Libpatch

Dynamic patching of binaries in userspace

Olivier Dion

Polytechnique Montréal
Dorsal laboratory

Summary

Summary

About the DORSAL lab at Polytechnique Montreal

- Monitoring and Debugging of High Performance Distributed Heterogeneous Systems, in collaboration with Ericsson, Ciena, AMD and EfficiOS.
- Dynamic instrumentation (uftrace, LTTng, libpatch), hardware tracing.
- GPU tracing, profiling and debugging (ROCm, ROCgdb and Theia Trace Compass).
- Runtime verification (lower overhead alternatives to ASan and TSan).
- Scalable trace analysis and visualisation (parallel Theia Trace compass).
- Trace analysis with Machine Learning (Trace Compass).

Summary

What is Libpatch?

C library

- Specializes in insertion of probes at runtime.
- Minimizes probes insertion and runtime overheads.
- Maximizes coverage (probe placement).

What is Libpatch?

C library

- Specializes in insertion of probes at runtime.
- Minimizes probes insertion and runtime overheads.
- Maximizes coverage (probe placement).

Supported architectures

- x86-64
- x86 (planned)
- arm (planned)
- aarch64 (planned)

What is Libpatch?

C library

- Specializes in insertion of probes at runtime.
- Minimizes probes insertion and runtime overheads.
- Maximizes coverage (probe placement).

Supported architectures

- x86-64
- x86 (planned)
- arm (planned)
- aarch64 (planned)

Dependencies

- capstone
- elfutils (libdw)
- libunwind (x86-64 only)
- liburcu (x86-64 only)

What is Libpatch? (continuation)

Kernel features

- membarrier(2) expedited sync core (4.14)
- PTRACE_SEIZE and PR_SET_PTRACER (3.4)
- procfs(5)

What is Libpatch? (continuation)

Kernel features

- membarrier(2) expedited sync core (4.14)
- PTRACE_SEIZE and PR_SET_PTRACER (3.4)
- procfs(5)

Toolchain

- Only glibc (for now)
- *gcc* ≥ 8
- *clang* ≥ 8

What is Libpatch? (continuation)

Kernel features

- membarrier(2) expedited sync core (4.14)
- PTRACE_SEIZE and PR_SET_PTRACER (3.4)
- procfs(5)

Toolchain

- Only glibc (for now)
- *gcc* ≥ 8
- *clang* ≥ 8

Signal ownership

- SIGTRAP (optional)
- SIGILL (don't mask it!)

Libpatch's API

```c
/* Library management. */
patch_err patch_init(const patch_opt *options, size_t options_count);
patch_err patch_fini(void);
patch_err patch_configure(const patch_opt *option);

/* Patches manipulation. */
patch_err patch_queue(uint64_t flags, patch_op *op);
patch_err patch_cancel(uint64_t cookie);
patch_err patch_commit(patch_result **results, size_t *results_count);
patch_err patch_uninstall_all(void);

/* Memory cleanup. */
patch_err patch_gc(void);
patch_err patch_drop_results(patch_result *results, size_t results_count);

/* Thread specific actions. */
patch_err patch_unwind(size_t at);
```

Quick demo

```c
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

#include <libpatch/patch.h>

static void probe(struct patch_probe_context *ctx, bool ret);
extern int func(int x);

static int exit_value = 0;

int func(int x)
{
        printf("func: x=%d\n", x);
        return x;
}
```

Quick demo (continuation)

```
static void probe(struct patch_probe_context *ctx, bool ret)
{
        int *e;
        int x;

        x = (int)ctx->gregs[PATCH_X86_64_RDI];

        if (ret) {
                printf("ret-probe: x=%d\n", x);
        } else {
                printf("probe: x=%d\n", x);
        }

        e = ctx->user_data;

        (*e)++;
}
```

Quick demo (continuation)

```
static void install_probe(void)
{
        patch_result *results;
        size_t results_count;
        patch_err err;
        patch_op op = {
                .type           = PATCH_OP_INSTALL,
                .addr.func_sym  = "func",
                .probe          = probe,
                .user_data      = &exit_value,
        };

        patch_init(NULL, 0);
        patch_queue(PATCH_FENTRY, &op);
        patch_commit(&results, &results_count);
        patch_drop_results(results, results_count);
}
```

Quick demo (continuation)

```
int main(void)
{
        int x;

        x = random();

        install_probe();

        printf("main: x=%d\n", x);

        func(x);

        patch_fini();

        func(x);

        return exit_value == 2 ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Quick demo (continuation)

```
main: x=1804289383
probe:x=1804289383
func: x=1804289383
ret-probe: x=243548224
func: x=1804289383
```

Summary

Some definitions

Probe
: A user defined procedure to be called for instrumentation purpose.

Some definitions

Probe
: A user defined procedure to be called for instrumentation purpose.

Probe site
: Virtual address desired to be instrumented by the user. Also called the patch origin.

Some definitions

Probe
: A user defined procedure to be called for instrumentation purpose.

Probe site
: Virtual address desired to be instrumented by the user. Also called the patch origin.

Trampoline
: Intermediate instructions for jumping further in the program.

Some definitions

Probe
: A user defined procedure to be called for instrumentation purpose.

Probe site
: Virtual address desired to be instrumented by the user. Also called the patch origin.

Trampoline
: Intermediate instructions for jumping further in the program.

Handler
: Procedure that saves the context of the current computation, calls a probe, restores the context and returns to the computation continuation.

Some definitions (continuation)

Out of Line Execution (OLX) buffer
: Relocated instructions from the probe site. Also called the computation continuation.

Some definitions (continuation)

Out of Line Execution (OLX) buffer
: Relocated instructions from the probe site. Also called the computation continuation.

Patch
: A patch is a divertion of the current computation, for calling a user defined callback and thereafter executing the patched instructions, before continuing the computation.

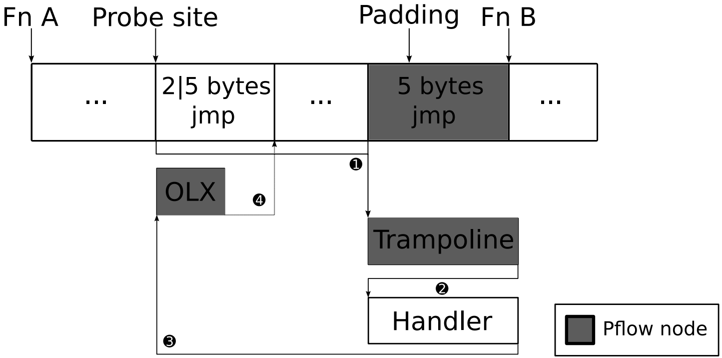
Some definitions (continuation)

Out of Line Execution (OLX) buffer
: Relocated instructions from the probe site. Also called the computation continuation.

Patch
: A patch is a divertion of the current computation, for calling a user defined callback and thereafter executing the patched instructions, before continuing the computation.

Pflow
: A thread is in a patch flow (pflow) from the moment it executes the first jump at the probe site, until it exits the pflow when it executes a branching inside the OLX buffer to the outside.

Patch control flow (pflow)



Summary

Instructions relocation

Problem

Instructions can be relative to the program counter.

- Relative branching.
- Anything with `rip`.

Instructions relocation

Problem

Instructions can be relative to the program counter.

- Relative branching.
- Anything with `rip`.

Solution A

Recompute the displacements.

- Kprobe does that.
- No overhead.
- Memory allocation restriction.

Instructions relocation

Problem

Instructions can be relative to the program counter.

- Relative branching.
- Anything with `rip`.

Solution A

Recompute the displacements.

- Kprobe does that.
- No overhead.
- Memory allocation restriction.

Solution B

Rewrite the instructions.

- Libpatch does that.
- Some overhead.
- No memory allocation restriction.

Instructions relocation examples

Generic case

```
mov DISP(%rip), %rax

;; becomes
push %rbx
movabs $ORIGINAL-RIP, %rbx
mov DISP(%rbx), %rax
pop %rbx

;; can become with register analysis
movabs $ORIGINAL-RIP, %rbx
mov DISP(%rbx), %rax
```

Instructions relocation examples (continuation)

Call

```
call $REL

;; becomes
push %rax
push %rax
;; X = ORIGINAL-RIP + 5
movabs $X, %rax
mov %rax, 0x8(%rsp)
pop %rax
jump *0x0(%rip)

;; can become with register analysis
push %rax
movabs $X, %rax
mov %rax, 0x8(%rsp)
jmp *0x0(%rip)
```

Instructions relocation (continuation)

- Mostly solved problem.
- Register analysis could help reduce overhead.

Trampoline anatomy

```c
struct trampoline_descriptor {
    u8              has_post_probe:1
    u8              k;
    patch_probe     probe;
    void           *user_data;
    uintptr_t       olx_addr;
    uintptr_t       real_pc;
};

/* Not a real C struct! */
struct trampoline {
    u8 instructions[];
    u8 padding[];
    struct trampoline_descriptor desc;
};
```

Trampoline anatomy

```
;; Skip red-zone (leaf functions only)
lea -128(%rsp), %rsp

;; Load trampoline descriptor at %rip + DISP1
push %rdi
lea DISP(%rip), %rdi

;; Call handler stored at %rip + DISP2 in pool
call *DISP2(%rip)

;; Some padding
;; ...

;; Trampoline descriptor (referenced by DISP1)
;; ...
```

Trampoline location

- A trampoline is allocated from a pool.
- Every trampoline is within its own cache line.
- At the end of each pool, handlers addresses are stored.
- Pools are by default around a thousand of pages in size
- Around 64 000 trampolines per pool
- Pools are placed at various strategic places in the program.
  - Before the program.
  - In the heap.
  - Near libraries.
  - Top of stack.

Trampoline location (continuation)

Example of a program with -pie

```
555555164000-555555554000 ... /memfd:libpatch:null-trampoline (deleted)
555555554000-555555555000 ... /home/old/projects/libpatch/a.out
555555555000-555555556000 ... /home/old/projects/libpatch/a.out
555555556000-555555557000 ... /home/old/projects/libpatch/a.out
555555557000-555555558000 ... /home/old/projects/libpatch/a.out
555555558000-555555559000 ... /home/old/projects/libpatch/a.out
555555559000-5555555e3000 ... [heap]
...
7ffff6ded000-7ffff6df2000 ... /home/old/projects/libpatch/a.out
7ffff6df2000-7ffff71e2000 ... /memfd:libpatch:lib-trampoline (deleted)
...
7ffff7fc8000-7ffff7fcc000 ... [vvar]
7ffff7fcc000-7ffff7fce000 ... [vdso]
...
7ffff7fff000-7ffff83ef000 ... /memfd:libpatch:stack-trampoline (deleted)
7ffffffde000-7ffffffff000 ... [stack]
```

Trampoline searching

- Finding a trampoline is hard.
- Find a jump offset from the probe site to a free trampoline.
- The jump offset is actually a pattern of bytes.
- Many algorithms used for more patterns.

Fit algorithm

- Instruction at the probe site is at least 5 bytes.
- Works for around 40% of the instructions.
- Maximum performance.
- 256^4 patterns.

Alias algorithm

- The first bytes of the next instruction(s) become part of the offset.
- Maximum performance.
- Between 1 and 256^3 patterns.

Punning algorithm

- The next instruction(s) are hijacked.
- Traps (or illegal opcodes) have to be placed on basic block entries.
- If there's a indirect branching, assume that all instructions are basic block entries.
- Possibly lower performance if traps are hit.
- Between 23^4 and 256^4 patterns.

NOP padding algorithm

- 2 bytes jump to a padding area between two functions emitted at -O1.
- The padding is transformed into a mini-trampoline.
- Lower performance dues to additional jump.
- 256^4 patterns.

Trampoline searching (continuation)

- Search for trampoline is heavily optimized.
- Could benefit from the unification of the patterns.

A word about W^X protection

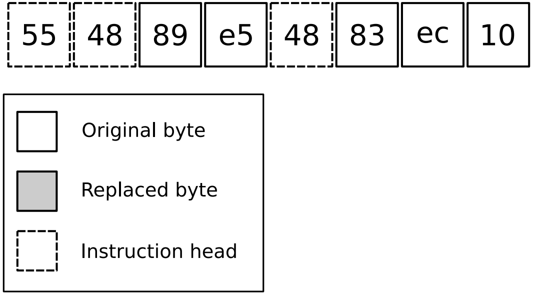
- There is support for write xor execute.
- Uses two virtuals mapping to same physical mapping.
- Breaks GDB and libdw when enabled.
- Not thoroughly tested!

Patching flow

- Punning algorithm was choose.
- There is a membarrier(2) between each step.

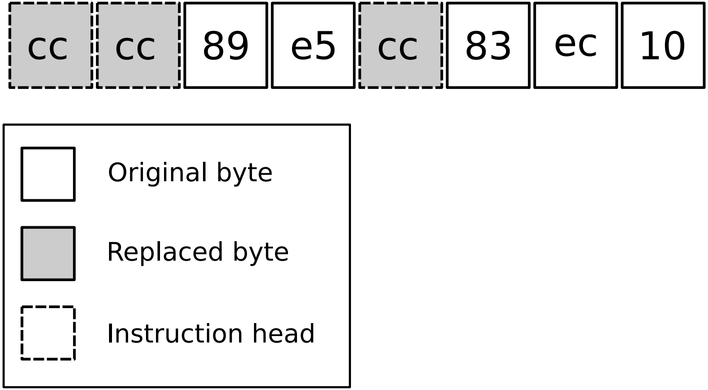
Original instructions

```
push %rbp          ;; 55
mov %rsp %rbp      ;; 48 89 e5
sub $0x10, %rsp    ;; 48 83 ec 10
```



Patching flow (continuation)

Step 1 - Lock patches



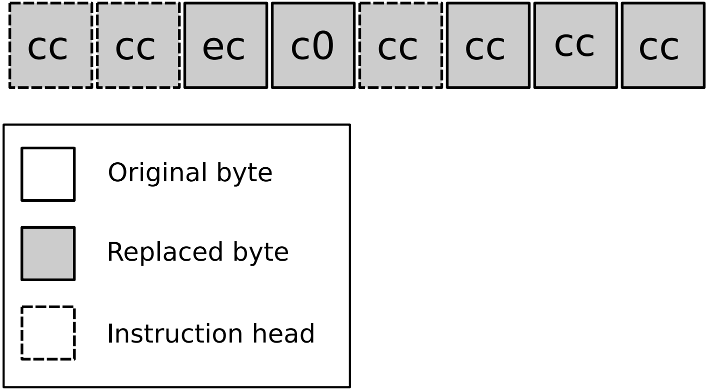
Patching flow (continuation)

Step 2

- Iterate over every thread with PTRACE_SEIZE.
- Check if the thread program counter is in a patching region.
- Check if the thread is in a signal handler and will return to a patching region.
- Move the thread to the corresponding OLX instruction.

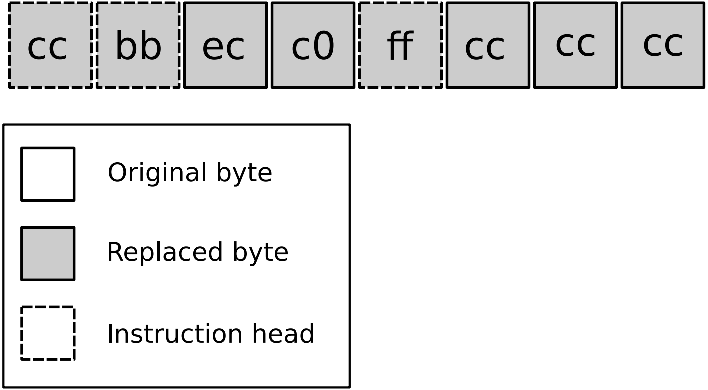
Patching flow (continuation)

Step 3 - Replace bodies



Patching flow (continuation)

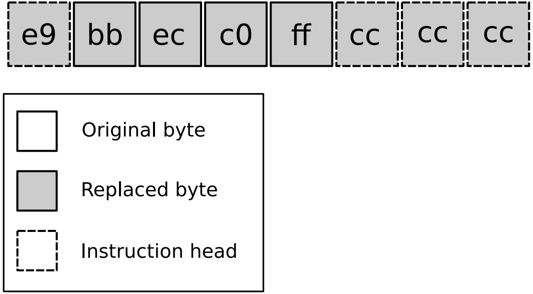
Step 4 - Replace instruction's head



Patching flow (continuation)

Step 5 - Unlock patches

```
jmp 0xffc0ecbb ;; e9 bb ec c0 ff
int3           ;; cc
int3           ;; cc
int3           ;; cc
```



Unpatching flow

- Same steps as patching.
- No need for moving threads out of critical regions.

Patch reclamation

- Restoring the bytes at the probe site is not enough.

Patch reclamation

- Restoring the bytes at the probe site is not enough.
- Any number of threads can already be in a patch flow of a removed patch.

Patch reclamation

- Restoring the bytes at the probe site is not enough.
- Any number of threads can already be in a patch flow of a removed patch.
- How can we reclaim the trampoline(s) and the OLX buffer?

Patch reclamation

- Restoring the bytes at the probe site is not enough.
- Any number of threads can already be in a patch flow of a removed patch.
- How can we reclaim the trampoline(s) and the OLX buffer?
- Libpatch does not use mutex nor reference counter.

Patch reclamation

- Restoring the bytes at the probe site is not enough.
- Any number of threads can already be in a patch flow of a removed patch.
- How can we reclaim the trampoline(s) and the OLX buffer?
- Libpatch does not use mutex nor reference counter.
- We know however that no new thread can enter a removed pflow.

Patch reclamation

- Restoring the bytes at the probe site is not enough.
- Any number of threads can already be in a patch flow of a removed patch.
- How can we reclaim the trampoline(s) and the OLX buffer?
- Libpatch does not use mutex nor reference counter.
- We know however that no new thread can enter a removed pflow.
- A naive solution would be to wait a grace period.

Patch reclamation

- Restoring the bytes at the probe site is not enough.
- Any number of threads can already be in a patch flow of a removed patch.
- How can we reclaim the trampoline(s) and the OLX buffer?
- Libpatch does not use mutex nor reference counter.
- We know however that no new thread can enter a removed pflow.
- A naive solution would be to wait a grace period.
- Libpatch solution is to unwind the thread stack and check for the addresses of the pflow nodes.

Patch reclamation

- Restoring the bytes at the probe site is not enough.
- Any number of threads can already be in a patch flow of a removed patch.
- How can we reclaim the trampoline(s) and the OLX buffer?
- Libpatch does not use mutex nor reference counter.
- We know however that no new thread can enter a removed pflow.
- A naive solution would be to wait a grace period.
- Libpatch solution is to unwind the thread stack and check for the addresses of the pflow nodes.
- Multiple policies.
  - Busy loop until the threads are out of the pflow.
  - Do not free the pflow if any thread is in it. Wait until next garbage collection.

Summary

Coverage

Coverage result with a corpus of 125 binaries from 51 packages for an average of 158871.2 instructions and 1756.56 functions per binary.

Binary	Instruction success rate	Fentry success rate
average	0.988182	0.992196
deviation	0.004909	0.024456
weighted average	0.992819	0.979783
weighted deviation	1.43541e-5	25.0112e-5
geometric average	0.988169	0.991861
geometric deviation	1.00502	1.02705

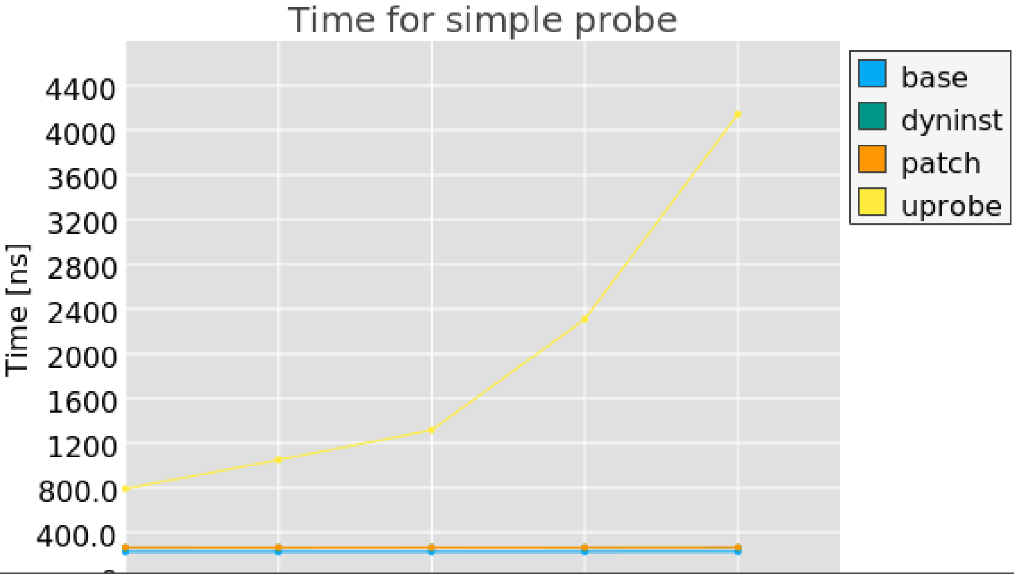
Benchmark

- AMD Ryzen 9 5950X
- SMT (Hyper-threading) disabled
- Frequency boosting disabled
- processor.max_cstate=1
- idle=poll
- Stack depth is more than 128 call frames
- 10^7 loops

Benchmark

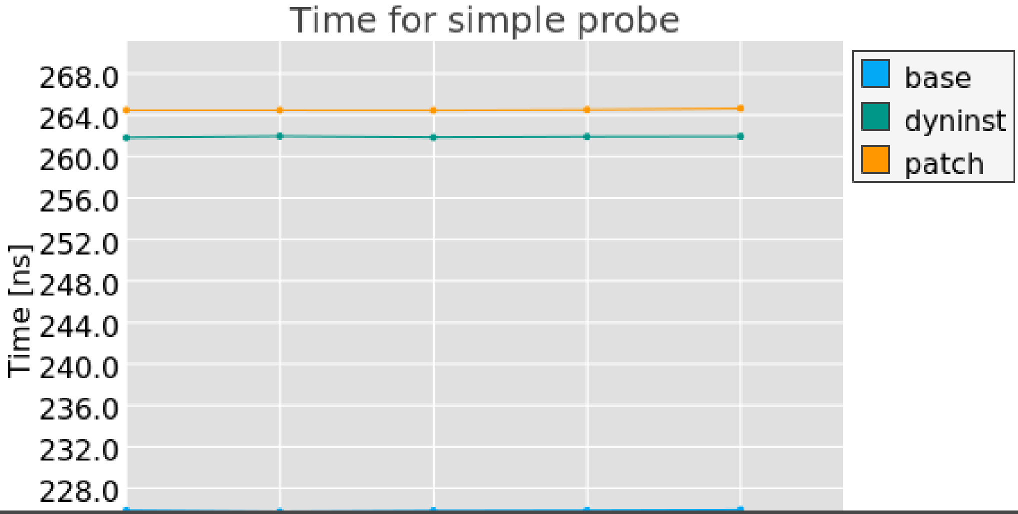
```c
void not_a_leaf(void) { }

uint64_t powmod(uint64_t b, uint64_t e, uint64_t m)
{
        uint64_t c;

        not_a_leaf();

        c = 1;
        for(size_t i=0; i<e; ++i) {
                c = (c * b) % m;
        }

        return c;
}
```

Benchmark (continuation)

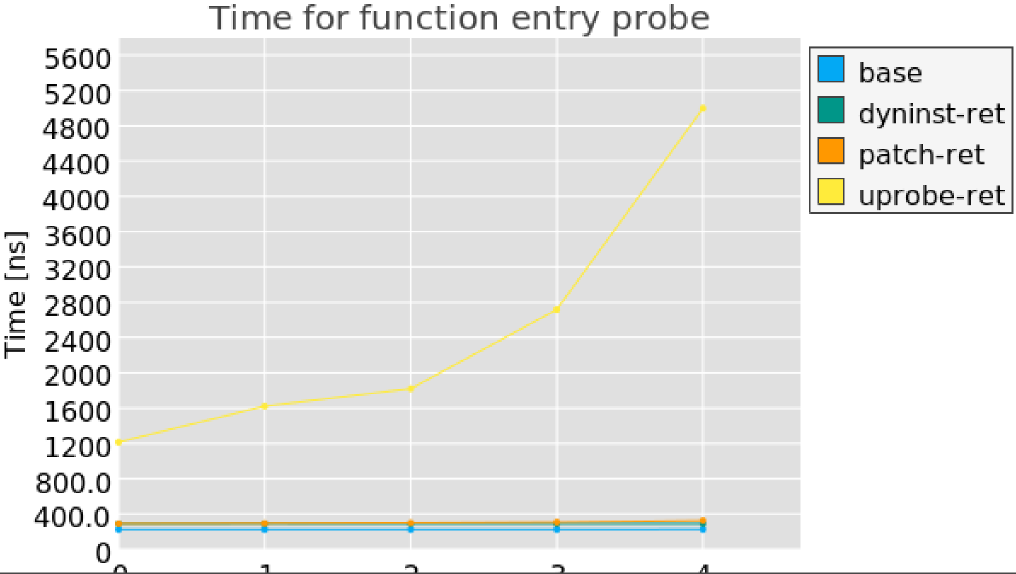


Benchmark (continuation)

Without uprobe

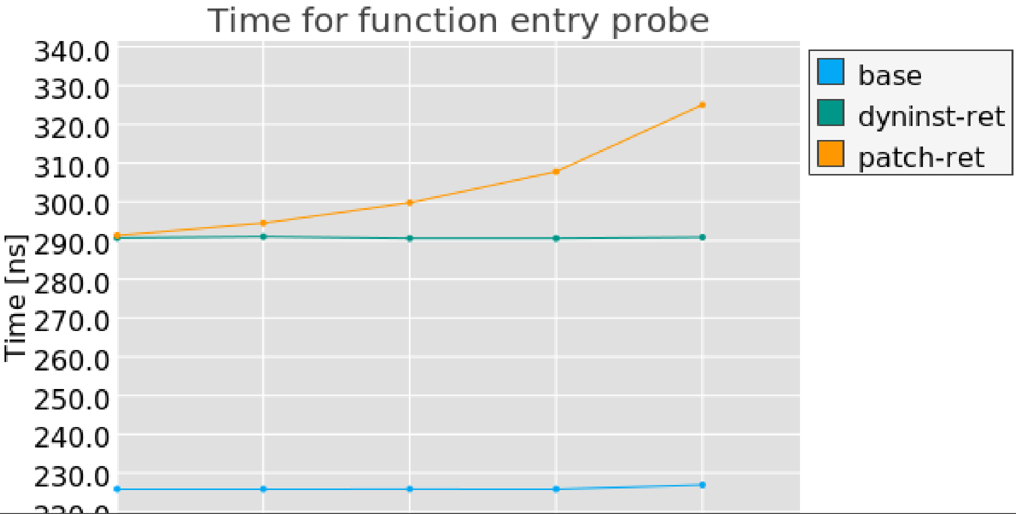


Benchmark (continuation)

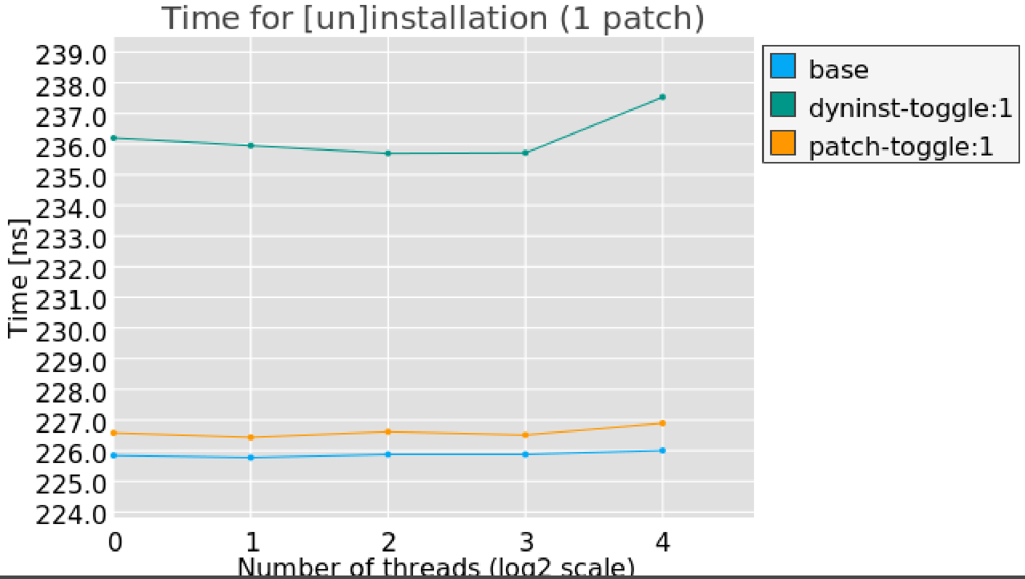


Benchmark (continuation)

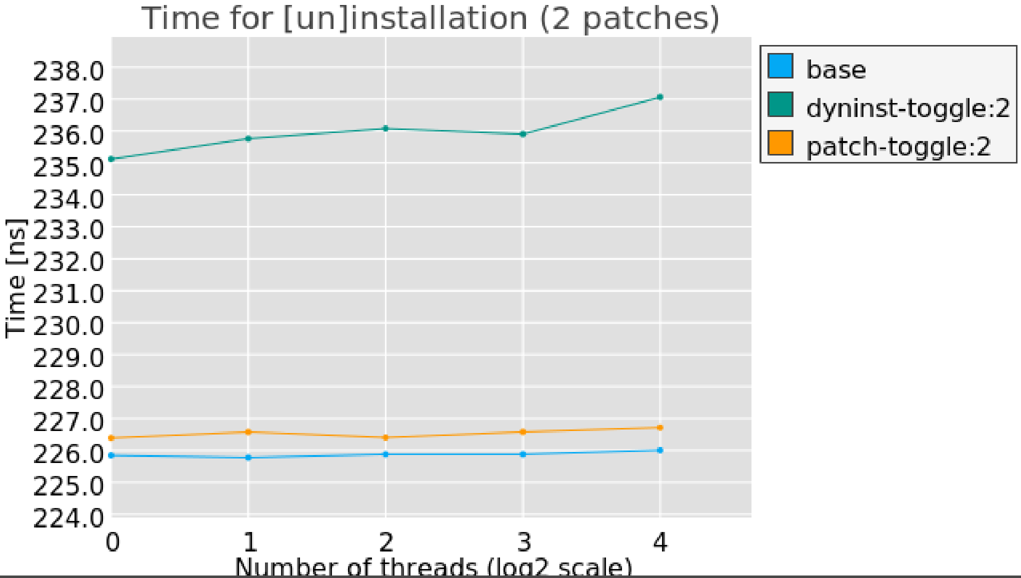
Without uprobe



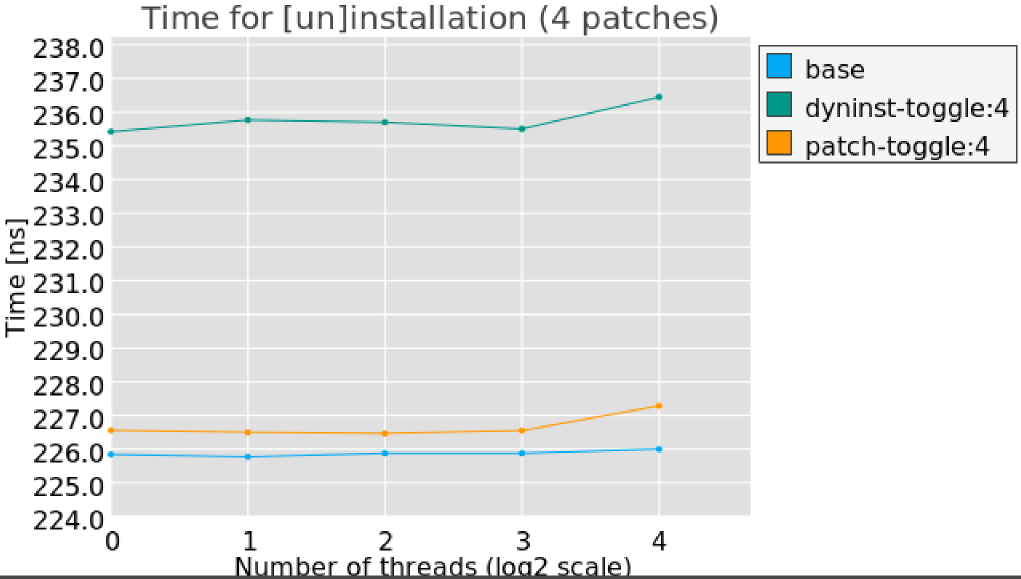
Benchmark (continuation)



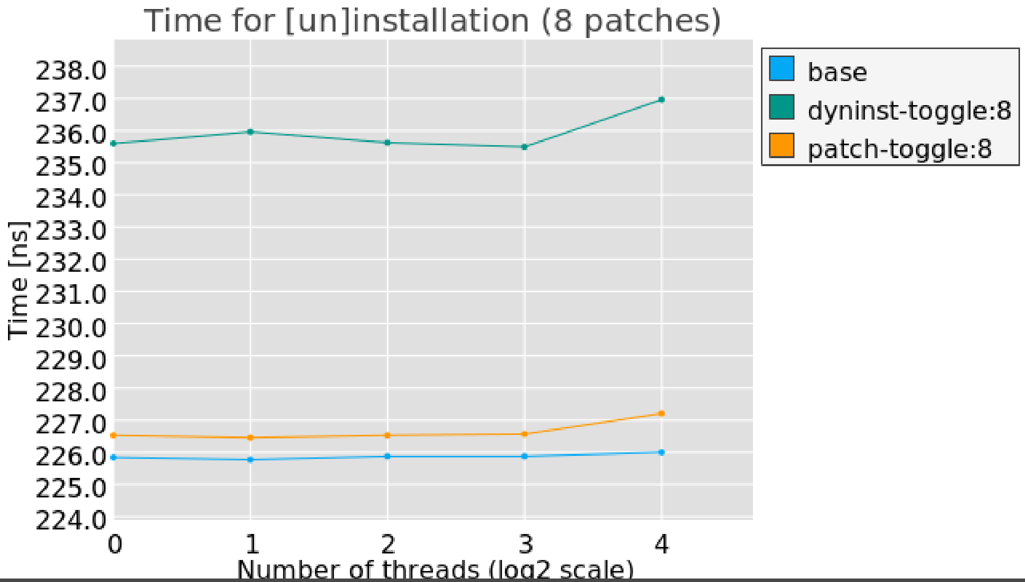
Benchmark (continuation)



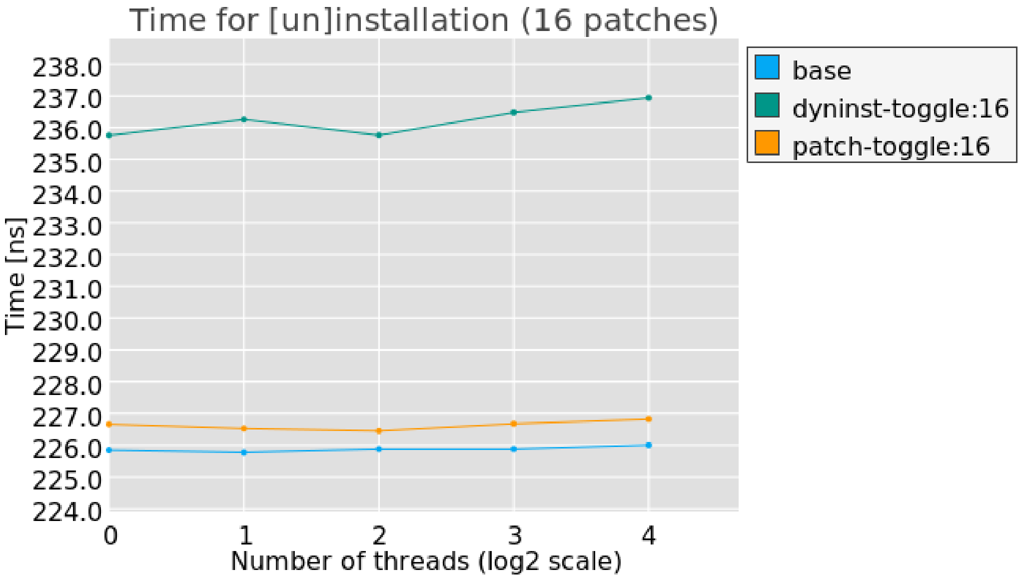
Benchmark (continuation)



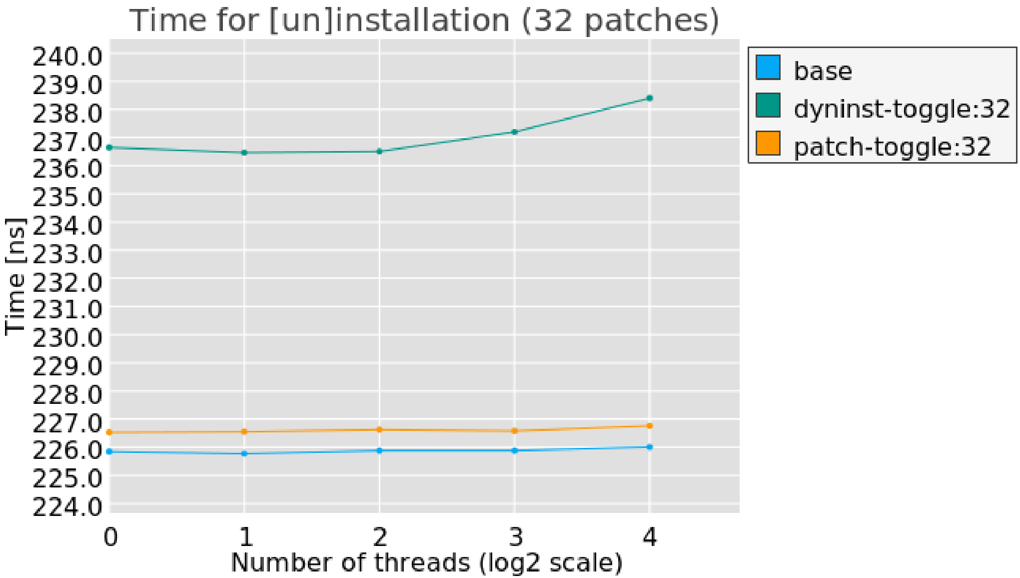
Benchmark (continuation)



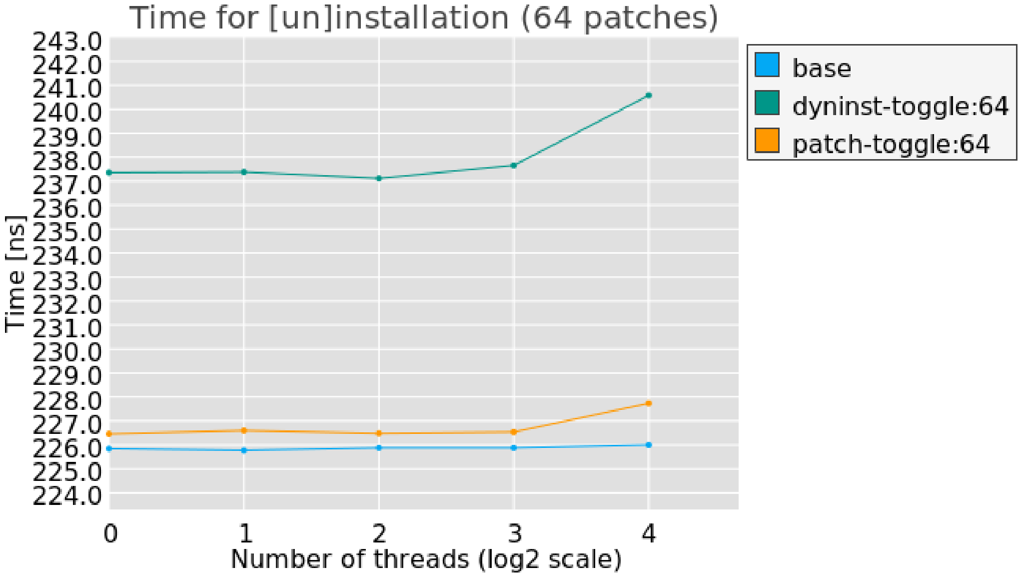
Benchmark (continuation)



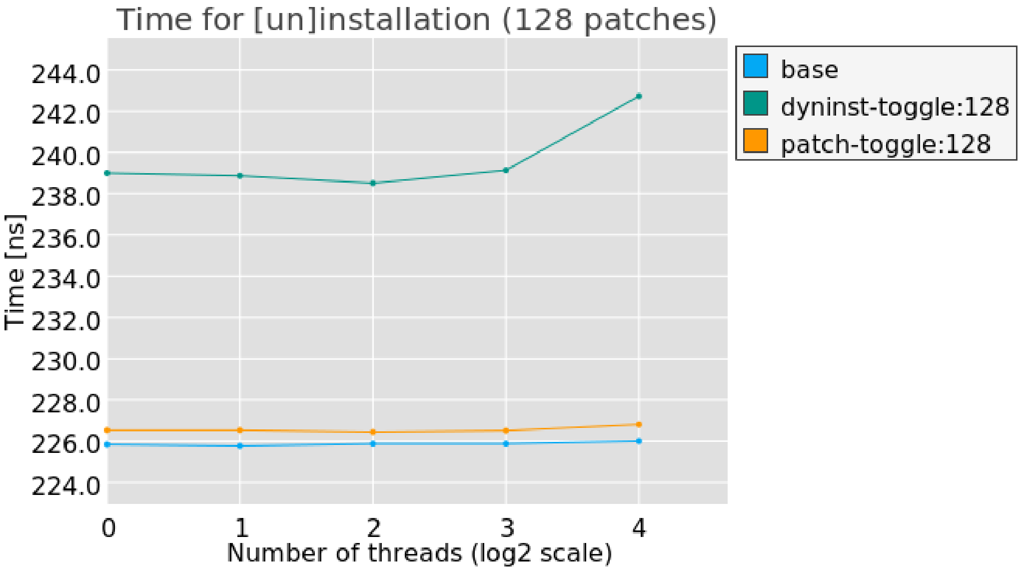
Benchmark (continuation)



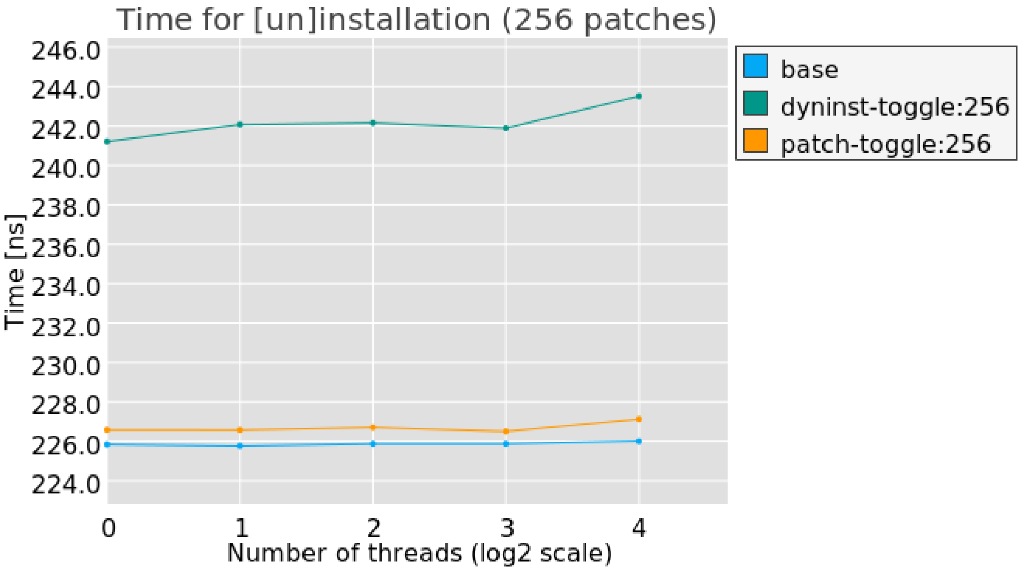
Benchmark (continuation)



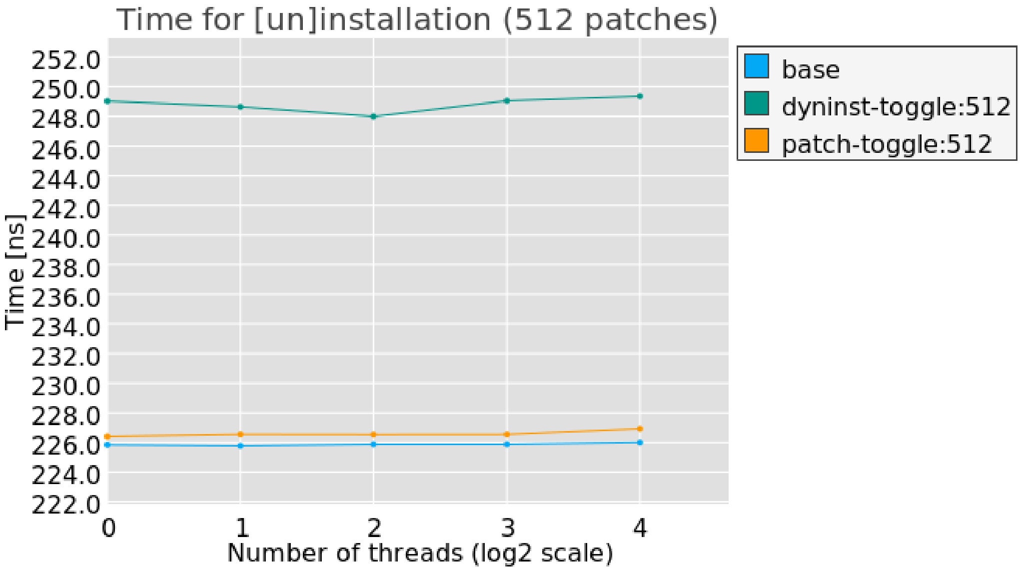
Benchmark (continuation)



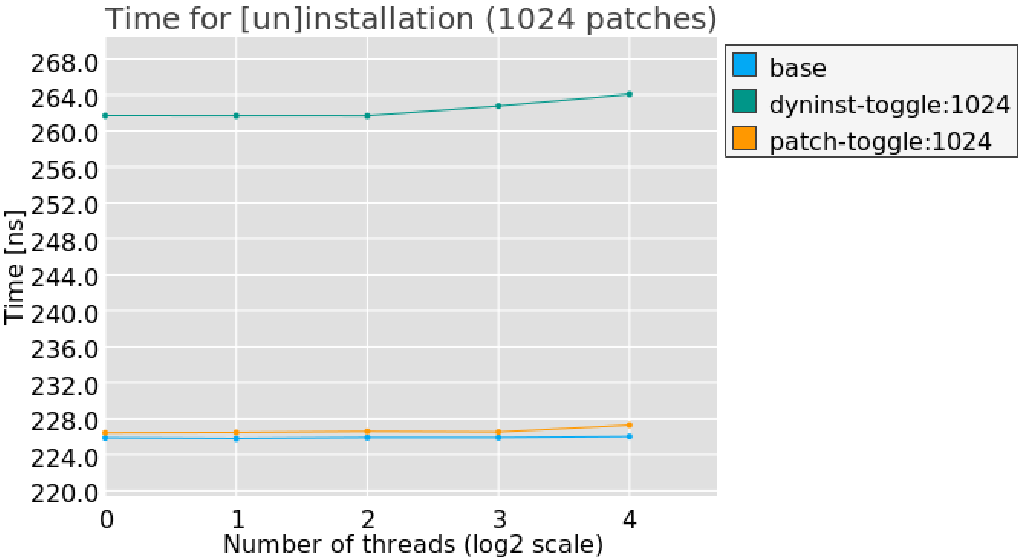
Benchmark (continuation)



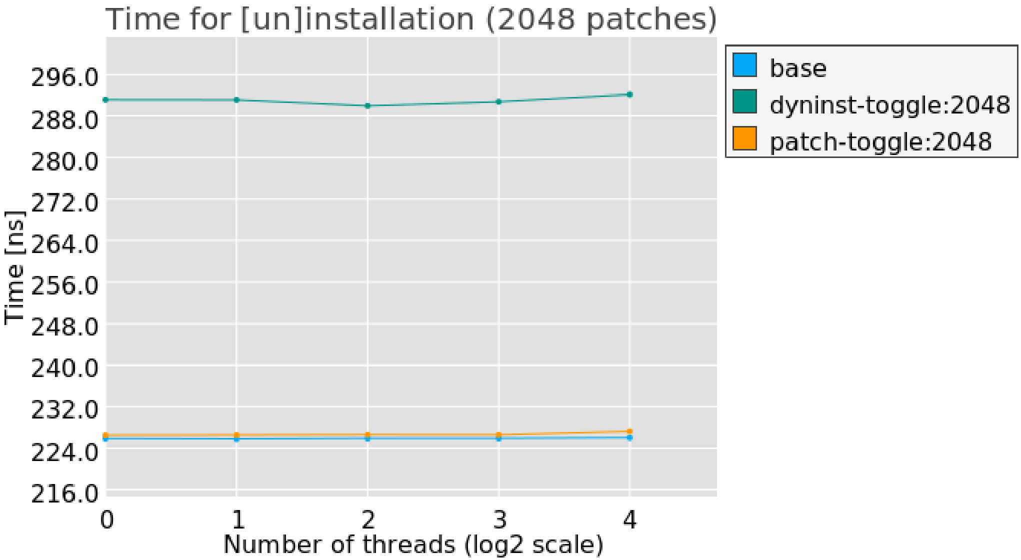
Benchmark (continuation)



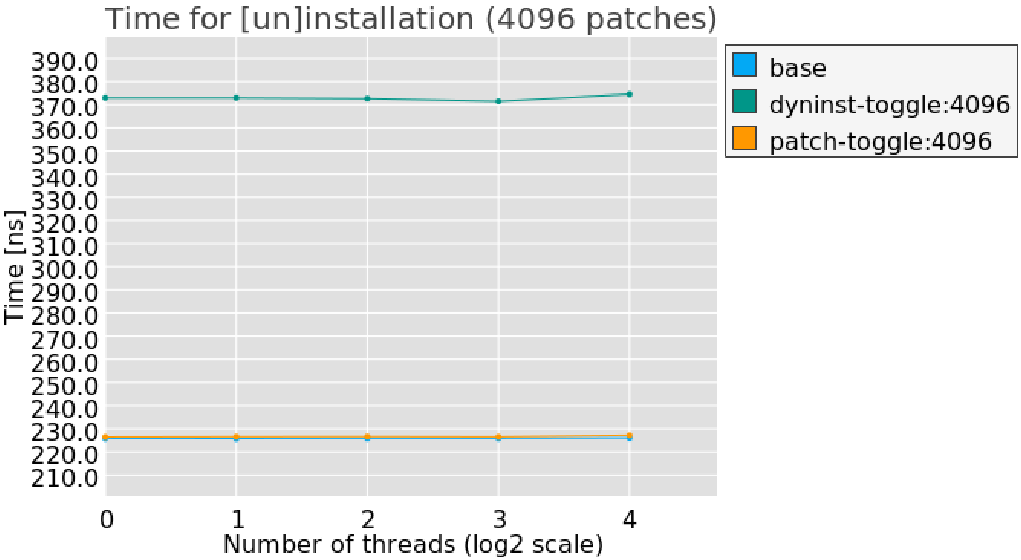
Benchmark (continuation)



Benchmark (continuation)



Benchmark (continuation)



Summary

Discussion

- DWARF basic blocks
  - There is an old patch for gcc.
  - But it never got merged fault of use case.
  - Would make Libpatch less conservative about indirect branches.

Discussion

- DWARF basic blocks
  - There is an old patch for gcc.
  - But it never got merged fault of use case.
  - Would make Libpatch less conservative about indirect branches.
- Need feedbacks for
  - Scenarios where the overhead of the GC is too high.
  - Memory usage.
  - The public API.
  - Error reporting.
  - W^X systems.

Conclusion

- Libpatch is currently only supported for x86-64.
- Support for ARM in the future.
- Almost two 9 for global coverage and already two 9 for function entry.
- Runtime overhead is close to Dyninst without its program analysis time and memory overhead.
- Insertion overhead scales well.

Questions

Questions?