump. However, this assumes that the successor instructions are themselves not jump targets, meaning that some control flow information must be known. Since this violates our design requirement of control flow agnosticism, the generalized approach cannot be used.

2.1.3 Baseline *B2*: Instruction Punning.

Another idea is to specially engineer jumps that can safely overlap with other instructions. This is known as *instruction punning*—an approach previously used by LiteInst [7] for dynamic instrumentation. The basic idea is to find a relative offset value (*rel32*) that *shares the same byte representation as any overlapping instruction*. The patch instruction can then be safely replaced with a relative near jump using this special *rel32* value. For example, consider the consecutive instructions:

```
mov %rax,(%rbx)   add $32,%rax
```

Suppose that we wish to patch the mov instruction which has a three-byte x86_64 machine-code representation. Using instruction punning, we can insert a five-byte relative jump provided the last two bytes of the *rel32* value agrees with the first two bytes (0x48 0x83) of the overlapping add instruction:



Instruction punning allows jumps to replace instructions smaller than five bytes. However, the location of the trampoline is now *constrained* and cannot be placed at an arbitrary address. In the example above, the trampoline must be placed at the relative offset *rel32*=0x83480000..0x8348ffff (under the little endian byte ordering of the x84_64). This is not always possible, since the relative offset may correspond to a virtual address that is either occupied by another object (e.g., .text, .data, or an existing trampoline), or may point to an *invalid address* (e.g., NULL or underflows into the *negative addresses* range). In the example above, the *rel32* value will be interpreted as a *negative offset* since the *most significant bit* (MSB) is set. If the resulting address is negative it cannot be used as a trampoline location. As such, baseline instruction punning can only cover a subset of all patch locations for most applications.

2.2 Our Approach

Although *B0* is control flow agnostic, it is far too slow for most applications. The combination of *B1* and *B2* improves performance, but only provides partial coverage of all patch locations (between 42–94% by our Section 6 experiments). Our approach is to design a new set of *patching tactics* that can similarly patch instructions without knowledge of control flow information. Thus, if *B1*/*B2* fail, we try new tactics *T1*/*T2*/*T3* based on combinations of instruction *padding*, *punning* and *eviction*. Each new tactic increases the probability that the patching operation succeeds. The final tactic (*T3*) is also designed to trade performance for coverage, and will likely succeed in cases where previous tactics have failed. We show that the combination of the baseline and proposed patching tactics leads to very high coverage for many real-world applications. We also implement our approach in the form of the E9Patch static binary rewriting tool. Here, "E9" refers to the opcode of the x86_64 jmpq instruction that is fundamental to our approach.

Assumptions. No static binary rewriting tool is perfectly assumption-free. E9Patch aims to minimize as many assumptions as is reasonably possible, including:

- E9Patch does not assume that the input binary was compiled with a specific compiler or programming language;
- E9Patch does not assume that symbol/debug information is available and works with *stripped* binaries;
- E9Patch does not assume that control flow information is available or can be recovered.
- E9Patch does not attempt to *symbolize* the binary.

That said, E9Patch does make some minimal assumptions. For example, since E9Patch modifies executable code (e.g., the `.text` section), there is an assumption that the patched instructions are not *read from* (as distinct from *executed*) or *written to* (self-modifying code). Like all binary rewriting systems, E9Patch assumes the instrumentation/patch is *transparent*, meaning that the program behaviour is not changed unintentionally through some side channel (e.g., timings, file mappings, etc.). Finally, the current E9Patch implementation assumes that the input binary itself does not already use overlapping/punned instructions. However, it may be possible to relax this assumption in future versions.

The E9Patch tool does not use a built-in disassembler, and instead relys on instruction information (e.g., locations and sizes) to be passed in as input from a suitable frontend. The E9Patch tool will then rewrite the binary assuming this information is correct. The motivation for this design is twofold. Firstly, our patching methodology is *local* meaning that it is possible to patch specific instructions without complete disassembly information being known. Secondly, binary disassembly is known to be a hard problem by itself [1, 37]. Since E9Patch is low-level, it also retains flexibility, allowing for the integration with different disassembly techniques (partial, linear, recursive, superset [2], probabilistic [27], etc.). For the purpose of the evaluation in Section 6, we implemented a basic wrapper frontend that applies *linear disassembly* to the (`.text`) section of the input binary.

3 Patching Tactics and Strategies

The baseline instruction patching methodologies (*B1*/*B2*) do not provide sufficient coverage for most applications. In this section, we design a new set of tactics (*T1*/*T2*/*T3*) that (1) boost the coverage of instruction patching, and (2) do not require control flow information to work correctly. Here, we consider a working example based on the following instruction sequence:

```
Ins1: mov %rax,(%rbx)    Ins3: xor %rax,%rcx
Ins2: add $32,%rax       Ins4: cmpl $77,-4(%rbx)
```

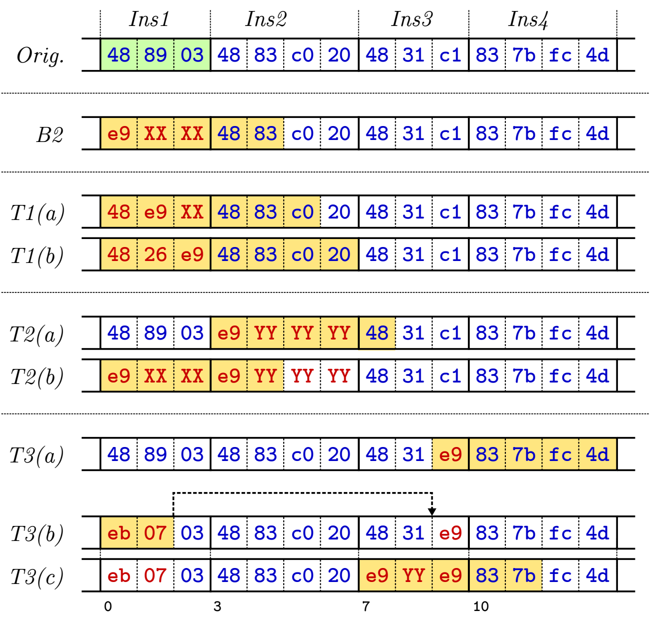
The machine code and instruction layout is shown in Figure 1 (*Orig*). We assume that the intended patch instruction is *Ins1* (highlighted). For the sake of example, we will assume that jumps to *negative relative offsets* (where the MSB of the *rel32* is set) are invalid. Thus, baseline instruction punning (Figure 1 line *B2*) yields an invalid trampoline location and cannot be used.

3.1 Tactic *T1*: Padded Jumps

The `x86_64` relative near jump is normally encoded in five bytes: one byte for the opcode and four bytes for the *rel32* offset. However, other encodings that use more bytes are possible. One idea is to *pad* the jump instruction with additional bytes in the form of *redundant instruction prefixes*. The `x86_64` supports multiple instruction prefixes (e.g., the REX prefix, segment overrides (`es`, `ss`, etc.), and operand override `0x66`) that do not change the semantics of relative near jump instructions.

Instruction padding is illustrated in Figure 1 lines *T1(a)* and *T1(b)*. Here, *T1(a)* uses a punned jump with a single byte of padding (using a redundant REX=`0x48` prefix), and *T1(b)* uses two bytes of padding (an additional redundant segment override prefix `es`=`0x26`). The more padding that is used the more constrained the relative offset becomes. For example, we have *rel32*=`0x83480000..0x8348ffff` for zero bytes of padding (*B2*), *rel32*=`0xc0834800..0xc08348ff` for one byte of padding (*T1(a)*), and *rel32*=`0x20c08348` for two bytes of padding (*T1(b)*). Assuming that negative offsets are invalid, only *T1(b)* yields a valid value.

Like baseline instruction punning (*B2*), tactic *T1* is control flow agnostic. However, the applicability of *T1* depends on the length of the patch instruction. For example, *T1* grants two additional patch attempts for the three-byte `mov` instruction from Figure 1, and this generalizes to one less than the length of the patch instruction for other cases. This also means that *T1* cannot be used to patch single-byte instructions since there is no room for additional padding. When applicable, each subsequent pun attempt is more constrained than the last. Nevertheless, even weakly constrained jumps may be invalid, as illustrated by *B2* and *T1(a)*.



Ins1
Ins2
Ins3
Ins4
Orig. 48 89 03 48 83 c0 20 48 31 c1 83 7b fc 4d
B2 e9 XX XX 48 83 c0 20 48 31 c1 83 7b fc 4d
T1(a) 48 e9 XX 48 83 c0 20 48 31 c1 83 7b fc 4d
T1(b) 48 26 e9 48 83 c0 20 48 31 c1 83 7b fc 4d
T2(a) 48 89 03 e9 YY YY YY 48 31 c1 83 7b fc 4d
T2(b) e9 XX XX e9 YY YY YY 48 31 c1 83 7b fc 4d
T3(a) 48 89 03 48 83 c0 20 48 31 e9 83 7b fc 4d
T3(b) eb 07 03 48 83 c0 20 48 31 e9 83 7b fc 4d
T3(c) eb 07 03 48 83 c0 20 e9 YY e9 83 7b fc 4d
0
3
7
10

Figure 1. Here *B2* is baseline instruction punning, tactic *T1* is *padded jumps*, tactic *T2* is *successor eviction*, and tactic *T3* is *neighbour eviction*. The patch location (*Ins1*) is highlighted in the (*Orig.*) instruction sequence. Here (`e9` + 4-bytes) is a 32-bit relative near jump, (`eb` + 1-byte) is an 8-bit relative short jump, and (`XX`/`YY`) represent byte values chosen by the rewriting tool.

3.2 Tactic *T2*: Successor Eviction

Even padded jumps may fail to find a valid trampoline location, meaning that more aggressive patching tactics may need to be employed. One idea is to relax the preservation of the successor instruction bytes, provided that an operationally equivalent *replacement instruction* can be found. For this we introduce the notion of *instruction eviction*. Essentially, instruction eviction replaces a *victim instruction* I_{Victim} with a jump instruction that targets an *evictee trampoline*. The evictee trampoline does nothing other than to execute (or emulate) I_{Victim} before jumping back. Since the evicted instruction is replaced by a jump, the byte representation also changes, making it possible to find new puns where previous attempts had failed.

Successor eviction is a two step process and is illustrated in Figure 1 *T2(a)* and *(b)*. In the first step *T2(a)*, the successor instruction (*Ins2*) is evicted using tactic *B2*, and is replaced by a jump instruction to an evictee trampoline (at some offset between `0x48000000..0x48ffffff`). For the sake of example, we shall assume a valid evictee trampoline location can be found. In the second step *T2(b)*, we essentially "reapply" *B2*/*T1* to *Ins1*. Since *Ins2* has been replaced by a jump, its byte representation has also changed, allowing for new valid puns to be discovered where previously none were available.

As with *T1*, successor eviction is control flow agnostic. Although the victim instruction is replaced by a jump, its semantics are otherwise unchanged, and the original set of jump targets is also preserved. Unlike *T1*, successor eviction can be applied to single-byte instructions. That said, instruction eviction also introduces extra redirections to evictee trampolines, and this may translate into additional performance overheads. As such, successor eviction is only applied to cases where *B1*/*B2*/*T1* failed to patch the instruction.

3.3 Tactic *T3*: Neighbour Eviction

If both *T1* and *T2* fail, another idea is to evict a "neighbouring" instruction rather than the successor. The space freed by the eviction can then be used to implement a "double" jump to the trampoline. This is the *neighbour eviction* tactic (*T3*).

Neighbour eviction requires an elaborate setup. First, a victim instruction I_{Victim} is chosen within the *unconditional short jump* distance of the patch instruction, i.e., within −128..127 bytes. Next, I_{Victim} is evicted, and replaced by two (possibly punned) relative jump instructions, $\mathcal{J}_{Victim}$ and $\mathcal{J}_{Patch}$:

1. Jump $\mathcal{J}_{Victim}$ redirects control flow from I_{Victim}'s location to I_{Victim}'s evictee trampoline. As with *T2*, this serves as a replacement of the victim instruction; and
2. Jump $\mathcal{J}_{Patch}$ redirects control flow to the trampoline implementing the original patch.

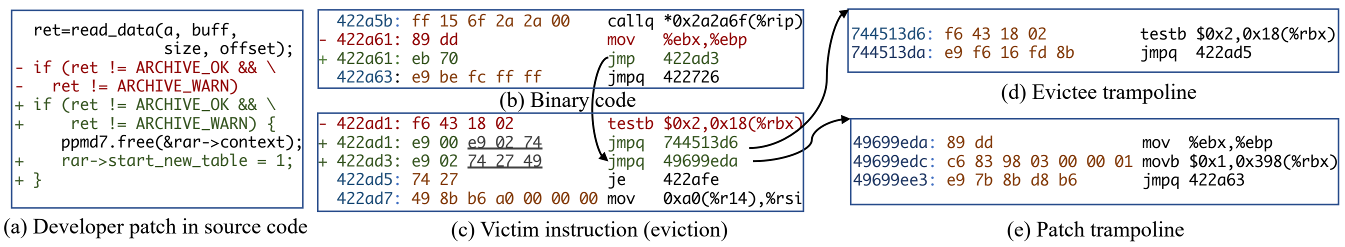
Finally, the patch location instruction is replaced by an *unconditional short jump* $\mathcal{J}_{Short}$ that redirects control flow to $\mathcal{J}_{Patch}$'s location. The patch trampoline can now be reached using a "double jump" ($\mathcal{J}_{Short}\rightarrow\mathcal{J}_{Patch}\rightarrow$trampoline) all while preserving the semantics of the victim instruction I_{Victim}. Alternatively, the victim instruction itself may happen to be a patch location. In this case, $\mathcal{J}_{Victim}$ will target I_{Victim}'s patch trampoline rather than an evictee trampoline.

Neighbour eviction is illustrated in Figure 1 *T3(a)(b)(c)*. In this example, instruction *Ins3* has been chosen for eviction. In the general case, both *Ins2* and *Ins4* are also potential candidates. Step *T3(a)* inserts a punned jump instruction ($\mathcal{J}_{Patch}$) *inside* victim *Ins3* by overwriting the last byte. In the general case, jump $\mathcal{J}_{Patch}$ may override any victim instruction byte except for the first. For the sake of example, we assume that the resulting offset *rel32*=`0x4dfc7d83` points to a valid trampoline location. Next, step *T3(b)* replaces the patch instruction with an *unconditional short jump* (opcode `0xeb` + one byte relative offset *rel8*=7). This sets up the jump $\mathcal{J}_{Short}\rightarrow\mathcal{J}_{Patch}$. Finally, step *T3(c)* replaces the victim instruction *Ins3* by a jump $\mathcal{J}_{Victim}$ to the evictee trampoline. Again, for the sake of example, we assume that offset *rel32*=`0x7b83e900..0x7b83e9ff` points to at least one valid evictee trampoline location.

Neighbour eviction (*T3*) is complex yet powerful, and can often be applied even when the other tactics have failed. The key is in the number of potential victim instructions. For example, if we assume an average instruction length of ~4 bytes, this translates into approximately 64 potential victims, meaning that at least one suitable victim is likely to be found. For this reason, neighbour eviction can boost patching coverage to at or near 100% for many applications. *T3* is also control flow agnostic since all potential jump targets are either preserved, patched, or replaced by an operationally equivalent instruction. In terms of performance, the "double jump" of neighbour eviction introduces an extra level of indirection compared to tactics *T1* and *T2*, and this can translate into additional runtime overheads. Accordingly, tactic *T3* is only applied to cases where *B1*/*B2*/*T1*/*T2* failed to patch the instruction.

Example 3.1 (Binary Repair)**.** One application of E9Patch is *binary repair* [33], i.e., fixing bugs at the binary-level rather than the source-code level. We consider a simple proof-of-concept case study based on the *use-after-free* vulnerability CVE-2019-18408[1]. Figure 2(a) shows the developer source-level patch that we intend to apply at the binary level. For the sake of example, we shall assume that the source code is unavailable, and that we choose to patch the first instruction (at address `422a61`) after the call to `free`. All of *B1*/*B2*/*T1*/*T2* fail to patch the instruction, meaning that *T3* must be used. To apply *T3*, we must evict a neighbour instruction, and in this case we choose the `testb` instruction at address `422ad1` (Figure 2(c)). The `testb` instruction is replaced by two punned jumps: $\mathcal{J}_{Victim}$ to the evictee trampoline of the evicted instruction (d), and $\mathcal{J}_{Patch}$ to the trampoline implementing the patch (e). Finally, the original instruction at address `422a61` is replaced by a short jump $\mathcal{J}_{Short}$ to $\mathcal{J}_{Patch}$.

[1] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-18408



```
  ret=read_data(a, buff,
                size, offset);
- if (ret != ARCHIVE_OK && \
-   ret != ARCHIVE_WARN)
+ if (ret != ARCHIVE_OK && \
+     ret != ARCHIVE_WARN) {
     ppmd7.free(&rar->context);
+    rar->start_new_table = 1;
+ }
```

(a) Developer patch in source code

```
  422a5b: ff 15 6f 2a 2a 00     callq *0x2a2a6f(%rip)
- 422a61: 89 dd                 mov   %ebx,%ebp
+ 422a61: eb 70                 jmp   422ad3
  422a63: e9 be fc ff ff        jmpq  422726
```

(b) Binary code

```
- 422ad1: f6 43 18 02           testb $0x2,0x18(%rbx)
+ 422ad1: e9 00 e9 02 74        jmpq  744513d6
+ 422ad3: e9 02 74 27 49        jmpq  49699eda
  422ad5: 74 27                 je    422afe
  422ad7: 49 8b b6 a0 00 00 00  mov   0xa0(%r14),%rsi
```

(c) Victim instruction (eviction)

```
744513d6: f6 43 18 02           testb $0x2,0x18(%rbx)
744513da: e9 f6 16 fd 8b        jmpq  422ad5
```

(d) Evictee trampoline

```
49699eda: 89 dd                 mov   %ebx,%ebp
49699edc: c6 83 98 03 00 00 01  movb $0x1,0x398(%rbx)
49699ee3: e9 7b 8b d8 b6        jmpq 422a63
```

(e) Patch trampoline

Figure 2. Binary repair example that fixes CVE-2019-18408 using *T3*. In sub-figure (c), the grey and underlined bytes are punned and shared by multiple instructions.

The result is essentially spaghetti code with overlapping instructions. Nevertheless, the correct patch semantics have been implemented while the set of jump targets have been preserved. For example, a jump that targets `422ad1` will execute the evictee trampoline, thereby preserving the original semantics of the evicted instruction. This example also highlights the *locality* of our patching methodology. Only two instruction locations are modified, and only partial disassembly of the region around the patch location is required. □

3.4 Strategy *S1*: Reserve Order Patching

Tactics *B1*/*B2*/*T1*/*T2*/*T3* can be used to patch individual instructions. However, many applications need to patch *multiple* instructions. Complications arise when the patching tactics interfere with each other. For example, suppose that an application needs to patch *both* instructions *Ins1* and *Ins2* from Figure 1. If we patch *Ins1* first using tactic *T1*, the relative offset (*rel32*) of the punned jump instruction will overlap with (and now depends on) *Ins2*'s specific byte values. Effectively, punning "locks-in" the byte values of any overlapping instruction. A similar problem exists for tactics *T2* and *T3*.

To manage multiple patch locations we use a *reverse order patching* strategy (*S1*). The basic idea is to patch instructions in order of "highest to lowest" address, thereby exploiting the property that instruction punning only ever introduces dependencies with successor instructions. For example, the reverse order patching strategy will patch *Ins2* first, modifying *Ins2*'s bytes, and possibly modifying/locking the bytes of *Ins3* or *Ins4* (depending on which patching tactic is applied). Only after *Ins2* is patched do we attempt to patch *Ins1*. This time, patching *Ins1* does not affect *Ins2*.

The reverse order patching strategy maintains a Boolean *lock state* for all relevant instruction bytes. Initially, all bytes are in the *unlocked* state. When a patching tactic is applied, some bytes will be *locked* to disallow further modification. An instruction byte will be locked if one of the following conditions apply:

1. *Modified*: The byte value was overwritten.
2. *Punned*: The byte value was not overwritten, but is used as part of a punned jump instruction (*B2*/*T1*/*T2*/*T3*).

The highlighted bytes in Figure 1 will be locked after the application of the corresponding tactic. For example, in Figure 1 *T3*, bytes {0, 1, 7..13} will be locked. Note that byte 2 (with value `0x03`) remains unlocked despite being part of the patch location instruction. This is because the byte was neither modified nor used by a punned jump instruction. Byte 2 can be modified by the application of a future *T3* patch operation. Tactics *T1*-*T3* are restricted to (1) only modify *unlocked* bytes, and (2) only lock bytes *after* the current patch location. This also restricts *T3* short jumps to positive offsets, effectively halving the number of potential eviction locations. However, we find that this restriction has a minimal impact in practice.

4 Memory and File Size Management

Tactics *B1*/*B2*/*T1*/*T2*/*T3* insert jumps to trampolines that must be loaded into the patched program's virtual address space. In the case of instruction punning, the corresponding trampoline locations are *constrained* by the byte values of overlapping instructions. This may prevent trampolines from being packed contiguously, potentially leading to high fragmentation and poor memory utilization. Furthermore, in the context of static binary rewriting, the file size of the patched binary must also be considered. Normally, executable code is directly `mmap`'ed from the binary (i.e., *file-backed mapping*), allowing for multiple instances of the same program to share the same physical memory resources (RAM, disk). Naïvely applying file-backed mapping to fragmented memory can significantly bloat the size of the patched binary.

Memory fragmentation may be partly mitigated by packing trampolines into the same virtual pages whenever possible—an idea first introduced by LiteInst [7]. For example, in Figure 1 *B2*, the trampoline can be placed at any relative offset within the range *rel32*=`0x83480000..0x8348ffff`. This trampoline can be grouped together with any other trampoline that happens to be placed within this range. That said, trampoline locations are often sufficiently constrained so as to prevent meaningful grouping. For example, only one exact relative offset *rel32*=`0x20c08348` is valid for Figure 1 *T1(b)*. In the worst case there will be ~1 trampoline per virtual page, leading to a very poor virtual memory utilization (e.g., ~2.8% from [7]).

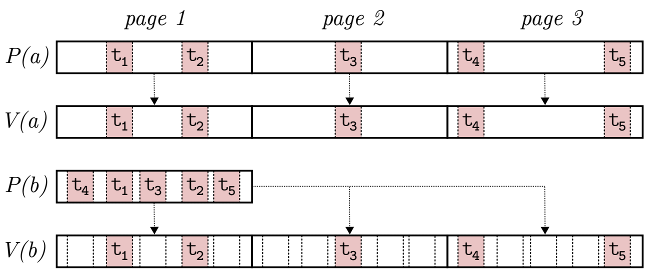


Figure 3. Physical page grouping example. Here, approach *(a)* implements a naïve one-to-one mapping between physical *P(a)* and virtual *V(a)* pages. Approach *(b)* implements *physical page grouping* by mapping a single "merged" physical page *P(b)* into the virtual address space *V(b)* three times—effectively reducing physical memory usage by two thirds.

4.1 Physical Page Grouping

Despite the potential for high virtual memory fragmentation, it may still be possible to optimize the *physical* memory usage of the patched program. For this we introduce *physical page grouping*—a space optimization designed to merge and share physical memory resources. As a motivating example, we consider a patched program using five trampolines t_1–t_5 spread over three virtual pages 1–3, as illustrated in Figure 3 *V(a)*. The memory between trampolines is not used, leading to poor virtual memory utilization. Furthermore, a naïve *one-to-one* mapping from physical *P(a)* to virtual *V(a)* memory will translate the problem into poor physical memory utilization. For example, in Figure 3 *P(a)*, a total of three (mostly empty) physical pages will be used. Assuming that *P(a)* is file-backed, this also bloats the size of the patched binary file.

Physical page grouping aims to optimize physical memory utilization by merging pages with non-overlapping trampolines. These "merged" physical pages can then be mmap'ed to the same virtual address locations as the naïve approach, effectively implementing a *one-to-many* mapping. For example, the three physical pages from Figure 3 *P(a)* can be merged into the single physical page from *P(b)*. This "merged" physical page can then be mmap'ed into the patched program's virtual address space *three* times, as shown by *V(b)*. This places each trampoline t_1–t_5 at the same virtual address as the naïve approach *V(a)*, but only uses a single physical page—reducing physical memory usage by two thirds.

The physical page grouping optimization takes as input the virtual addresses and sizes of each trampoline after all relevant instructions have been patched. It outputs a set of physical pages (to be incorporated into the rewritten binary) as well as a set of mappings (i.e., mmap calls) from physical to virtual pages that will be applied during program loading. The main challenge for physical page grouping is to find sets of pages that can be merged. For this, our E9Patch implementation divides the virtual address space into a set of *blocks* B of M consecutive pages. Here, M is some predetermined granularity that controls the aggressiveness of the optimization, and with M=1 being the most aggressive. Trampolines that span block boundaries are treated as two mini-trampolines in two different blocks. Next, a *partitioning algorithm* organizes the elements of B into a set of *groups* $G_B \subseteq \mathcal{P}(B)$ such that (1) each $b \in B$ appears in exactly one group, and (2) for all $grp \in G_B$ and for all $b_1, b_2 \in grp$, then the trampolines in b_1 and b_2 are disjoint relative to the respective block base. Each group can then be merged into a single physical block that is mapped into the patched program's virtual address space multiple times. For the example in Figure 3, we use M=1 and the partitioning algorithm yields G_B={{*page 1*, *page 2*, *page 3*}}. In general, partitioning is a *combinatorial optimization problem*, and many different partitioning algorithms are possible. For E9Patch, we found that a simple greedy algorithm gives reasonable results for reasonable performance.

Physical page grouping has the side effect of loading trampolines into redundant locations. For example, all five trampolines t_1–t_5 are loaded into each virtual page 1–3 from Figure 3 *V(b)*. However, these redundant locations remain unused, and do not affect the behaviour of the patched program. Another issue is that physical page grouping may generate large numbers of mappings. Depending on the application, this number may exceed the default mapping limit for Linux (`vm.max_map_count=65536`). One solution is to raise the mapping limit, however this requires privileged/`root` access and may not always be possible. Another solution is to use a *coarser granularity* M>1 to reduce the number of mappings in exchange for increased physical memory usage. For M≥64, the number of mappings will always be below the default system limit for a single binary. The current E9Patch implementation supports multiple granularities, allowing the user to tune the number of mappings (versus physical memory usage) accordingly.

5 Implementation

The E9Patch tool takes as input an unpatched binary (executable or shared object), disassembly information (instruction locations and sizes), a set of *patch instruction locations*, and a set of *trampoline templates*. E9Patch then outputs a rewritten binary with one of *B1*/*B2*/*T1*/*T2*/*T3* applied to each patch location instruction. The rewritten binary also incorporates the trampoline pages decided by *physical page grouping*. E9Patch is low-level by design, and can be used as the foundation for many different applications, such as binary repair, hardening and instrumentation. To the user, the rewritten binary behaves as a "drop-in" replacement of the original, with no additional dependencies or configuration. To achieve this, E9Patch directly edits/rewrites ELF files.

5.1 ELF Rewriting

The ELF file format is primarily designed to simplify linking and minimize loading time, rather than be a file format amenable to rewriting. Nevertheless, E9Patch avoids many of the complications of ELF rewriting by strictly patching existing segments *in place*. This means that data is never moved, and avoids the need to recompute ELF file offsets.

Some new data, such as executable trampoline and instrumentation code, also needs to be added to the patched binary. For this, E9Patch appends the new data to the *end* of the file, also avoiding the need to move existing data. The new physical pages must also be mapped into the program's virtual address space during program loading. To do so, E9Patch integrates a small loader into the output binary. The loader replaces the entry point, and `mmap`s the trampoline/instrumentation pages into their correct positions before returning control flow to the real entry point. We now summarize some of the main features of E9Patch.

Position Independent Executables. E9Patch can be applied to both *position independent executable* (PIE) and non-PIE binaries. Indeed, PIE binaries are becoming increasingly common in modern Linux distributions thanks to the security benefits offered by *address space layout randomization* (ASLR). Large security-sensitive programs, such as Google Chrome and Firefox, are PIE by default.

Interestingly, PIE binaries are *easier* to patch than non-PIE binaries. This is because PIE code segments will be loaded into a high memory addresses by the dynamic linker—a safe distance from the invalid negative address range. Non-PIE code is typically loaded at a low fixed addresses chosen by the (static) linker. For example, `ld` chooses a low address (e.g., `0x400000`) by default, meaning that most negative offsets will be invalid. Thus with PIE, the number of valid offsets for punned jump instructions effectively doubles. That said, it is important for static binary rewriting tools to support both PIE and non-PIE binaries. Non-PIE binaries will continue to be used into the foreseeable future.

Shared Objects/Libraries. E9Patch can be applied to shared objects/libraries (e.g., `libc.so`) in addition to executables. The rewriting process is essentially the same. However, unlike PIE, we found that negative offsets are generally incompatible with the dynamic linker. This is because other shared objects tend to be loaded into this address range.

Mixing Patched/Non-Patched Code. E9Patch does not move instructions, making it possible to safely mix patched and non-patched binary code without additional precautions. For example, the main executable may be patched but the library dependencies need not be, or vice versa. In contrast, other binary rewriting tools work by moving instructions to new locations. This can create a problem if the non-patched code calls a pointer to a function that has been relocated, i.e., the *callback problem*. To solve the issue, some tools require the entire dependency tree to be rewritten.

5.2 Limitations

The combination of tactics *T1*–*T3* can significantly boost patching coverage for many applications. However, perfect coverage is not guaranteed. This mostly occurs for hard cases, including:

(L1) virtual address space shortages,
(L2) single-byte instructions,
(L3) attempting to patch many instructions.

A program that has very large code or data segments (L1) may limit the virtual address space available for trampolines [7]. Single-byte instructions cannot be patched using *T1*, and *T3* can only target a single (punned) short jump location, thereby limiting applicability (L2). For the `x86_64`, this mostly affects `ret`, `push` and `pop`, since most other common instructions are 2 bytes or larger. Finally, since patching tactics can be interdependent, attempting to patch all (or nearly all) instructions can cause interference and limit applicability (L3). Fortunately, (L1) does not apply to most programs, and (L2) and (L3) are irrelevant for many applications. For example, a binary hardening tool that instruments all pointer dereference instructions ($\geq$2 bytes) will not be affected by (L2) nor (L3). Furthermore, (L3) is irrelevant for binary repair, one of the main application domains for E9Patch.

Assuming that an instruction cannot be patched, the corrective action largely depends on the application. For example, binary hardening can usually tolerate some reduced coverage. For other applications that prioritize coverage over performance, using *B0* as a fallback may be appropriate.

6 Evaluation

In this section we evaluate the timing, coverage, file size and scalability of a prototype version of E9Patch. We also present a practical application in the form of binary memory error detection using *low fat pointers* [12, 13].

6.1 Performance

To evaluate the performance of E9Patch we use the full[2] SPEC2006 [18] benchmark suite, including programs implemented in `C`, `C++`, and Fortran. We compile each benchmark using the default system compiler (`gcc`/`g++`/`gfortran`). We also choose to compile in non-PIE mode in order to make patching more challenging. We also instrument several default binaries that were installed with Ubuntu 16.04.6 LTS. For this, we choose binaries that were used in the preparation of this paper (such as `pdflatex`, etc.) as well as some prominent shared library dependencies. We have also tested E9Patch on many other system binaries not included in Table 1, and all work as expected. Finally, to demonstrate

[2]Excluding `481.wrf` which failed to compile using modern `gfortran`.

Table 1. Patching Statistics. Binaries marked by (†) are *position independent executables* (PIE).

Binary	Size	Jmp/Jcc instructions (*A1*)								Heap write instructions (*A2*)							
	(MB)	#Loc	Base%	*T1%*	*T2%*	*T3%*	Succ%	Time%	Size%	#Loc	Base%	*T1%*	*T2%*	*T3%*	Succ%	Time%	Size%
perlbench	1.25	36821	86.88	7.40	1.45	4.27	100.00	459.59	174.28	7522	71.16	24.42	1.18	3.23	100.00	244.90	116.66
bzip2	0.07	1484	79.85	13.61	2.22	4.31	100.00	280.85	199.45	1044	68.39	26.05	2.49	3.07	100.00	279.67	170.95
gcc	3.77	97901	85.66	8.29	1.62	4.43	100.00	364.41	164.50	14328	70.60	24.95	0.68	3.78	100.00	148.73	109.90
bwaves	0.08	314	71.34	2.87	0.32	25.48	100.00	107.08	137.01	1168	92.55	7.36	0.00	0.09	100.00	139.02	142.43
gamess	12.22	125620	59.91	15.01	5.05	19.76	99.73	226.16	131.14	279592	87.58	9.65	0.50	2.20	99.94	321.89	136.93
mcf	0.02	295	68.47	20.00	4.41	7.12	100.00	194.92	203.75	220	75.91	20.00	1.36	2.73	100.00	141.02	221.51
milc	0.14	1940	80.62	13.40	1.29	4.69	100.00	115.03	157.13	699	84.84	13.16	0.29	1.72	100.00	117.54	119.14
zeusmp	0.52	3191	53.74	11.66	2.98	30.30	98.68	145.34	125.28	6106	82.61	12.15	0.39	4.67	99.82	131.50	128.74
gromacs	1.20	12058	80.19	11.49	1.38	6.94	100.00	116.16	133.01	16940	93.87	5.50	0.11	0.53	100.00	148.07	123.71
cactusADM	0.91	12847	78.94	13.32	2.30	5.44	100.00	101.43	140.70	5420	86.85	11.62	0.41	1.13	100.00	119.48	113.45
leslie3d	0.18	2584	44.43	27.67	12.46	15.44	100.00	151.89	174.56	2761	91.34	8.22	0.04	0.40	100.00	172.08	138.47
namd	0.33	4879	73.42	13.88	2.75	9.96	100.00	146.78	154.81	2498	71.46	28.14	0.20	0.20	100.00	138.01	120.42
gobmk	4.03	17912	75.88	14.72	2.57	6.83	100.00	368.97	113.80	2777	79.33	15.56	0.94	4.18	100.00	179.24	102.30
dealII	4.20	61317	71.31	14.99	4.50	9.19	100.00	386.08	144.34	25590	80.47	17.83	0.17	1.52	99.99	168.86	112.27
soplex	0.49	10125	79.72	11.57	2.58	6.13	100.00	244.23	162.93	4188	83.05	15.28	0.53	1.15	100.00	162.98	121.64
povray	1.19	20520	86.92	7.39	1.49	4.20	100.00	408.33	146.34	9377	84.50	13.46	0.37	1.66	100.00	186.36	116.37
calculix	2.17	30343	70.48	17.75	2.89	8.88	100.00	132.78	141.24	32197	85.62	13.02	0.38	0.98	100.00	126.13	128.26
hmmer	0.33	6748	77.71	13.96	1.99	6.34	100.00	182.94	174.52	3061	75.11	22.64	0.65	1.60	100.00	468.53	129.85
sjeng	0.16	3473	83.01	10.14	1.79	5.07	100.00	444.13	177.02	683	84.77	12.74	0.15	2.34	100.00	134.78	123.32
GemsFDTD	0.58	9120	41.62	17.28	21.44	19.66	100.00	104.78	166.74	10345	93.23	6.54	0.04	0.18	100.00	111.64	132.30
libquantum	0.05	732	75.55	15.85	3.42	5.19	100.00	325.81	190.57	186	76.34	17.74	0.00	5.91	100.00	269.68	139.82
h264ref	0.58	9920	80.30	13.58	1.22	4.90	100.00	206.61	151.60	4981	81.87	15.42	0.80	1.91	100.00	178.89	122.04
tonto	6.21	48247	52.65	22.84	8.63	15.88	100.00	196.21	125.54	164788	90.05	9.09	0.15	0.71	100.00	192.72	141.53
lbm	0.02	106	67.92	17.92	3.77	10.38	100.00	103.80	193.33	111	93.69	6.31	0.00	0.00	100.00	110.13	148.74
omnetpp	0.79	9568	78.08	13.96	2.16	5.79	100.00	203.90	135.45	5020	74.12	18.57	3.01	4.30	100.00	144.81	117.53
astar	0.05	769	78.54	13.78	2.21	5.46	100.00	287.64	180.98	491	72.91	23.01	0.61	3.46	100.00	137.64	152.03
sphinx3	0.21	3500	79.20	12.17	2.03	6.60	100.00	196.27	170.99	1159	73.94	22.95	0.78	2.33	100.00	129.17	123.55
xalancbmk	5.99	81285	75.66	14.10	3.50	6.74	100.00	474.07	137.04	32761	79.51	17.61	0.43	2.45	100.00	130.16	111.38
#Total/Avg%	47.74	613619	72.79	13.95	3.73	9.48	99.94	210.81	157.43	636013	81.63	15.68	0.60	2.09	99.99	164.71	130.90
inkscape† *0.91*	15.44	195731	97.83	1.31	0.86	0.00	100.00	–	130.40	105431	99.96	0.03	0.01	0.00	100.00	–	109.58
gimp 2.8.16	5.75	71321	71.75	18.69	2.49	7.08	100.00	–	135.74	15730	84.83	12.59	0.64	1.95	100.00	–	106.00
vim† *7.4*	2.44	72221	99.18	0.23	0.60	0.00	100.00	–	173.31	13279	99.92	0.02	0.06	0.00	100.00	–	110.77
git 2.7.4	1.87	44441	80.06	11.91	2.14	5.88	100.00	–	169.16	9072	68.06	27.62	1.16	3.16	100.00	–	113.60
pdflatex 2.6	0.91	22105	82.05	10.46	2.06	5.42	100.00	–	168.72	6060	70.61	24.97	1.25	3.17	100.00	–	118.70
xterm 322	0.54	11593	79.12	12.45	3.04	5.39	100.00	–	166.23	2681	89.11	9.40	0.41	1.08	100.00	–	113.16
evince† *3.18.2*	0.42	3636	99.59	0.30	0.11	0.00	100.00	–	131.63	716	99.86	0.00	0.14	0.00	100.00	–	107.86
make 4.1	0.21	4807	79.34	12.96	1.71	5.99	100.00	–	182.78	1383	74.98	20.46	0.94	3.62	100.00	–	125.48
libc.so 2.23	1.87	52393	81.19	11.55	2.23	5.03	100.00	–	247.67	24686	74.32	21.98	1.05	2.64	100.00	–	203.87
libc++.so 6.0.21	1.57	20593	75.14	13.02	4.60	7.24	100.00	–	184.99	15442	67.56	27.76	0.99	3.68	100.00	–	168.80
Chrome† *78.0*	152.51	3800565	93.20	4.68	1.87	0.25	100.00	–	226.31	2624800	99.38	0.49	0.11	0.01	100.00	–	197.68
FireFox† *70.0*	0.52	13971	98.02	0.54	1.44	0.00	100.00	–	269.22	7355	99.90	0.10	0.00	0.00	100.00	–	208.06
libxul.so 70.0	115.03	1463369	68.55	15.08	5.26	11.10	99.99	–	194.55	666109	75.72	20.61	0.62	3.06	100.00	–	174.22

scalability, we instrument some very large binaries such as Google Chrome [17] and FireFox (`libxul.so`) [16].

E9Patch is a general binary rewriting tool that has many potential applications, such as binary repair, instrumentation and hardening. Typically, binary repair will focus on a few locations corresponding to bugs (e.g., see Example 3.1), whereas instrumentation/hardening will need to modify multiple locations. For this evaluation we focus on instrumentation as it is the more challenging application. Specifically, we choose two test applications (*A1*/*A2*) that instrument:

1. *A1*: All `jmp`/`jcc` jump instructions; and
2. *A2*: All instructions that may write to heap pointers.

The former is a rough analogue for basic-block counting which is a common benchmark for static binary rewriting tools. However, since E9Patch does not have basic block information by design, we instrument jump instructions instead. The latter will be used for a hardening application presented in Section 6.3. For these experiments, we use an "empty" instrumentation that merely executes/emulates the displaced instruction before returning control flow back to the main program. This will demonstrate the baseline performance of E9Patch's patching methodology.

The patching statistics are shown in Table 1. Here, we instrument the (`.text`) section for each application. Column (Size) is the binary size in *MB*, (#Loc) is the number of patch locations, (Base%) is the percentage of successful patchings using baseline methods (*B1*+*B2*), (*T1*/*T2*/*T3*%) is the percentage of successful patchings for each tactic, (Succ%) is the overall percentage of successful patchings (*B1*+*B2*+*T1*+*T2*+*T3*), (Time%) is the overall runtime performance overhead, and (Size%) is the overall output binary size over the original. For the latter, the *physical page grouping* optimization (Section 4) has been applied with a granularity of M=1 (i.e., maximum aggression). All experiments are run on a Xeon Silver 4114 Processor (2.20GHz with 32GB of RAM).

Coverage. Each patching tactic is not guaranteed to succeed, meaning that the *coverage* (i.e., the ratio of successfully patched instructions) is a concern. Despite this, the Table 1 results show that E9Patch achieves very high coverage, and can patch nearly every benchmark with a 100.00% score. In total, E9Patch patches ~1.05×10^7 instructions while only 1098 fail. The exceptions are discussed below.

Table 1 also shows the relative coverage breakdown for each patching tactic *T1*-*T3*. Here, (*Base*%) represents the baseline coverage if *B1*/*B2* are used in isolation. In this case, only 72.79% of all jump instructions and 81.63% of all heap write instructions will be patched. Each subsequent tactic, *T1*-*T3*, improves the coverage, allowing for more instructions to be successfully patched. Our results also highlight the importance of the *neighbour eviction* (*T3*) tactic. Without *T3*, the overall coverage would be merely ~90.5% (i.e., *Base*+*T1*+*T2*) for *A1* rather than ~100%. This is because *T3* by itself has a high coverage, and can be used to patch instructions that could not be handled by other tactics.

The Table 1 results also highlight a clear difference between PIE and non-PIE binaries. Since PIE binaries allow trampolines to be placed in the negative address range, the probability that any given patching tactic succeeds is much higher. Even the baseline (Base%) for PIE binaries is >93%. This result is important since PIE binaries are becoming increasingly common in modern Linux distributions thanks to the enhanced security benefits of address space layout randomization (ASLR).

Despite the overall success, some benchmarks, such as `gamess` and `zeusmp`, did not achieve 100% coverage. On closer examination, both of these programs statically allocate very large (`.bss`) sections, and this limits the usable virtual address space available for trampolines, making instruction patching more difficult (see limitation (L1) from Section 5.2). Even under these conditions, E9Patch can still patch >98.5% of all instructions. Most of the other tested binaries (including web browsers) do not make large static allocations, and are therefore not affected by (L1). Finally, we

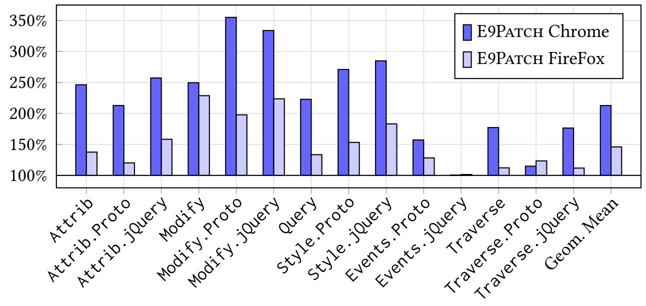


Figure 4. Relative E9Patch runtime overheads of Chrome and FireFox using the Dromaeo DOM browser benchmarks.

note that E9Patch can patch 100% of all instructions when `gamess` and `zeusmp` are recompiled in PIE mode.

File Size. Each patched instruction makes use of a trampoline that must be incorporated into the output binary. Since trampoline locations cannot be fully controlled, there is the potential for high address space fragmentation and file size bloat. With *physical page grouping* (Section 4) enabled, we see that the overall file size is more manageable at +57.43% for jump instructions (*A1*) and +30.90% for heap write instructions (*A2*). We also reran each benchmark with physical page grouping disabled, i.e., by using a naïve one-to-one mapping between physical and virtual memory. In this case, the average file size balloons to +2239.83%/+568.96% for *A1*/*A2* respectively. This highlights the importance of physical memory optimization when large numbers of instructions need to be patched.

Runtime Performance. To measure the performance, we run each of the SPEC2006 benchmarks and compare the overhead versus the original binary. We only measure the performance for SPEC since other programs do not have standard benchmarks. Furthermore, we will measure the performance for web browsers separately. Overall, we see that E9Patch introduces a +110.81% overhead for jump instructions (*A1*), and a +64.71% overhead for heap write instructions (*A2*).

To maximize scalability, E9Patch avoids relocating binary code and preserves the set of jump targets. In contrast, other static binary rewriting tools more aggressively relocate instructions, allowing for patch/instrumentation code to be *inlined* rather than jumping to/from trampolines. Inlining generally gives better performance assuming that the binary can be rewritten correctly. For example, there is a +60.48% overhead for Mulitverse [2] (empty instrumentation), a +62% overhead for PEBIL [21] (basic block counting), and a ~70% overhead for DynInst [3] (basic block counting). Compared to inlined instrumentation, our approach executes (at least) two additional instructions (2× `jmpq`) which incurs extra overheads. The trade-off is a robust design that does not need control flow information, allowing E9Patch to scale to very large binaries.

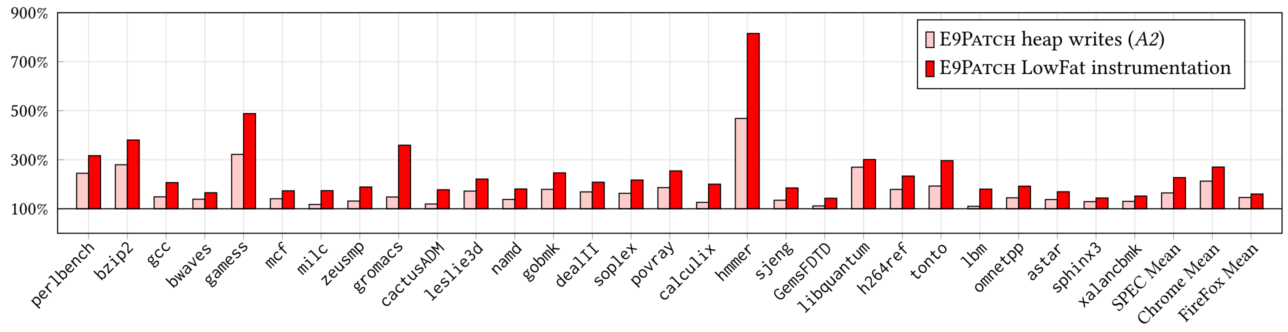


Figure 5. SPEC2006 and web browser timings for heap writes (Table 1/Figure 4) and LowFat instrumentation.

Language Agnosticism. For the evaluation, we tested programs compiled from (combinations of) C, C++, Fortran and (inlined) assembly. Because E9Patch is very low-level, it is also language agnostic by design, meaning that E9Patch is able to instrument/patch programs compiled from a variety of programming languages. In contrast, some existing binary rewriting tools incorporate compiler or language-specific assumptions in order to implement control flow recovery. Such assumptions may fail when applied to binaries compiled from other languages.

6.2 Scalability: Chrome and FireFox

One of the core aims of E9Patch is to scale to very large binaries. To evaluate scalability, we use E9Patch to instrument web browsers such as Google Chrome [17] and FireFox [16]. Each browser has a binary size exceeding 100*MB*—an order of magnitude larger than the largest SPEC benchmark (`gamess`). We found that the Chrome (`.text`) section contains a mixture of data and code, which proved to be a challenge for our prototype linear disassembler frontend. To work around the issue, we only disassemble after the `ChromeMain` symbol (which still represents >97.5% of the `.text` section). FireFox also arranges its binaries differently, with the bulk of the code placed into a shared object (`libxul.so`), which is patched using E9Patch. To measure the performance, we use the Dromaeo *Document Object Model* DOM browser benchmarks [11]. We choose these benchmarks to minimize the execution time spent in *Just-in-Time* (JIT) compiled Javascript code. For this experiment we instrument the *heap write instructions* application (*A2*), and the instrumentation statistics are shown in Table 1.

The results are shown in Figure 4. Overall, we see that E9Patch introduces a ~113% overhead for Chrome and a ~46% for FireFox. These results are consistent with the performance measurements for the SPEC benchmarks. FireFox seems to be less sensitive to the E9Patch instrumentation compared to Chrome, possibly because FireFox spends more time in JIT'ed code or non-instrumented shared objects. Regardless, our results show that E9Patch is able to scale to very large binaries such as web browsers.

6.3 Application: Binary Heap Write Hardening

As a proof-of-concept demonstration application, we choose to harden binaries against heap pointer spatial memory errors (e.g., buffer overflows). We choose to only instrument writes since out-of-bound reads are not necessarily errors at the binary level [5]. We also exclude non-heap pointer instructions using registers `%rsp` (stack) and `%rip` (globals). To detect spatial memory errors, we use a variant of *low fat pointers* [12, 13]—a method for encoding bounds information (i.e., base+size) into the bit representation of the object pointer itself. However, the default low fat pointer instrumentation schema of [12] is non-local and therefore difficult to apply at the instruction-level. Instead, we use low fat pointers to enforce *redzones* by ensuring that the property ($p - base(p) \geq 16$) holds for all pointer writes. Here, p is the written-to pointer, $base(p)$ is the low fat pointer operation that retrieves the object base address [12], and 16 is the size of the *redzone* in bytes. Pointer p is calculated by converting the patch location instruction into an `x86_64` *load effective address* (`lea`). Next, p is passed to a *redzone-check function* that is called by each trampoline. Finally, the standard `libc` memory allocation functions (`malloc`, `calloc`, etc.) are replaced by `LD_PRELOAD`'ing a low fat runtime library (`liblowfat.so`) [23]. The library has also been modified to insert redzones around each allocated object.

The results are shown in Figure 5. Here we compare against the empty instrumentation of Table 1. For the SPEC benchmarks, the overall overhead increases from +64.71% (*A2*) to +127.27% for heap write bounds checking. For the browsers Chrome/FireFox, the overall overhead increases from +113%/+46% to +170%/+60% respectively. Higher overheads are to be expected since bounds checking executes more instructions. The overhead for source-level instrumentation can be lower, sometimes as little as +13% [12]. However, source-level instrumentation can be inlined and optimized by the compiler, something that is difficult to replicate at the binary level. Furthermore, the source-level implementation of LowFat [23] only supports C/C++, whereas E9Patch works on binaries and does not assume source code availability. Finally, we note that our implementation is a proof-of-concept that can likely be optimized as future work.

7 Related Work

We briefly review the related work and compare it against E9Patch. See [40] for a recent survey.

Static Rewriting Tools. Many different static binary rewriting systems and tools have been proposed. Some tools, such as Vulcan [35], Alto [28], SASI [14], PEBIL [21], and Diablo [9], assume that the input binary was produced by a specific/specialized compiler or that symbol/debug information has been preserved (i.e., *non-stripped*). Unlike E9Patch, these tools do not work on binaries that break these assumptions. Other static binary rewriters, such as (static) DynInst [6], BIRD [30], and PSI [42], attempt to relax these assumptions in exchange for better compatibility. To do so, these tools typically implement some combination of (1) signal handlers (e.g., `SIGTRAP`), (2) non-punned jumps replacing one or more instruction, or (3) global binary rewriting that inlines instrumentation as necessary. Signal handlers are generally too slow for most applications, and the alternatives require control flow information in order to safely rewrite the binary. Static analysis-based control flow recovery generally relies on assumptions/heuristics that are known not to scale [2]. Alternatively, control flow information can be recovered dynamically by the rewritten binary, e.g., by using *address translation* [34, 42] to effectively implement a hybrid static/dynamic design. However, even this may suffer from the *callback problem*, where non-patched code calls a pointer to a function that has been relocated. One solution is to globally rewrite all indirect calls/jumps—including the entire shared library dependency tree. In contrast, E9Patch uses a *local* binary rewriting methodology that is applicable to individual binaries, requires only partial disassembly information, and preserves the set of jump targets—thereby eliminating the need for control flow recovery (of any kind).

Other tools such as Egalito [41], SecondWrite [32], and McSema [25] attempt to lift binary code into an *intermediate representation* (e.g., LLVM IR [22]) that can be recompiled into a new binary. Similarly, tools such as Uroboros [37, 38] and RetroWrite [10] attempt to *disassemble* binaries into a form amenable to *reassembly*, possibly after modification. To work correctly, these tools make several assumptions about the input binary, such as assuming specific languages (e.g., `C` for Uroboros/RetroWrite) or *position independent code* (Egalito/RetroWrite). In contrast, E9Patch can statically rewrite binaries without making such assumptions. Similarly, Mulitverse [2] also aims to minimize assumptions by using a "brute force" disassembly over all possible offsets. However, Mulitverse also implements a global rewriting approach that inherits the limitations described above, such as requiring that all shared library dependencies be rewritten.

Dynamic Rewriting/Instrumentation Tools. An alternative to static is *dynamic rewriting*, which patches binary code at runtime as the program executes. Dynamic rewriting can be scalable since dynamic analysis tends to be accurate whereas static analysis tends to be approximate. While static rewriting can be done offline (rewrite once, execute many times), dynamic rewriting is done online, and this can add additional runtime performance overheads. Pin [24] and DynamoRIO [4] dynamically analyze and instrument programs using a callback mechanism. These tools use *just-in-time* (JIT) recompilation of instrumented functions and basic blocks "on-the-fly". This requires a complex infrastructure, and the program is JIT'ed rather run "natively". As such, these tools are generally too heavyweight for some applications such as binary repair. The DynInst [6] framework also supports dynamic instrumentation using a similar methodology to that of the static case.

LiteInst [7] originally proposed instruction punning for dynamic instrumentation rather than static binary rewriting. Like E9Patch, LiteInst uses alternative tactics should baseline instruction punning (*B2*) fail:

- Instrument a predecessor instruction from the same basic block; or
- Replace overlapping instructions with illegal opcodes.

The former requires control flow information (i.e., basic blocks) in order to ensure that the instrumentation will be called (the previous instruction in memory is not necessarily the last executed), and the latter requires control flow information in order to avoid expensive signal handlers. Since E9Patch has no knowledge of control flow information, neither are appropriate for our setting. Finally, since static rewriting is offline, E9Patch can apply more aggressive optimizations such as physical page grouping.

8 Conclusion

This paper presented E9Patch, a powerful and scalable static binary rewriting tool. The key idea behind E9Patch is to exclusively use *control flow agnostic* binary rewriting methodologies that can safely patch `x86_64` instructions without the need for (or knowledge of) control flow information. By doing so, E9Patch can statically rewrite binaries without the need for a control flow recovery step and any associated assumptions/heuristics.

However, existing binary rewriting methods are either not control flow agnostic (e.g., instruction relocation), suffer from poor performance (e.g., `int3`/`SIGTRAP`), or suffer from poor coverage (e.g., *instruction punning* [7]). To solve this problem, we develop a new suite of instruction patching tactics and strategies—such as instruction *padding* and *eviction*—that are both control flow agnostic, have good performance, and have very good (near 100%) coverage for many common applications. As such, E9Patch is very robust, and is able to scale to very large binaries (including web browsers such as FireFox [16] and Chrome [17]), all while maintaining reasonable performance and memory overheads.

References

[1] D. Andriesse, X. Chen, V. Van Der Veen, A. Slowinska, and H. Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale X86/X64 Binaries. In *Security Symposium*. USENIX.

[2] E. Bauman, Z. Lin, and K. Hamlen. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *Network and Distributed System Security Symposium*. Internet Society.

[3] A. Bernat and B. Miller. 2011. Anywhere, Any-time Binary Instrumentation. In *Workshop on Program Analysis for Software Tools*. ACM.

[4] D. Bruening. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.D. Dissertation.

[5] D. Bruening and Q. Zhao. 2011. Practical Memory Checking with Dr. Memory. In *Code Generation and Optimization*. IEEE.

[6] B. Buck and J. Hollingsworth. 2000. An API for Runtime Code Patching. *High Performance Computing Applications* 14, 4 (2000).

[7] B. Chamith, B. Svensson, L. Dalessandro, and R. Newton. 2017. Instruction Punning: Lightweight Instrumentation for x86-64. In *Program Design and Implementation*. ACM.

[8] L. Davi, A. Sadeghi, and M. Winandy. 2011. ROPdefender: A Detection Tool to Defend Against Return-oriented Programming Attacks. In *Symposium on Information, Computer and Communications Security*. ACM.

[9] B. De Sutter, B. Bus, and K. De Bosschere. 2005. Link-time Binary Rewriting Techniques for Program Compaction. (2005).

[10] S. Dinesh, N. Burow, D. Xu, , and M. Payer. 2020. RetroWrite : Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *Security and Privacy*. IEEE.

[11] Dromaeo 2020. Dromaeo Benchmarks. http://dromaeo.com/.

[12] G. Duck and R. Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *Compiler Construction*. ACM.

[13] G. Duck, R. Yap, and L. Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers. In *Network and Distributed System Security Symposium*. Internet Society.

[14] U. Erlingsson and F. Schneider. 2000. SASI Enforcement of Security Policies: a Retrospective. In *DARPA Information Survivability Conference and Exposition*. IEEE.

[15] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Computer and Communications Security*. ACM.

[16] Firefox 2020. Firefox Web Browser. https://www.mozilla.org/.

[17] Google Chrome 2020. Google Chrome Web Browser. https://www.google.com/chrome/.

[18] J. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *Computer Architecture News* 34, 4 (2006).

[19] J. Hiser, A. Nguyen-Tuong, W. Hawkins, M. McGill, M. Co, and J. Davidson. 2017. Zipr++: Exceptional Binary Rewriting. In *Workshop on Forming an Ecosystem Around Software Transformation (FEAST '17)*. ACM.

[20] Y. Hu, Y. Zhang, and D. Gu. 2019. Automatically Patching Vulnerabilities of Binary Programs via Code Transfer From Correct Versions. *IEEE Access* 7 (2019).

[21] M. Laurenzano, M. Tikir, L. Carrington, and A. Snavely. 2010. PEBIL: Efficient Static Binary Instrumentation for Linux. In *Symposium on Performance Analysis of Systems Software*.

[22] LLVM. 2020. https://llvm.org.

[23] LowFat 2020. LowFat. https://github.com/GJDuck/LowFat.

[24] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Programming Language Design and Implementation*. ACM.

[25] McSema. 2020. https://github.com/lifting-bits/mcsema.

[26] X. Meng and B. Miller. 2016. Binary Code is Not Easy. In *International Symposium on Software Testing and Analysis*. ACM.

[27] K. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin. 2019. Probabilistic Disassembly. In *International Conference on Software Engineering*. IEEE.

[28] R. Muth, S. Debray, S. Watterson, and K. De Bosschere. 2001. Alto: A Link-time Optimizer for the Compaq Alpha. *Software: Practice and Experience* 31 (2001).

[29] S. Nagy and M. Hicks. 2019. Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing. In *Security and Privacy*. IEEE.

[30] S. Nanda, W. Li, L. Lam, and T. Chiueh. 2006. BIRD: Binary Interpretation using Runtime Disassembly. In *Symposium on Code Generation and Optimization*. IEEE.

[31] N. Nethercote and J. Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Programming Language Design and Implementation*. ACM.

[32] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. Keromytis. 2011. Retrofitting Security in COTS Software with Binary Rewriting. In *Future Challenges in Security and Privacy for Academia and Industry*. Springer.

[33] E. Schulte, J. DiLorenzo, W. Weimer, and S. Forrest. 2013. Automated Repair of Binary and Assembly Programs for Cooperating Embedded Devices. In *Architectural Support for Programming Languages and Operating Systems*. ACM.

[34] M. Smithson, K. Elwazeer, K. Anand, A. Kotha, and R. Barua. 2013. Static Binary Rewriting without Supplemental Information: Overcoming the Tradeoff between Coverage and Correctness.

[35] A. Srivastava, A. Edwards, and H. Vo. 2001. *Vulcan: Binary Transformation In A Distributed Environment*. Technical Report MSR-TR-2001-50.

[36] L. Van Put, B. De Sutter, M. Madou, B. De Bus, D. Chanet, K. Smits, and K. De Bosschere. 2005. LANCET: A Nifty Code Editing Tool. *Software Engineering Notes* 31, 1 (Sept. 2005).

[37] S. Wang, P. Wang, and D. Wu. 2015. Reassembleable Disassembling. In *Security Symposium*. USENIX.

[38] S. Wang, P. Wang, and D. Wu. 2016. UROBOROS: Instrumenting Stripped Binaries with Static Reassembling. In *Software Analysis, Evolution, and Reengineering*. IEEE.

[39] R. Wartell, V. Mohan, K. Hamlen, and Z. Lin. 2012. Securing Untrusted Code via Compiler-agnostic Binary Rewriting. In *Annual Computer Security Applications Conference*. ACM.

[40] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl. 2019. From Hack to Elaborate Technique - A Survey on Binary Rewriting. *Computing Surveys* 52, 3 (2019).

[41] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. Wu, J. Yang, and V. Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. ACM.

[42] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar. 2014. A Platform for Secure Static Binary Instrumentation. In *Virtual Execution Environments*. ACM.

[43] M. Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Security Symposium*. USENIX.