

# UC Irvine

## UC Irvine Previously Published Works

### Title

Polynima: Practical Hybrid Recompilation for Multithreaded Binaries

### Permalink

<https://escholarship.org/uc/item/5jh5q046>

### Authors

Deshpande, Chinmay

Parzefall, Fabian

Hetzelt, Felicitas

et al.

### Publication Date

2024-04-22

### DOI

10.1145/3627703.3650065

### Copyright Information

This work is made available under the terms of a Creative Commons Attribution License, available at <https://creativecommons.org/licenses/by/4.0/>

Peer reviewed



# Polynima - Practical Hybrid Recompilation for Multithreaded Binaries

Chinmay Deshpande  
University of California, Irvine  
Irvine, California, USA  
cddespa@uci.edu

Felicitas Hetzelt  
University of California, Irvine  
Irvine, California, USA  
fhzelt@uci.edu

Fabian Parzefall  
University of California, Irvine  
Irvine, California, USA  
fparzefa@uci.edu

Michael Franz  
University of California, Irvine  
Irvine, California, USA  
franz@uci.edu

## Abstract

The maintenance of software distributed in its binary form can become challenging over time, due to the lack of vendor support or obsolete build environments. This can be costly when dealing with critical security vulnerabilities that are difficult to fix on a binary level. Moreover, advances in compiler technologies of the past decades remain unavailable to the users of such legacy binaries for performing optimizations and transformations. Binary recompilers aim to bridge this divide by “lifting” binary executables to compiler-level intermediate representations (IR) and “lowering” them back again. But, current recompilers fail on that promise as they rely on unsound heuristics or impose high tracing overheads. Crucially, *no existing recompiler* addresses the specific challenges imposed by multithreaded programs that are ubiquitous in the modern software space.

To address these challenges, we present Polynima, a binary recompiler that supports the lifting and recompilation of x86/x64 multithreaded binaries while introducing a moderate 1.23x slowdown. We propose a hybrid control flow recovery approach that combines the benefits of static and dynamic techniques while providing an efficient strategy to handle unknown paths. Polynima enables the use of the rich LLVM compiler ecosystem to fix and improve legacy multithreaded binaries, which we demonstrate by mitigating a critical synchronization issue in a FTP server binary. We also leverage its functional IR to introduce a novel dynamic analysis to detect implicit synchronization primitives in binaries, which we use to further improve performance of the output. Finally, we evaluate the *generality and correctness* of

Polynima by recompiling a diverse set of *real-world, multithreaded* binaries and benchmark suites. To our knowledge, Polynima is the first recompiler to be able to do so.

**CCS Concepts:** • **Software and its engineering** → **Software reverse engineering**; *Dynamic analysis*; • **Theory of computation** → *Concurrency*;

**Keywords:** binary analysis, binary lifting, recompilation, multithreaded programs

## ACM Reference Format:

Chinmay Deshpande, Fabian Parzefall, Felicitas Hetzelt, and Michael Franz. 2024. Polynima - Practical Hybrid Recompilation for Multithreaded Binaries. In *Nineteenth European Conference on Computer Systems (EuroSys '24)*, April 22–25, 2024, Athens, Greece. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3627703.3650065>

## 1 Introduction

The maintenance of software distributed as binary executables becomes challenging over time. Due to obsolete build environments, unavailability of the original source code or the lack of vendor support, recompiling legacy programs from source may not be possible. This denies the users of such programs the substantial advances in modern compiler technologies, like those relating to compile-time optimizations and security hardening. More importantly, the lack of access to the compiler ecosystem can be costly when dealing with critical security vulnerabilities that are difficult to fix at a binary level. At the same time, replacing legacy software can be very expensive and often infeasible.

Binary recompilation is a rewriting technique that enables the use of the analysis and transformation infrastructure of a compiler, by lifting machine code to a compiler-level intermediate representation (IR). State-of-the-art recompilers [3, 5, 15, 16, 42] target LLVM IR [28] due to its active community, modular design and the rich tooling support. But, most of these tools have not seen widespread adoption in practice due to their inability to handle complex binaries and the use of unsound heuristics [29, 39].

Moreover, with the ubiquity of multicore processors, programs are designed to fully leverage the benefits of the un-



This work is licensed under a Creative Commons Attribution International 4.0 License.

*EuroSys '24, April 22–25, 2024, Athens, Greece*

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0437-6/24/04

<https://doi.org/10.1145/3627703.3650065>

derlying hardware. This is typically achieved through multithreading, which involves distributing work among parallelly executing program contexts known as threads. But, multithreaded software is often plagued with an entirely new class of issues, such as those relating to resource synchronization, making their maintenance even harder. Unfortunately, *none* of the state-of-the-art recompilers address the specific challenges involved in the recompilation of multithreaded binaries.

Existing recompilers are either *entirely* static [5, 15, 16, 42] or dynamic [3] in their approach toward control flow recovery. Static disassembly is fast but employs heuristics to predict targets of indirect control transfers which can be imprecise [35]. Imprecision is acceptable for performing analyses on top of the IR but program-wide transformations require correct relocation of all code pointers, which can be hard to perform statically. Dynamic recompilers [3], on the other hand, analyze concrete executions of the target program, enabling them to handle disassembly and indirect control flows by design. However, their approach is inefficient due to the high tracing overhead required to generate an output that supports a sufficient subset of the original binary’s functionality. Current techniques also fail to implement effective strategies to handle novel control flows that may be realised during execution of the recompiled binary.

After lifting to IR, the compiler is free to reorder shared memory accesses, which may lead to erroneous and divergent program outcomes that break original program semantics [38]. Recent work shows that existing approaches that insert memory fences to prevent such reorderings may be overly conservative and therefore impede IR-level optimizations in some cases [8]. Besides, recompilers must also correctly handle constructs such as hardware-supported atomic instructions, callback functions and most importantly, the per-thread program stack.

We present Polynima, the first binary recompiler that supports the general lifting and recompilation of a wide range of multithreaded x86/x64 binaries. Polynima enables the use of LLVM’s mature analysis and transformation ecosystem by recovering, and precisely representing, original program semantics in the lifted IR. We generate standalone replacements of input binaries while introducing a moderate 1.23x average slowdown in performance.

We propose a novel technique to perform control flow recovery using a hybrid approach that combines the efficiency of static analyses with the precision of dynamic analyses. To reduce the tracing overhead imposed by dynamic lifting, Polynima initially performs a fast static lift using a commercial-off-the-shelf (COTS) disassembler. As this information can be imprecise, we provide a lightweight tracer module that enables resolving targets of indirect control transfers upfront. Finally, to efficiently recover from unknown transfers seen during recompiled binary execution, we implement a control flow miss handling strategy that we call *additive lift-*

*ing*. Additive lifting iteratively reconstructs the control-flow graph (CFG) by dynamically integrating newly discovered control targets.

To address the unique challenges imposed during the lifting of multithreaded binaries, Polynima implements techniques to correctly handle crucial constructs such as callback functions, hardware atomic instructions and the per-thread program stack. Next, we present an innovative strategy to detect implicit synchronization primitives in binaries, which we use to remove superfluous fences inserted in the IR for preserving memory access ordering. We implement this, and other *refinements*, through dynamic analysis based instrumentation that leverages Polynima’s functional IR and the low performance overhead of the recompiled output.

Finally, we evaluate Polynima’s recompilation capabilities against a diverse set of complex multithreaded binaries that includes real-world utilities and benchmark suites. We show a compelling use-case where our prototype enables us to retrofit protections against CVE-2023-24042 in a LightFTP binary by leveraging LLVM’s transformation infrastructure. We also evaluate the efficacy of our fence removal optimization and show that it can notably improve output performance by enabling further off-the-shelf compiler optimizations. To summarize our contributions:

- We present Polynima, the *first practical* binary recompiler for multithreaded x86/x64 binaries that introduces a moderate 1.23x slowdown. We implement a hybrid control flow recovery approach that combines the benefits of static and dynamic techniques while also providing an efficient miss handling strategy.
- Polynima supports a wide range of real-world utilities and benchmark suites, demonstrating its ability to *correctly* lift and recompile complex multithreaded machine code. Crucially, we demonstrate that Polynima makes available the transformation ecosystem of the compiler to perform modifications to multithreaded binary programs.
- We leverage the functional IR to design an innovative dynamic analysis for detecting implicit synchronization primitives in binaries, that we use to remove superfluous fences inserted to prevent memory reorderings.

## 2 Related Work and Current Limitations

### 2.1 Control Flow Recovery

Precise recovery of the complete CFG is necessary for correct recompilation. Failure to do so may lead to imprecise code pointer relocations and, hence, run time crashes in the recompiled binary while realizing such paths. This is hard [22] because compilers do not attach any labels to differentiate between code and data pointers in machine code. Further, control flow between basic blocks may use indirect jump and call transfers whose targets are difficult to infer statically.

Static approaches to disassembly and control flow recov-

ery try to identify function entry points and then employ a recursive descent approach for exploring reachable machine code. Recompilers like McSema [16] and Rev.Ng [15] often use a combination of heuristics and program analyses, such as value-set analysis (VSA) [6], to predict an over-approximate set of targets of indirect control transfers. These heuristics achieve high accuracy for instructions that load their destination address from jump tables [14, 43], but resolving other indirect transfers remains a challenge. For instance, IDA Pro [23], which McSema relies on for recovering control flow information, performs poorly when recovering targets of indirect calls [35]. mctoll [42] employs heuristics to identify jump tables and resolve a subset of indirect intra-function control transfers. It cannot identify possible targets of indirect calls precisely, either. Although fast, static approaches to lift binaries are forced to balance accuracy and coverage. They produce programs that suffer from incomplete code discovery and generate code that fails to replicate the original’s semantics.

BinRec [3] is a dynamic recompiler that handles precise code and control flow recovery by design as it records information about paths realized during concrete execution runs of the input program. As enabling deobfuscation of input programs is one of their goals, BinRec induces a tight coupling between CFG recovery and the IR translator module as a part of their lifting frontend. This imposes a notable lifting overhead due to the time required for setting up and tracing the input binary inside a processor emulator-like execution environment.

A control flow miss occurs when, during its execution, the recompiled binary attempts to perform a control flow transfer to an address that was not discovered during lifting. Such misses are frequently triggered by non-deterministic program behaviors [3], which are particularly common in multithreaded code. None of the existing static recompilers implement support for such control flow misses. As achieving full coverage of the input binary for a dynamic recompiler is hard, BinRec proposes *incremental lifting*. When a control flow miss occurs, BinRec terminates execution of the running program and initiates a new trace of the original binary using their lifting frontend. To keep the tracing overhead manageable, this trace is started at the newly discovered address instead of the beginning of the program. Since the incremental trace is started at an arbitrary address, it is prone to trigger runtime faults before discovering any new targets due to uninitialized stack and heap memory. To mitigate this, BinRec performs path exploration only until the next conditional control flow which is inefficient.

Also, one of the core promises of recompilation is cross-ISA (instruction set architecture) translation which is valuable for programs compiled for legacy architectures. But, current dynamic approaches rely on access to the original execution environment or an emulator to recover the CFG, which is not guaranteed.

## 2.2 Lifting to Emulated IR

**2.2.1 Handling stack memory.** Recompilers typically implement lifting by performing line-by-line translation of machine code to LLVM IR. The lifted IR aims to faithfully *emulate* the execution of each machine instruction on a virtual CPU state that consists of registers, flags, and stack memory. As a result, each low-level instruction may map to several LLVM IR statements, leading to a verbose and unrefined IR. Although compiler-level optimizations are sufficient to optimize away some dead code, designing techniques to refine the lifted IR is a topic of active research [17, 29, 36].

Stack memory is critical to program execution as it stores function-local variables, spilled register values, and arguments for calls. Recompiled programs usually work with two stacks of execution, (1) the *native* stack, that contains variables and spills that are a byproduct of the emulation, (2) the *emulated* stack, that includes variables and spills of the input program. Recompilers such as McSema, BinRec, and Rev.Ng model the emulated stack as a global array of bytes. However, their implementation is not general as they do handle the multithreaded case where each thread of execution needs to work with its own emulated stack.

Using an emulated stack hinders transformations [29] that rely on dataflow analysis of values in memory, as the compiler treats all accesses to the emulated stack opaquely. To mitigate this, some recompilers split the emulated stack into individual chunks and move them to the lifted program’s native stack. For instance, mctoll performs static analysis to identify the maximum bound on the per-function stack frame size and creates a (function-) local allocation that represents the original program’s stack frame. But, this approach is not general as previous work has shown that statically inferring the maximum frame size of functions in binaries is hard [13]. In fact, the frame may not be bounded at all for programs that call `alloca` (or one of its many variants) with a dynamically determined size argument or use Variable Length Arrays (VLAs). Insufficiently allocated stack frames could lead to runtime faults after recompilation due to out-of-bounds frame local accesses reaching into unknown memory.

Moreover, this optimization of recovering the per-function stack frame relies on precisely identifying and translating *all* accesses to the local stack frame in the original program. Statically performing this procedure is largely heuristics-driven, as identifying if any stack reference escapes is undecidable in the general case. This issue is especially critical for lifting multithreaded binaries as imprecision in identifying stack-exclusive accesses may lead to erroneous and unsynchronized shared memory writes. Due to the lack of generality of this approach, previous work [38] that builds on mctoll could not evaluate specific binaries from the Phoenix benchmark suite [37].

**2.2.2 Hardware atomic instructions.** Multithreaded binaries may leverage program constructs, such as those pro-

vided by the programming language, compiler, or the underlying hardware, that pose new challenges for recompilation. Atomic instructions are critical for implementing synchronization primitives, such as locks and semaphores, in machine code. On the x86/x64 hardware, this includes read-modify-write (RMW) operations (e.g., `lock add`, `lock inc`) and compare-exchange operations (e.g., `lock cmpxchg`). Atomic memory accesses on the x86/x64 architecture assert that all observers see the access as having happened or not happened at all, and never partially happened [24]. The executing thread has exclusive ownership of the data for the duration of the instruction to ensure that no partial state is ever exposed to other observers on the system.

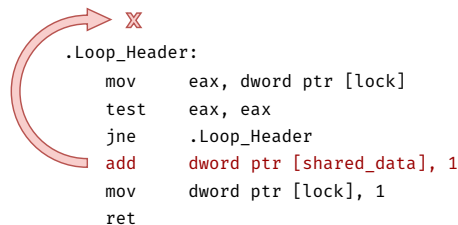
Precise lifting of such instructions to LLVM IR is hard for multiple reasons. Consider the translation of the `lock cmpxchg dword ptr [rsi], ecx` instruction as a representative example. This instruction compares the value in the `eax` register with the value stored in the destination operand i.e., memory pointed to by `rsi`. If the two values are equal, the second operand is loaded into the destination, else the destination operand is loaded into `eax`. The instruction also updates the zero bit of the `EFLAGS` register depending on the result of the equality. Note that all of the sub-operations are executed as part of the same hardware instruction. However, the programming model and the set of available abstractions in lifted LLVM IR differ notably from that assumed by the underlying hardware. For instance, it is not possible to represent the update to the (virtual) `eax` register and the compare-exchange as part of an indivisible IR instruction.

Of all the recompilers, only McSema supports the *translation* (binary to IR) of hardware atomic instructions to LLVM IR by using the appropriate compiler intrinsics. Unfortunately, its authors conveyed to us that its recompilation capabilities (binary to IR to binary) are experimental and need an expert operator to fix issues manually.

**2.2.3 Callback functions.** Correct handling of callback functions is crucial to support multithreaded binaries that use lightweight processes as a threading mechanism (as opposed to user-level threads). This is because the underlying interface of `clone` which is used to spawn threads on Linux requires an entry point for the new execution context. These are considered to be *external* entry points, as the control flows in to the binary program from external library code. Full-program recompilation forces code layout changes, making existing code pointers invalid. To ensure correct support for external entry points, recompilers need to precisely identify and rewrite *all* instances of function pointers passed to external procedures. They must also implement the sound handling of the execution context switch from external library code to the recompiled binary, and back.

RevNg supports external entry points through static linking of libraries and treating them as indirect calls. However, analyzing and rewriting statically-linked code is generally

```
std::atomic<bool> lock;
void thread_func2() {
    while (lock.load(std::memory_order_acquire));
    shared_data += 1;
    lock.store(true, std::memory_order_release);
}
```



```
.Loop_Header:
    mov     eax, dword ptr [lock]
    test   eax, eax
    jne    .Loop_Header
    add    dword ptr [shared_data], 1
    mov    dword ptr [lock], 1
    ret
```

**Figure 1.** Memory access reordering at the IR-level may lead to erroneous outcomes. The write to `shared_data` may be reordered across the critical section during recompilation due to the lack of original ordering semantics.

infeasible, as it incurs a substantial performance overhead and does not scale. McSema and mctoll try to statically identify function pointer arguments passed to external functions to rewrite them. But, tracking pointer values in machine code, especially if they are passed across function boundaries, can be hard. Also, it may be impossible to precisely solve this problem statically if pointer values are materialized in registers or memory during execution.

BinRec does not attempt to rewrite function pointer arguments and instead inserts trampolines at the original address of function entry points. The trampolines divert control to helpers that marshal native state into emulated state, execute the lifted code and then translate the emulated state back to native before jumping back into external library code. Although this approach is sound, BinRec does not handle the case where the entry point may be executing as part of a different thread. Specifically, it does not correctly initialize the virtual CPU state and the thread-local emulated program stack on entry which may cause faults at run time.

**2.2.4 Memory access reordering.** Information about the relative ordering of memory accesses, which may be specified implicitly using synchronization barriers or explicitly using source annotations, is lost during program compilation. Hence, the lifted IR obtained from such binaries contains no ordering information. Failure to preserve the original program ordering may lead to erroneous and divergent program outcomes due to the compiler reordering shared memory accesses at the IR-level.

Consider the example shown in Figure 1 which represents a shared memory access that is synchronized using a spinlock. The source-level memory ordering semantics attached to the *acquire* load and the *release* store of `lock` assert that write to `shared_data` will not be reordered across the criti-

cal section. However, the generated machine code implicitly encodes these semantics due to the lossy nature of the compilation process and the guarantees provided by the underlying x86/x64 ISA. For instance, (1) as naturally aligned stores and loads upto 64 bits are guaranteed to be atomic, the compiler emits an ordinary `mov` for the load operation, (2) the strong Total Store Ordering (TSO) model prevents the Store-Store reordering between the `add` and the `mov`. But, after lifting, the compiler is free to reorder these memory accesses in the IR (such as in the case of `shared_data`) which may break original program semantics.

To remedy this issue, Lasagne [38] formalizes the idea of the LLVM IR Concurrency Model (LIMM) and discusses a sound strategy to lift memory accesses in multithreaded binaries to LLVM IR. They insert appropriate fences for each memory access preventing the compiler from reordering them. As fences can be costly for performance and hinder off-the-shelf optimizations, they also propose optimizations that remove fences for stack-exclusive accesses and merge adjacent (redundant) fences.

Recent work [8] has shown that Lasagne’s fence placement strategy may impose stricter restrictions than necessary for specific programs, incurring a high performance cost for recompiled binaries. Lasagne primarily targets cross-ISA translation to ARM64 (a weaker memory model), which requires that the IR impose the strict x86/x64 memory model for all memory accesses, except those that target the thread-local stack. But, when recompiling binaries for the same architecture, almost all inserted fences are superfluous for programs that synchronize shared memory accesses through exclusive use of externally provided barriers and primitives (e.g., those provided by the `pthread` library). This occurrence is common, as correctly implementing custom primitives is hard and programmers often rely on third-party libraries to achieve this. In fact, all programs in the Phoenix benchmark suite exhibit this property.

### 3 Design and Implementation

Polynima is a full-transformation recompiler consisting of modules that perform control flow recovery, translation of machine code to LLVM IR, optimization and lowering. Figure 2 gives an overview of the system architecture. Recompiled output generated through static-only analyses acts as a functional replacement for the input binary. Although this initial output representation only supports control flows that are recovered through the COTS disassembler, we instrument the lifted IR to handle unknown transfers at runtime. Our optional dynamic analyses, such as those for optimizing the lifted IR, build on top of this representation.

#### 3.1 Compatibility

Our prototype supports a wide range of binaries, but we impose certain reasonable restrictions on the input for imple-

mentation reasons. We support the recompilation of x86/x64 Linux-based C and C++ binaries for their original architectures. We assume that the program stack grows downwards and that the stack pointer register (`esp/rsp`) points to the top of the thread-local stack.

We operate on inputs without relocation information. This is typical of most legacy binaries which are primary application targets for Polynima. To handle code and data pointer relocations, we map the input binary at its original load address as part of the output. Therefore, the output contains the original binary code in addition to the recompiled lifted code.

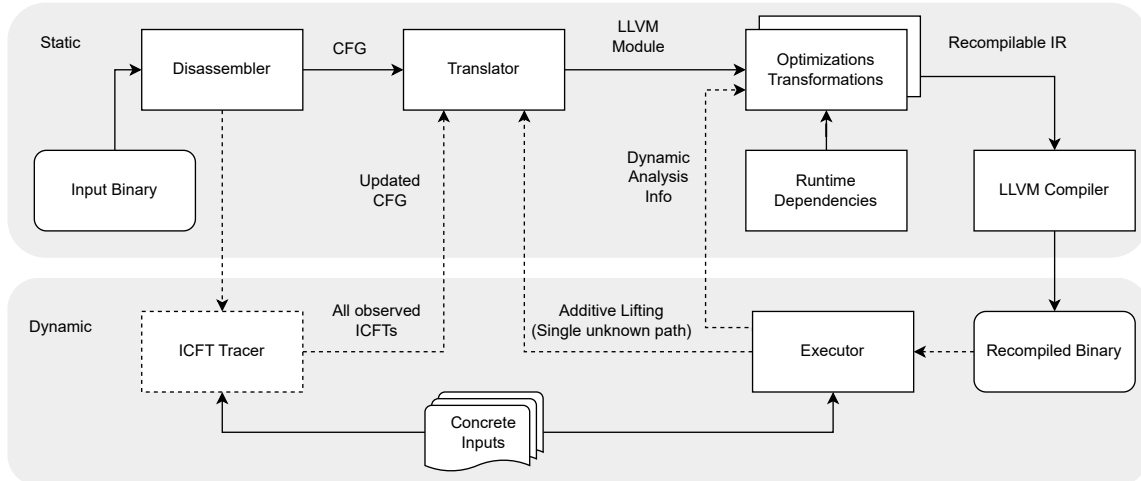
Binaries may use threading models and synchronization primitives as exposed by POSIX threads (`pthread`), C11 (`thread.h`), C++11 (`std::thread`) and OpenMP programming models. Supported programs may also implement custom primitives provided by functions from any of the above interfaces, such as C11 (`atomic.h`) or C++11 (`std::atomic`). We also handle compiler builtins, such as the `__sync_` variants, that typically lower to use hardware atomic instructions.

Linux threads, in this case defined as lightweight processes, are spawned by calling the `clone` system call which is wrapped by library functions such as `pthread_create` or `thrd_create`. Polynima supports external library calls with unknown interfaces through *stack-switching*, where the native stack pointer points to the emulated stack for the duration of the call. However, in the case that the call enters a new thread context, it would work with its own thread-local stack. Implementing stack-switching in such a scenario would involve dealing with four stacks of execution, making the implementation overly complex. For that reason, we require the knowledge of signatures for library functions that spawn threads, such that we can lift them to execute in the context of the native stack. We do not support user-level threads that can be achieved through `get/setcontext()`, `make/swapcontext()`, and `long/setjmp()`.

We do not handle lifting of the `syscall` instruction. This occurrence is rare, as portable software usually relies on the existence of native shared libraries (such as `glibc`) on the target system to interact with the kernel. We currently do not support binaries with self-modifying code. Additive lifting enables us to recompile binaries with overlapping instructions and obfuscated control flow by design, but we do not evaluate our prototype on that capability barring a hand-written example. We also assume that the underlying memory model does not imply the support of precise exceptions as it requires that the recompiled binary preserve semantics for instruction rollback when interrupted by the CPU.

#### 3.2 Control Flow Recovery

For the initial lift, Polynima consumes information about function entry points, the basic blocks belonging to them,



**Figure 2.** Overview of Polynima. Dashed lines indicate optional steps.

and the direct control transfers between identified basic blocks from a COTS disassembler. We treat jump and call instructions as basic block terminators and explicitly label control transfers as *jump*-based or *call*-based in the CFG. Basic blocks are labeled as *direct* if the terminator instruction encodes the transfer’s target address, and *indirect* otherwise.

For indirect control transfers, we assume a set of known targets and lift them as switch statements, that select their target based on the current value of the emulated program counter (PC). Each switch case represents a possible value that the PC could assume in the original program, and is mapped to the corresponding lifted block in the IR. But, obtaining a set of possible targets for an indirect control transfer is a hard problem. Polynima thus implements a hybrid approach that can use static as well as dynamic analysis results. We currently support three distinct ways to achieve this.

**Static.** Modern disassemblers implement various heuristics to resolve jump tables and infer targets of indirect calls. Polynima *uses but does not expect* disassembler-provided targets for indirect jumps and calls, benefitting from advances in static CFG recovery. As the control flow is conditioned on the actual PC value at runtime, Polynima can also graciously handle incorrectly predicted targets. However, as statically collected information can be imprecise, we may observe previously unknown control flows during the execution of the recompiled binary.

**Additive.** To support dynamically discovered targets, Polynima implements additive lifting. We achieve this by instrumenting the terminating switch statements of all indirect blocks to jump to a custom runtime after encountering an unknown PC value. On encountering a new path, the runtime updates the on-disk representation of the CFG with this information and then stops program execution.

Starting at this target, we perform a static recursive descent style exploration of the original binary control flow

and integrate back all the discovered paths into the known CFG. This technique is useful for jump-table style control transfers where the paths from the newly discovered block eventually join with the rest of the known CFG through direct transfers. We then rerun the recompilation pipeline to generate a new binary that supports the additional paths. The entire process can be thought of as a recompilation *loop*, with each intermediate output supporting statically known and dynamically discovered control flow. Discovering new paths by natively executing the recompiled output is an efficient and complementary strategy to static CFG recovery techniques for handling control flow misses.

Crucially, additive lifting enables *on-device* lifting. This can be particularly useful for recompiling legacy binary programs without access to their original execution environment or a suitable emulator. Users can statically generate a fully functional recompiled output, that supports known control transfers, for their target environment through Polynima. This is possible as the recompilation process enables the linking of new libraries, patching unsupported instructions and compiling for a different ISA. With the newly gained ability to natively run the program on the target architecture, unknown paths can be additively recovered during program execution.

**Dynamic.** We note that the performance of the above approach is directly proportional to the total time required for each recompilation run. This can be inefficient when, (1) the time required for an individual lift-and-lower step is high, such as in the case of large binaries, (2) unseen control flows are observed a long time from execution start.

To resolve this, we provide an optional and low-overhead Indirect Control Flow Target (ICFT) tracer that can be used upfront to augment the statically recovered CFG. Given a set of inputs, it observes concrete executions of the program and records all targets of indirect control transfers. It then merges

information recorded across the different runs, providing the benefits of an entirely dynamic recompiler.

Note that additive lifting complements the ICFT tracer module. Non-deterministic behaviors may lead to certain program paths never being exercised even after extensive tracing, which necessitates sound handling of the unknown control flows in the recompiled binary. In fact, such behaviors are particularly common in multithreaded machine code due to the various thread interleavings that are possible at runtime. Recent work also observed such behaviors in binaries from the SPEC benchmark suite where pointers were being used as keys into hashmaps [36].

### 3.3 Multithreading Support

**3.3.1 Atomic Instructions.** Support for hardware-provided atomic instructions is necessary to generally handle multithreaded machine code. A naive approach to their translation is to decompose them into distinct loads and stores, with all the accesses synchronized using a global (spin)lock. This maintains all the guarantees in terms of exclusive access to memory and the ordering of accesses. But, a major drawback is that *all threads* executing an atomic instruction, irrespective of whether the referenced memory locations alias, have to (spin)wait.

To optimize this, we map atomic instructions to the appropriate compiler builtins at the LLVM IR-level during lifting. Listings 1 and 2 show the translated IR blocks for both the approaches. Here, we perform the write to the (virtual) `eax` as part of a separate instruction that depends on the result of the `cmpxchg`. However, we need to ensure that (1) the loads from the virtual registers (`eax`, `ecx`) are not reordered after the `cmpxchg` and before any other stores that target them. (2) the conditional store to `eax` is not reordered after any use of `cmpxchg`. To prevent such instruction reorderings, we mark the `cmpxchg` as sequentially consistent (`seq_cst`) and surround the translated IR block with compiler barriers. Since registers are not accessed indirectly, we can be certain that no other thread will race to write to the storage location of the `eax` register. We manually check the correctness of such translations for all supported hardware atomic instructions.

To preserve atomicity guarantees for memory operations asserted by the ISA, we maintain original alignments for, (1) global variables, by placing them at their original addresses, (2) program stack, by initializing the emulated program stack with the ISA mandated alignment.

**3.3.2 Per-thread Stack.** Polynima-lifted IR operates on a virtual CPU state that consists of registers, flags and stack memory, that are represented as global variables. For lifted functions, we implement a conservative version of the prototype recovery algorithm as described in Elwazeer et al. in [17]. Functions take as arguments output registers (registers they may read and write to) and input registers (registers they may only read from). All functions only rely on

```

lock(@global_lock)
temp = *%rsi
if %eax == %temp:
    %flags.z = 1
    *%rsi = %ecx
else:
    %flags.z = 0
    %eax = %temp
    *%rsi = %temp
unlock(@global_lock)

```

Listing 1. Naive

```

compiler_barrier()
old = %eax
new = %ecx
orig = cmpxchg *%rsi,
%old, %new seq_cst
%flags.z = %orig == %old
if !%flags.z:
    %eax = %orig
compiler_barrier()

```

Listing 2. Optimized

the validity of the stack pointer register passed in as an argument, and do not make any other assumptions about the stack.

To support multithreaded binaries, we mark variables that represent the global state as `thread_local` ensuring that each thread operates on its own copy of the virtual state. We repurpose the callback wrappers to identify if the binary is in a new thread of execution, and use it to initialize relevant thread-local CPU state such as the segment registers and flags. We allocate memory that acts as the emulated stack for the call-graph starting at the thread-specific entry point, and copy over caller-provided arguments from the native stack into it. The emulated stack pointer then is initialized to point into this allocation.

**3.3.3 Callbacks.** External library calls take function pointers as arguments when, (1) performing callbacks, such as in the case of `qsort` that requires a user-defined comparator function, (2) spawning new threads of execution, such as in the case of `pthread_create` which requires an entry-point in the new execution context. Statically identifying the values of the arguments to such calls is hard, as function pointers could be materialized in registers or loaded from memory at run time. To remain general, recompilers that recover functions must assume that any lifted function could be used as an external entry point.

We insert trampolines at addresses of each of the function starts in the original binary that jump to custom wrappers that enable transition of the execution context from the library code to the lifted code. We implement them to also support handling the case when the execution is part of a new thread. This way, irrespective of whether we can statically identify if the original binary spawns new threads, we perform correct recompilation. However, to achieve this we need to mark all lifted functions as `external` at the IR-level since LLVM could optimize away or inline functions that act as possible external entry points. This increases the overall code size as we need to preserve all function bodies and their callback wrapper implementations during the recompilation process. This approach also hinders interprocedural compiler optimizations which affects the performance of the recompiled binary.



To that end, we implement a dynamic analysis based instrumentation pass, on top of the lifted IR, that records the names of functions used as callbacks for a given set of inputs. We merge information collected across different runs and subsequently remove wrappers for functions that are not observed as external entry points and unmark them as external. This makes them available to the compiler for aggressive optimization, which benefits the recompiled output in terms of code size and performance. Note that this is an optional optimization step, and that the recompiled binary provided as input to this stage is a fully functional replacement of the original input.

**3.3.4 Fence Insertion.** We adopt the fence insertion strategy formalized by Lasagne [38] for x86/x64 to handle memory access reordering at the IR-level. LLVM IR [30] provides acquire and release fences with semantics as specified by the C++ standard [25]. We insert, (1) an acquire fence after every load, preventing successor loads and stores from being reordered *before* it, (2) a release fence before every store, preventing predecessor loads and stores from reordering *after* it. We apply these rules to memory accesses executing in the context of the original binary instead of those generated as a byproduct of the lifting process. We also identify stack-local accesses by tracking loads from and stores to locations derived *directly* (addition and subtraction operations) from the emulated stack pointer. We remove fences that were inserted to protect such accesses.

### 3.4 Fence Optimization

Programs synchronize shared memory accesses through primitives such as barriers, locks, mutexes and semaphores. To achieve this, programmers often rely on external libraries for the implementations of such primitives, like those provided by `std::atomic`, `threads`, and `OpenMP`. But, they can also choose to implement custom or *implicit* synchronization primitives when available constructs are too slow or do not provide specific guarantees and control. Our key insight is that demonstrating the absence of implicit primitives in the binaries of data-race free programs can be leveraged to remove superfluous fences during their recompilation. Note that as we recompile binaries for the same architecture, we care about memory access reorderings only at the IR-level.

**3.4.1 Approach.** Consider an analysis which identifies if given machine code *does not* implement any implicit synchronization primitives. We describe two scenarios where this is the case, and discuss the issue of shared memory access reordering during lifting in each case. Since all shared memory accesses in a data-race free program must be synchronized,

- *the code exclusively uses synchronization mechanisms provided by external libraries:* In this case, the compiler is prohibited from reordering memory accesses across an external call as it is unaware of the possible

side effects it may have. Therefore, it conservatively preserves the original access ordering irrespective of whether fences are inserted in the IR.

- *there exist no shared memory accesses that need synchronization:* If no two threads are racing to access the same location, each memory access can be considered to be synchronized. In this case, reorderings maintain original program semantics, making the inserted fences redundant.

Hence, if it can be shown that the given program does not implement any implicit synchronization primitives, we can remove fences that are inserted to prevent reorderings in the IR. To that end, we design a dynamic analysis based instrumentation pass to detect implicit synchronization primitives in machine code.

Prior work [19, 26, 33, 34, 40] identifies that the basic pattern necessary for implementing such a construct is a spinloop. Since multiple definitions of a spinloop exist in literature, we choose the most permissive one as described in AtoMig [8].

For each loop that we identify in the lifted IR, our analysis procedure checks if it is *NOT a spinloop*. We achieve this by showing that it is possible to exit the loop due to the influence of a local value that is, (1) not loop-constant and, (2) lacks external dependencies. A value is defined to have an external dependency if it depends on a shared memory access through some data flow.

Note that AtoMig detects spinloops to identify potentially racy memory accesses based on instruction influence analysis of the *spin controls*. They transform such operations to be sequentially consistent, to achieve the correct translation of programs written for a stronger memory model (TSO) to a weaker memory model (WMM). With Polynima, we aim to identify spinloops to infer if the binary implements custom synchronization primitives as part of its code.

We illustrate the various cases through examples in Listing 3. We assume that the value `%op` is one of the operands for a loop termination condition and that the rest of the statements for each of the examples belong to a loop body.

Let us first consider the spinloop cases. (a) has a direct external dependency on `@g` and (b) has an indirect external dependency (through `store`) on `@g`. In both of these cases, external dependencies (`@g`) may be modified by other threads between loop iterations, which may invalidate assumptions about loop termination. This is usually how spinloops are implemented, where one thread spins until it can get exclusive access to a protected shared resource. (c), on the other hand, has a local store of a constant value 1. Constant value stores do not affect loop termination across different iterations. For all of these cases, we mark the loops as potentially spinning.

Now, we consider the non-spinloop cases. (d) depends on a local store of a non-constant value. This is seen in cases where local program variables are accessed through memory

```

@g = global i32 @0
define void @samples() {
    %1 = alloca i32
    ; (a) Spinloop
    %op = load i32, i32* @g

    ; (b) Spinloop
    %5 = load i32, i32* @g
    store i32 %5, i32* %1
    ; ...
    %op = load i32, i32* %1

    ; (c) Spinloop
    store i32 1, i32* %1
    ; ...
    %op = load i32, i32* %1

    ; (d) Non-spinloop
    %7 = load i32, i32* %1
    %8 = add i32 %7, 1
    store i32 %8, i32* %1
    ; ...
    %op = load i32, i32* %1

    ; (e) Non-spinloop
    %3 = phi i32 [0, %entry],
           [%op, %back]
    ; ...
    %op = add i32 %3, 1
}

```

**Listing 3.** Examples of spinloops and non-spinloops

loads and stores instead of through registers, such as for unoptimized IRs. Whereas, (e) depends on a loop-modified value, which demonstrates a typical case of a locally-stored loop index being used to execute a fixed number of iterations. For both of these cases, we also ensure that there is no dataflow of an external dependency into %op. The influence of such an operand on the loop termination condition would be sufficient to mark it as non-spinning according to our definition above.

**3.4.2 Analysis.** We first recursively inline all lifted functions in the body of their callers to enable data flow tracking across procedure calls. Next, we perform the LLVM-provided loop simplification pass to restructure loops such that they have dedicated exit blocks. This enables the precise analysis of their termination conditions. Polynima’s functional IR representation enables us to rely on LLVM’s standard compiler passes to perform these transformations.

We annotate and instrument all memory access sites i.e. loads, stores, RMWs and CmpXCHGs to record the memory location and the access type i.e. local or shared. Polynima can differentiate between stack-local and shared memory accesses as we control the allocations for each thread’s emulated stack. We then run the recompiled binary with a set of concrete inputs to record dynamic analysis information for the instrumented memory accesses. After merging data collected across various runs, we map each memory access site to a list of tuples, each containing the observed location and the access type.

Next, we iterate over all loops in the lifted IR and analyze them individually. Polynima performs an instruction influence analysis, which we model as a backwards dataflow analysis, for operands of each of the loop termination conditions. The goal here is to identify if any of the operand values are influenced by loop-modified local value. It is trivial to perform this analysis for values that are not influenced

by memory accesses, by following their use-def chains in the IR. We typically observe this for source-level variables that are mapped to a register storage for the duration of the loop body, such as loop indices. In this case, we benefit from lifting general-purpose registers as SSA values.

Performing this analysis for source variables that are stored in memory, and influence the loop termination condition, requires chasing memory loads and stores. We resolve these queries using the dynamically recorded information. For all sites that access *shared* memory locations, we assume an external dependency and discard checking further. For *local* accesses, we collect all intra-loop stores made to that location and trigger another backwards dataflow analysis for the stored values. If the stored value is, (1) not loop-constant and, (2) lacks external dependencies, we can assert that the loop is non-spinning.

Once we identify that all lifted loops in a binary are non-spinning, we conclude that the fences that were inserted to prevent IR-level memory reorderings are superfluous.

**3.4.3 Limitations.** *False positives.* Falsely asserting the absence of implicit synchronization may lead to unsound recompilation. Our approach can fail for programs that synchronize shared memory accesses *without spinning*, using only loads and stores. This occurrence is uncommon, as construction of any non-trivial and wait-free synchronization mechanisms requires the usage of atomic read-modify-write accesses [20].

We do not support programs that use primitives based on sleep-based contention, and other asynchronous methods (signals and syscalls). But, we argue that through our dynamic analysis we can detect timeout-based synchronizing loops in at least one running thread because, (1) we assume that the programs under analysis are data-race free, (2) for progress to be made toward program completion, at least one of the spinning loops has to successfully exit. We did not find evidence of the use of any of the above mechanisms as part of our benchmarks.

*False negatives.* Our dynamic approach is limited by the loop-coverage that is achieved through the provided inputs. We may falsely identify that a program uses implicit synchronization if we, (1) do not realise a loop body during the execution runs, (2) identify a non-spinning loop to be spinning. Polynima also does not build precise happens-before relationships or perform lockset analysis to identify shared memory accesses that belong to a loop but are already synchronized. This may result in false negatives that are not resolvable through dynamic analysis. In such a scenario, we remain conservative and preserve all inserted fences in the IR, possibly affecting performance but not correctness.

## 4 Evaluation

Our evaluation is guided by the following research questions:

**RQ1:** Does Polynima make available the transformation in-

infrastructure, available as part of LLVM, to fix and improve legacy multithreaded binaries?

**RQ2:** Can we recompile a diverse set of complex real-world multithreaded binaries while maintaining correctness and ensuring a reasonable performance cost?

**RQ3:** How effective is our fence optimization approach?

**RQ4:** Does our hybrid control flow recovery approach improve state-of-the-art?

**Environment and Software.** We wrote a wrapper around the radare2 [4] disassembler to output a static (JSON-based) control flow graph representation that includes functions and the basic blocks belonging to them. The ICFT tracer, implemented as a Pin [31] tool, augments this representation with dynamically collected indirect control transfers. We then invoke the translator module, which is built on top of S<sup>2</sup>E’s [11] RevGen [10] utility. This provides us the infrastructure to translate individual machine code basic blocks to LLVM IR (LLVM 14). S<sup>2</sup>E achieves this by first translating machine code to QEMU’s TCG intermediate representation and then to LLVM IR. Our translations for atomic instructions are implemented on top of the upstream S<sup>2</sup>E.

In the lifted IR, we stitch together lifted basic blocks to create functions based on the recovered control flows. Finally, the rest of our lifting pipeline builds on top of BinRec [3], leveraging passes that enable us to deinstrument the IR emitted by the translator and its infrastructure for lowering the lifted bytecode.

Polynima can be accessed through a single command-line utility that provides facilities for project management, disassembly, lifting and (additive) recompilation of binaries. Users need only provide inputs that exercise control flows for the optional dynamic analyses. Writing patches for binaries using Polynima is akin to writing a compiler-level pass for LLVM IR, with the option of adding a runtime component that can be linked in. A custom transformation pass can be integrated with the standard compiler infrastructure by registering it with LLVM’s PassManager.

We conducted our experiments on a Ubuntu 20.04 LTS system with an Intel i7-8700K CPU running at a base clock of 3.70 GHz, 32 GB RAM, and 6 cores. To ensure stable performance, we disabled frequency scaling, hyperthreading, and frequency boosting. We ran each input five times for performance experiments, summed up their means, and calculated the normalized runtime as a fraction of the baseline. We compiled all binaries with gcc-8, with stack-protector and position-independent execution disabled (`-fno-stack-protector -no-pie`), and optimization level 03, except for ConcurrencyKit, which defaults to 02.

**Comparison with other lifters.** We tried running other state-of-the-art lifters identified in Liu et al. [29] to lift the binaries that we choose for our evaluation. The authors of RetDec [27] suggest that the tool is designed as a binary lifter, instead of a recompiler, and that the IR is unsuitable for re-

compilation. Likewise, McSema’s [16] authors conveyed that the tool’s main focus is binary lifting and its overall recompilation capabilities are experimental. To evaluate Rev.Ng [15], we used musl-gcc and statically compiled a multithreaded version of the simple “hello world” program. Although we recover a translated binary, we observe faults during execution of the `do_fork` procedure, indicating a lack of support for multithreaded machine code. Lasagne [38], which builds on top of mctoll [42], supports the lifting and recompilation of a subset of multithreaded binaries. However, we could not lift any other binaries apart from those belonging to the Phoenix benchmark suite using their prototype.

To our knowledge, Polynima is the **only** binary recompiler that supports real-world multithreaded programs while maintaining original program semantics.

#### 4.1 Exploit Detection (RQ1)

We focus on detecting and mitigating CVE-2023-24042 [12], a recently discovered synchronization bug in LightFTP which enabled path traversal and possibly other security issues. The bug manifests because the variable (and the context) used to track the requested file name and the session user name is reused across the different threads creating a race condition. Below is the sequence of steps a malicious user would perform for a directory traversal exploit,

- Send the `LIST` command with an existing directory name as the parameter, writing the path in `context->FileName` and spawning a blocked handler thread.
- Send the `USER` command with a filename of choice (e.g., `/etc/passwd`) as the parameter, overwriting `context->FileName` with this value. Note that no checks are performed for this write.
- Connect to the data socket which unblocks the handler thread for the `LIST` command. The handler now uses the value stored in `context->FileName`, which has been overwritten in the previous step.

We identify that the program calls `stat` to check the file status in function `ftpLIST` before spawning the handler thread. The handler function `list_thread` then calls `opendir` to open a directory stream corresponding to the requested path and return a list of files. We write an LLVM pass which records and compares the path arguments passed to the `stat` and `opendir` calls. During benign execution of the program, both would correspond to the same value, but would be different in the case of an exploit.

The operator is enabled to take various actions in an exploit scenario with a Polynima-recompiled LightFTP binary. They may divert the code to a custom runtime handler, written in plain C/C++, similar to a “patch” in source programs. They could also choose to log the event for forensics or stop the server entirely. Since we lift external calls and their arguments, it is also possible to replace the value stored in `context->FileName` with the older value to protect against

the exploit. Also, the operator has complete control over the set of valid control transfers in the lifted IR. They may choose to completely disable certain allowed commands by either rewriting their handler implementations or by limiting the available targets of a jump-table style command dispatch.

The actual fix for this bug involved major changes, maintaining a per-handler context structure consisting of the file’s username and path. We argue that in the cases where fixes for such bugs are unavailable due to lost source or lack of vendor support, Polynima’s capabilities to generate a replacement binary are demonstrably valuable. The compiler pass and the runtime instrumentation code account for only about 70 lines of C++. Thus, we enable a *usable and powerful interface* for performing program-wide transformations that can leverage the LLVM compiler infrastructure.

#### 4.2 Compatibility and Performance (RQ2)

We test Polynima on a large and functionally diverse set of binaries that comprises real-world utilities and benchmark suites listed in Table 1. *We report correct outputs across all the test cases that we run.*

**memcached** uses pthreads along with compiler builtins for threading and synchronization. We use the tool memaslap to check the correctness and benchmark the recovered binary performance under load. We run memaslap for 2 minutes with the default configuration of the get/set request proportion (0.9/0.1) with 2 and 4 threads in each case. In both cases, the recovered binary reports a less than a 1% difference in the total number of operations performed.

**pigz** exclusively uses functions provided by pthreads. We benchmark pigz by compressing two files with compression levels fast, default and slow and across the use of 1 / 2 / 4 threads. We observe negligible differences in data processed (in mbs per second) and the total time required for compression in each of the configurations.

**mongoose**. We compile the default multi-threaded web-server example to test mongoose which uses pthreads. We configure the siege utility to spawn 25 concurrent threads sending requests to the server for 2 minutes. The average response time for the original server binary is reported to be 2.02s v/s the 2.03s for the recovered one, indicating a minimal performance difference.

**LightFTP**. For LightFTP, which also uses pthreads, we stress test the upload and download speeds for the original and recovered binary. We achieve this by sending concurrent upload and download requests of 1 MB files for ~45 seconds. The average upload times differ by a margin of 2.4% and the download times differ by 9%.

**Phoenix**. Table 2 contains the results for the Phoenix benchmark suite, which contains map-reduce style programs that are used to benchmark parallel executions. Phoenix also uses pthreads for synchronization and threading. We use the provided small, medium, and large input datasets to evaluate performance of the recompiled binaries.

We first highlight the performance of Polynima recompiled binaries for the 00 baseline. For unoptimized binaries, recompiled binaries perform at par or better than the input with an average speedup of 0.98x, with a maximum speedup of 0.90x in the case of *histogram*. In these cases, we observe performance benefits as the compiler, (1) is effective in optimizing the lifted IR, (2) is free to choose SIMD instructions available as part of the underlying hardware for efficient lowering. These results show that Polynima could be useful as a post-release optimizer, for binaries that were originally compiled with little to no optimizations for an older CPU version.

**gapbs**. The gapbs benchmark suite contains reference implementations of various graph processing algorithms. Programs use OpenMP for parallelization, specifically annotating loop bodies with `#omp parallel` pragmas for concurrent execution. They also use primitives from `std::atomic`, that lower to x86/x64 hardware atomic instructions, for synchronization.

We evaluate all gapbs binaries (Table 3) on integer inputs, for which we use uniform-random graph inputs of size  $2^{20}$  for each binary. With gapbs, we observe similar trends as Phoenix i.e. close to original performances for unoptimized binaries and slowdowns for the optimized versions.

**Performance Discussion**. We use the geometric mean of the results for the unoptimized (00) and optimized (03) Phoenix and gapbs benchmark suites to compute the overall 1.23x slowdown. We now discuss the major reasons for degradation in recompiled output performance for optimized binaries (03) in gapbs and Phoenix.

Recompiled output performs memory accesses which are part of the original binary onto an emulated stack, which helps Polynima remain general in its approach to lifting binaries. However, note that most optimizations in the LLVM ecosystem are designed to work with an IR that contains program variables along with their type information. Since we do not recover this, LLVM has to treat the emulated stack as entirely opaque, which prevents off-the-shelf optimizations from being fully effective.

We also notice the cost introduced due to the non-optimal lifting of SIMD instructions and floating point operations. Polynima relies on QEMU [9] helpers to provide translations for such instructions, which are based on emulating them on the virtual CPU state. For certain vector instructions, LLVM can resynthesize them into intrinsics after lifting, but this translation is not optimal. Also, we represent the XMM / YMM registers as globals (as opposed to being function-local) in the IR, which prevents further optimizations. The performance impact is most visible in the *linear\_regression* benchmark, where the core algorithm is implemented as a packed sequence of SIMD instructions in the original binary.

Finally, with OpenMP, each of pragma-annotated loops compile into a distinct function which acts as an entry point

**Table 1.** Supported Benchmarks. Lasagne builds on top of mctoll.

Benchmark	Description	LOC	Polynima	Lasagne	McSema	BinRec	Rev.Ng
memcached [41]	Key-Value Store	24.4k	✓	✗	✗	✗	✗
mongoose [32]	Web Server	7.4k	✓	✗	✗	✗	✗
pigz [1]	Compression Tool	6.4k	✓	✗	✗	✗	✗
LightFTP [21]	FTP Server	2.4k	✓	✗	✗	✗	✗
Phoenix [37]	Data Processing	4.4k	7/7	5/7	0/7	0/7	0/7
gapbs [7]	Graph Processing	2.8k	8/8	0/8	0/8	0/8	0/8
CKit (spinloops) [2]	Sync. Primitives	1.3k	11/11	0/11	0/11	0/11	0/11

**Table 2.** Performance of Polynima recompiled binaries on the Phoenix benchmark suite. Results in the FO column report performance after application of the fence removal optimization.

Benchmark	O0	O0 FO	O3	O3 FO
histogram	0.90	0.82	1.01	1.01
kmeans	0.91	0.58	1.43	1.11
linear_regression	1.07	0.97	3.71	3.60
matrix_multiply	0.98	0.94	1.25	1.25
pca	0.98	0.72 (✗)	2.46	2.46 (✗)
string_match	1.08	1.07	1.34	1.29
word_count	0.97	0.92	1.03	0.89
<b>Geomean</b>	0.98	0.85	1.56	1.46

**Table 3.** Performance of Polynima recompiled binaries on the gapbs benchmark suite.

Benchmark	32-bit		64-bit	
	O0	O3	O0	O3
bc	1.20	2.48	1.26	1.17
bfs	0.87	1.02	0.94	1.01
cc	0.93	0.97	0.88	1.02
cc_sv	0.92	0.97	0.88	1.04
pr	1.90	2.94	1.37	1.81
pr_spmv	2.03	3.08	1.45	1.92
sssp	0.85	1.06	0.89	1.01
tc	1.30	1.42	1.40	1.41
<b>Geomean</b>	1.18	1.55	1.12	1.32

into a new thread context. This involves handling a large number of callbacks, 19 on average, which we identify to be another reason for the performance slowdown. Callback-handling includes marshaling of the native registers, copying arguments to the emulated stack, and copying returned registers back to the native state after execution of the lifted function.

We could not reliably recompile most of our benchmark

programs with other recompilers. Polynima builds on top of BinRec which outperforms McSema and Rev.Ng recompiled binaries on single-threaded benchmarks [3]. As a result, we expect Polynima to perform better than or at least as well as BinRec in comparison to them. Lasagne reports performance results for a subset of binaries from the Phoenix benchmark suite for the downstream task of cross-ISA translation to a different architecture (ARM64), which we do not support yet.

**ckit.** ConcurrencyKit implements custom concurrency primitives using compiler builtins (C99) that compile down to use hardware atomic instructions. We first successfully perform correctness checks for all 11 spinlock implementations using the validation test suite. We then use the latency benchmark test as part of the regressions suite to compute the average latency (in terms of number of clock cycles required) for each spinlock. Each individual test consists of a sequence of lock and unlock operations, executed in a loop. As these involve the lifting and lowering of various hardware atomic instructions, the results help us evaluate our approach to their translation. We report that the recompiled binary performance is close to the original in almost all cases (Appendix A Table 5), which validates our earlier claims of efficiency and correctness.

### 4.3 Implicit Synchronization Detection (RQ3)

Next, we evaluate the precision of Polynima’s spinloop detection as well as the performance improvements that we derive due to the subsequent fence removal. We first validate our approach on the various spinlock implementations in ConcurrencyKit as representative examples of *implicit* synchronization primitives. Then, we evaluate it on the Phoenix benchmark suite, which explicitly uses external synchronization primitives, where we benefit the most by proving the absence of implicit synchronization. Recompiled binaries are run against provided inputs with instrumentation that records information for all memory accesses.

*False positives.* We do not observe any false positives in the experiments that we perform.

*False negatives.* In *histogram*, we fail to cover one loop body that swaps data bytes depending on the endianness of the

underlying architecture. Since no inputs we provide would cover this loop (on x86/x64), we manually analyze it as a non-spinloop and report the results.

We observe a false negative in the case of the *pca* binary, as it requires a precise happens-before analysis for proving that a certain loop is non-spinning. This does not affect correctness however, as we default to preserving already inserted fences. We still report the results after performing fence removal to demonstrate the impact on recompiled binary performance.

*True positives.* Apart from the two cases mentioned above, we cover and check that all other loops from Phoenix are correctly identified as non-spinning.

*True negatives.* We correctly identify all spinloops in binaries compiled from the validation test suite for the various spinlock implementations in ConcurrencyKit.

We refer to Table 2 for performance discussion of this optimization. We observe that removing superfluous fences leads to a notable improvement in performance for nearly all test cases. The average speedup observed for unoptimized binaries is improved to 0.85x, further pushing the case for using Polynima as a post-release optimizer. Removing fences enables off-the-shelf compiler optimizations to be more effective. Crucially, we observe an astounding 27% improvement in performance for *kmeans* after removing redundant fences. Improvements are also observed for optimized binaries, where the slowdown is improved to 1.46x.

#### 4.4 Lifting Time (RQ4)

**Overall lift time.** We now compare the performance of our control flow recovery approach with that of BinRec and McSema. As neither of the above recompilers support multithreaded binaries, we apply Polynima to O3-compiled binaries from the SPECint 2006 benchmark suite.

For Polynima, we statically collect the CFG and augment it with information from the ICFT Tracer, which is driven with the *ref* inputs for each binary. We ensure the correctness of our control flow recovery process by checking the output of the recompiled binary against the *ref* inputs. Our prototype was unable to handle *403.gcc* and *483.xalancbmk* due to failed IR translation for certain superfluous code paths.

We report the total time taken to disassemble, trace, and recompile with Polynima in Table 4. We refer to the BinRec paper [3] for relevant numbers for BinRec and McSema. Polynima performs orders of magnitude faster than BinRec while also providing the same precision in terms of the recovered control flow. Also, our performance is comparable to McSema, an entirely static lifter.

To highlight the importance of our hybrid approach, we also report the number of indirect control flows recorded during the tracing process for each program. Consider the case of *429.mcf* and *462.libquantum* that contain no indirect transfers. In such a case, an entirely static approach is efficient and preferable as the disassembler generated output

**Table 4.** Lifting Times (in s) the for SPEC INT 2006 binaries against *ref* inputs and the total number of ICFTs (indirect control flows) recorded in the process

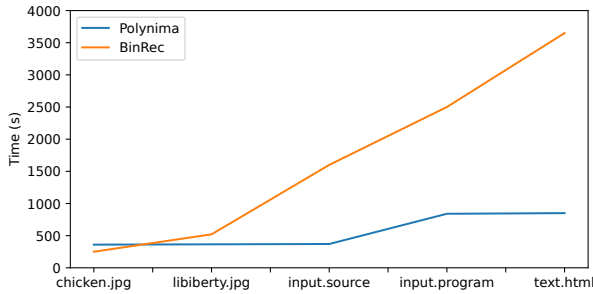
Benchmark	Polynima	BinRec	McSema	ICFTs
401.bzip2	47	69389	3385	21
403.gcc	1380	28468	7378	2350
429.mcf	130	227999	8	0
445.gobmk	634	72307	1063	1241
456.hmmmer	427	144529	189	34
458.sjeng	1399	548342	368	69
462.libq.	425	176536	16	0
464.h264ref	1885	65202	586	116
473.astar	265	119436	18	2
483.xalanc.	–	–	17103	–
<b>Geomean</b>	445	137074	238	–

can be considered precise and complete. However, BinRec performs poorly for both the benchmarks as it needs to trace through the entire program before being able to generate the recompiled output.

On the other hand, for a program such as *445.gobmk* it is difficult for a static disassembler to precisely resolve such a large number of indirect control transfers (1241). Recent work was unable to functionally verify McSema-recompiled binaries for more than half of the SPEC benchmark suite [29]. In this case, Polynima’s hybrid approach performs notably better than BinRec, while providing the same precision.

**Additive lifting.** We lift all of our multithreaded benchmark binaries using additive lifting to test the scalability and robustness of the approach. To evaluate its performance, we compare against BinRec’s incremental lifting and report the results in Figure 4. We use the *401.bzip2* binary from the SPEC benchmark suite as it was chosen as the demonstrative example in the original paper. We start our measurements by considering a recompiled binary that supports the SPEC *test* inputs. We then measure the time taken (represented by the Y-axis) by both approaches for increasingly complex input files (represented by X-axis).

To summarize, Polynima decouples the process of CFG collection from translating machine code to IR. Performing the IR translation *offline* is key for recompilation to scale to large binaries. Unlike BinRec, that executes the input program inside a full-scale processor emulator, we run the recompiled output natively. That way, we leverage the relatively low overhead of the recompiled output and do away with the long startup times and emulation cost. Whenever Polynima discovers a new control transfer, it statically explores the CFG starting at this block and retrofits discovered paths backs into the known CFG. As a result, we see recompilation loops only triggered for *chicken.jpg* and *input.program*, where we explore yet unknown sections of the input CFG.



**Figure 4.** Lifting times for BinRec’s Incremental lifting v/s Polynima’s Additive lifting for 401 .bz2

## 5 Discussion

### 5.1 Stack-Variable Recovery

Compiler passes are designed to be used with *refined* IRs with higher-level constructs such as program variables that partition the program’s stack memory into distinct objects. This is typically seen in IRs that are compiled down from source code. Our prototype does not recover a mapping between stack slots and variable objects, which would enhance optimizations and enable fine-grained transformations such as AddressSanitizer and SafeStack. SecondWrite [17] determines such a mapping using a heuristics-driven conservative static analysis. On the other hand, WYTIWYG [36] decomposes the problem of local variable recovery into a series of instrumentation-based dynamic analyses.

Fences, inserted to prevent IR-level memory reorderings, can prevent aggressive compiler optimizations and thus completely defeat the purpose of stack variable recovery. In such a case, our fence optimization approach would effectively “unlock” optimizations when recompiling multithreaded machine code. We leave the integration of such an approach within Polynima as future work.

### 5.2 Coverage

Polynima’s dynamic analyses rely on inputs that provide comprehensive coverage of the program. These analyses include removing unnecessary callback wrappers to reduce code size, and determining if our fence optimization is safely applicable to the lifted program. The ICFT tracer module also benefits from inputs that drive coverage as it reduces the total number of recompilation runs required through additive lifting. Reliance on such inputs is a limitation that is typical of dynamically-driven analyses across literature.

But, we emphasize that the above analyses are optional for generating a functionally compatible recompiled binary with Polynima. Our hybrid control flow recovery approach, conservative callback handling and fence insertion (which we adopt from Lasagne) ensure that correct recompilation can be achieved through static-only techniques. This way, Polynima benefits from the availability of extensive test-suites

for known programs and advancements in techniques such as fuzzing and symbolic execution that recover coverage-inducing inputs for unknown binaries.

### 5.3 SIMD instructions

First-class support for SSE / AVX / AVX512 instructions and consequently XMM / YMM / ZMM registers is crucial for achieving close-to-original performance when recompiling binaries optimized for modern architectures. Lasagne [38] implements such translations for certain SSE-based floating point instructions. Instrew [18] optimally lifts SIMD instructions and vector registers in machine code to LLVM IR for dynamic binary translation. With Polynima, we aim to achieve ahead-of-time recompilation to generate a functionally compatible and performant replacement of the original binary. Precisely translating individual SIMD instructions by mapping them to the relevant LLVM intrinsics at the IR-level, similar to how we handle hardware atomic instructions, is key to solving this problem. Finally, most of the limitations that we impose upon input binaries (Section 3.1) can be resolved with additional engineering effort.

## 6 Conclusion

We presented Polynima, the *first practical* binary recompiler for multithreaded x86/x64 binaries. We implement a hybrid control flow recovery approach that combines the benefits of static and dynamic techniques while also providing an efficient miss handling strategy. Our tool makes available the rich LLVM compiler ecosystem to fix and improve legacy binaries, which we demonstrate by mitigating a recently published synchronization issue in a vulnerable FTP server binary. Polynima is evaluated on a wide range of real-world utilities and benchmark suites, demonstrating its ability to *correctly* lift complex multithreaded machine code. Finally, we leverage the functional IR to design an innovative dynamic analysis for detecting implicit synchronization primitives in binaries, which we use to remove superfluous fences and improve recompiled binary performance.

## Acknowledgments

We express our gratitude to our reviewers and to our shepherd, Redha Gouicem, who greatly improved the paper with their valuable remarks. We also thank Hongyu Chen for help with proofreading the manuscript.

This material is based upon work partially supported by the Office of Naval Research (ONR) under contracts N00014-22-1-2232 and N00014-21-1-2409, and the Defense Advanced Research Projects Agency (DARPA) under contracts W31P4Q-20-C-0052, 140D04-23-C-0063 and 140D04-23-C-0070. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of ONR or DARPA.

## References

- [1] [SW] Mark Adler, pigz 2007. URL: <http://zlib.net/pigz>, vcs: <https://github.com/madler/pigz>.
- [2] [SW] Samy Al Bahra, ConcurrencyKit 2011. URL: <https://github.com/concurrencykit/ck>.
- [3] Anil Altinay et al. 2020. BinRec: dynamic binary lifting and recompilation. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)* Article 36. Association for Computing Machinery, Heraklion, Greece, 16 pages. ISBN: 9781450368827. DOI: 10.1145/3342195.3387550.
- [4] [SW] Sergi Alvarez, Radare2: Libre Reversing Framework for Unix Geeks 2006. radare.org. URL: <https://rada.re/n/radare2.html>, vcs: <https://github.com/radareorg/radare2>.
- [5] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. 2013. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. Association for Computing Machinery, Prague, Czech Republic, 295–308. ISBN: 9781450319942. DOI: 10.1145/2465351.2465380.
- [6] Gogul Balakrishnan and Thomas Reps. 2010. Wycinwyx: what you see is not what you execute. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32, 6, 1–84.
- [7] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP benchmark suite. (2017). arXiv: 1508.03619 [cs.DC].
- [8] Martin Beck, Koustubha Bhat, Lazar Stričević, Geng Chen, Diogo Behrens, Ming Fu, Viktor Vafeiadis, Haibo Chen, and Hermann Härtig. 2023. AtoMig: automatically migrating millions of code from TSO to WMM. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)*. Association for Computing Machinery, Vancouver, BC, Canada, 61–73. ISBN: 9781450399166. DOI: 10.1145/3575693.3579849.
- [9] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. USENIX Association, Anaheim, CA, (Apr. 2005). <https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator>.
- [10] Vitaly Chipounov and George Candea. 2011. Enabling sophisticated analyses of x86 binaries with RevGen. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, New York, NY, USA, 211–216. DOI: 10.1109/DSNW.2011.5958815.
- [11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. Association for Computing Machinery, Newport Beach, CA, USA, 265–278. ISBN: 9781450302661. DOI: 10.1145/1950365.1950396.
- [12] 2023. CVE-2023-24042. (2023). <http://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2023-24042>.
- [13] Chinmay Deshpande, David Gens, and Michael Franz. 2021. StackBERT: machine learning assisted static stack frame size recovery on stripped and optimized binaries. In *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security (AISec '21)*. Association for Computing Machinery, Virtual Event, Republic of Korea, 85–95. ISBN: 9781450386579. DOI: 10.1145/3474369.3486865.
- [14] Alessandro Di Federico and Giovanni Agosta. 2016. A jump-target identification method for multi-architecture static binary translation. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '16)* Article 17. Association for Computing Machinery, Pittsburgh, Pennsylvania, 10 pages. ISBN: 9781450344821. DOI: 10.1145/2968455.2968514.
- [15] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. RevNg: a unified binary analysis framework to recover cfgs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction (CC 2017)*. Association for Computing Machinery, Austin, TX, USA, 131–141. ISBN: 9781450352338. DOI: 10.1145/3033019.3033028.
- [16] Artem Dinaburg and Andrew Ruef. 2014. McSema: static translation of x86 instructions to LLVM. In *ReCon 2014 Conference, Montreal, Canada*.
- [17] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. 2013. Scalable variable and data type detection in a binary rewriter. In number 6. Vol. 48. Association for Computing Machinery, New York, NY, USA, (June 2013), 51–60. DOI: 10.1145/2499370.2462165.
- [18] Alexis Engelke and Martin Schulz. 2020. Instrew: leveraging llvm for high performance dynamic binary instrumentation. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20)*. Association for Computing Machinery, Lausanne, Switzerland, 172–184. ISBN: 9781450375542. DOI: 10.1145/3381052.3381319.
- [19] Rajiv Gupta. 1989. The fuzzy barrier: a mechanism for high speed synchronization of processors. *SIGARCH Comput. Archit. News*, 17, 2, (Apr. 1989), 54–63. DOI: 10.1145/68182.68187.
- [20] Maurice Herlihy. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13, 1, (Jan. 1991), 124–149. DOI: 10.1145/114005.102808.
- [21] [SW] hfiref0x, LightFTP. URL: <https://github.com/hfiref0x/LightFTP>.
- [22] R. Nigel Horspool and Nenad Marovac. 1980. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23, 3, (Aug. 1980), 223–229. DOI: 10.1093/comjnl/23.3.223.
- [23] [SW], IDA Pro 1991. HexRays. URL: <https://www.hex-rays.com/ida-pro>.
- [24] Intel Corporation. 2023. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Volume 1. (Jan. 2023). Retrieved Oct. 2, 2023 from <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf>.
- [25] ISO/IEC. 2020. *ISO/IEC 14882:2020, Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland.
- [26] Bohuslav Krena, Zdenek Letko, Rachel Tzoref, Shmuel Ur, and Tomáš Vojnar. 2007. Healing data races on-the-fly. In *Proceedings of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging*, 54–64. DOI: 10.1145/1273647.1273658.
- [27] Jakub Kroutek, Peter Matula, and Petr Zemek. RetDec: an open-source machine-code decompiler. (2017).
- [28] Chris Lattner and Vikram Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, 75–86. DOI: 10.1109/CGO.2004.1281665.
- [29] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, and Yuyan Bao. 2022. SoK: demystifying binary lifters through the lens of downstream applications. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1100–1119.
- [30] LLVM. 2022. LLVM atomic instructions and concurrency guide. (Mar. 25, 2022). Retrieved Oct. 2, 2023 from <https://releases.lldm.org/14.0.0/docs/Atomics.html>.
- [31] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*.
- [32] [SW] Sergey Lyubka, Mongoose 2004. Cesanta Software Limited. URL: <https://mongoose.ws>, vcs: <https://github.com/cesanta/mongoose>.



- [33] Peter Magnusson, Anders Landin, and Erik Hagersten. 1994. Queue locks on cache coherent multiprocessors. In *Proceedings of 8th International Parallel Processing Symposium*. IEEE, 165–171.
- [34] John M Mellor-Crummey and Michael L Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9, 1, 21–65.
- [35] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 2021. Sok: all you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 833–851.
- [36] Fabian Parzefall, Chinmay Deshpande, Felicitas Hetzelt, and Michael Franz. 2024. What you trace is what you get: dynamic stack-layout recovery for binary recompilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)*. Association for Computing Machinery, La Jolla, CA, USA. DOI: 10.1145/3620665.3640371.
- [37] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, 13–24.
- [38] Rodrigo CO Rocha, Dennis Sprokholt, Martin Fink, Redha Gouicem, Tom Spink, Soham Chakraborty, and Pramod Bhatotia. 2022. Lasagne: a static binary translator for weak memory model architectures. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 888–902.
- [39] Eric Schulte, Michael D Brown, and Vlad Folts. 2022. A broad comparative evaluation of x86-64 binary rewriters. In *Proceedings of the 15th Workshop on Cyber Security Experimentation and Test*, 129–144.
- [40] Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. 2008. Dynamic recognition of synchronization operations for improved data race detection. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, 143–154.
- [41] [SW] Anatoly Vorobey and Brad Fitzpatrick, Memcached 2003. Danga Interactive. URL: <https://memcached.org>, vcs: <https://github.com/memcached/memcached>.
- [42] S Bharadwaj Yadavalli and Aaron Smith. 2019. Raising binaries to LLVM IR with MCTOLL (WIP paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 213–218.
- [43] Mingwei Zhang and R Sekar. 2015. Control flow and code integrity for COTS binaries: an effective defense against real-world ROP attacks. In *Proceedings of the 31st Annual Computer Security Applications Conference*, 91–100.

## A CKit Performance

**Table 5.** Performance of the original and the recompiled output (in terms of number of clock cycles required) on the latency tests in CKit

Spinlock	Native	Recovered
ck_anderson	31	25
ck_cas	26	25
ck_clh	26	26
ck_dec	26	24
ck_fas	26	25
ck_hclh	57	57
ck_mcs	56	54
ck_spinlock	26	25
ck_ticket	36	49
ck_ticket_pb	36	35
linux_spinlock	26	23