Egalito: Layout-Agnostic Binary Recompilation

David Williams-King
dwk@cs.columbia.edu
Columbia University

Hidenori Kobayashi
hidenori.kobayashi@gmail.com
Canon Inc.†

Kent Williams-King
kwk@brown.edu
Brown University

Graham Patterson
Bloomberg L.C.†

Frank Spano
Bloomberg L.C.†

Yu Jian Wu
Columbia University

Junfeng Yang
junfeng@cs.columbia.edu
Columbia University

Vasileios P. Kemerlis
vpk@cs.brown.edu
Brown University

Abstract

For comprehensive analysis of all executable code, and fast turn-around time for transformations, it is essential to operate directly on binaries to enable profiling, security hardening, and architectural adaptation. Disassembling binaries is difficult, and prior work relies on a process virtual machine to translate references on the fly or inefficient binary code patching. Our Egalito recompiler leverages metadata present in current stripped x86_64 and ARM64 binaries to generate a complete disassembly, and allows arbitrary modifications that may affect program layout without any constraints from the original binary. We utilize our own layout-agnostic intermediate representation, which is low-level enough to make the regeneration of output code predictable, yet supports a dual high-level representation for sophisticated analysis. We demonstrate nine binary tools including a novel continuous code randomization technique where Egalito transforms itself, and software emulation of the control-flow integrity in upcoming hardware. We evaluated Egalito on a large set of Debian packages, completely analyzing 99.9% of a selection of 867 executables and libraries; a majority of 149 applicable Debian packages pass all tests under Egalito. On SPEC CPU 2006, thanks to our binary optimizations, Egalito actually observes a 1.7% performance speedup.

CCS Concepts. • **Software and its engineering** → **Software reverse engineering**; • **Security and privacy** → **Systems security**; **Software and application security**; *Software security engineering*; Information flow control.

Keywords. binary analysis, binary rewriting, recompilation, application security, software hardening

ACM Reference Format:
David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *Proc. of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3373376.3378470

1 Introduction

Software written in compiled languages ultimately runs in binary form, and the majority of software distributors provide binaries directly to their end-users. Since binaries are so widespread, it is desirable for many DevOps activities, including profiling, security hardening, and architectural adaptation, to apply directly to binaries. Applying changes to source code or compiler infrastructure has high turn-around time, requires the cooperation of many parties, and may not be possible in the case of commercial or third-party libraries and applications. Furthermore, security hardening and architectural adaptations must be applied comprehensively to all library dependencies, or risk compromise through an untransformed component; only at the binary level is it possible to transform all code that will actually run.

However, manipulating binaries directly is difficult. The developer who examines binaries feels more like an archaeologist than an engineer—dealing with artefacts of an unknown buildsystem, no blueprints, and many details lost to time. Automated binary rewriting tools must treat binary code as a black box that can be emulated but whose structure is unknown. The main difficulty of binary rewriting is in handling unidentified code and code references (pointers). Existing binary rewriting frameworks use either: 1) a process virtual machine to translate references on the fly; or 2) binary code patching, leaving code at original locations. Common frameworks like DynamoRIO [10] and Pin [38]

†Work done while at Columbia University.

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378470

use virtualization. Binary patching, on the other hand, often requires significant expertise with reverse-engineering tools like IDA Pro [20]. Both mechanisms incur performance overhead that is likely unacceptable for production environments. Generated outputs also do not behave like ordinary binaries, impeding debugging and compatibility with other tools.

In this work, we aim to create a binary rewriting framework that manipulates and outputs *ordinary-looking* and *highly performant* binaries. Specifically, we aim to allow program code (and layout) to change arbitrarily without any constraints from the original binary—i.e., the framework is *layout agnostic*. This requires complete disassembly: we must have confidence that every pointer has been found and updated to allow the code to run bare-metal within a new address-space layout. Our observation is that while malware analysis is and will always be a herculean task [41], for binaries the user depends on, it is important simply to be able to handle the actual output of compilers. Furthermore, today's binaries have recently started to include more metadata, which can assist with analysis. Most importantly, major Linux distributions have shifted to position-independent binaries over the past few years [18, 21, 39, 53]. Note that position-independent is a much weaker property than layout-agnostic: the former allows a single linear shift through the selection of a base address, while the latter allows piecewise permutation or relocation of each instruction. Nevertheless, this additional (position-independent) metadata allows more powerful binary analyses than in the past.

We present *Egalito*: a binary transformation framework that performs *complete* and *precise* binary analysis, and can generate output binaries that do not use patching or virtualization. Egalito lifts (stripped) modern Linux binaries into a standalone, layout-agnostic intermediate representation (IR) that allows arbitrary modifications to program layout. Essentially, Egalito is a compiler backend in reverse, followed by transformations and then normal code generation. Hence, we call Egalito a *binary recompiler*. Tools written with Egalito are structured as modular *recompiler passes* that manipulate IR. Egalito is fully functional on x86_64 and ARM64 architectures, with RISC-V support underway.

Our IR is not a high-level representation like LLVM IR [37] that a compiler (i.e., LLVM) uses for optimization [35]; rather, it is lower-level, like LLVM's `MachineInstr` or GCC's Register Transfer Language (RTL) [29]. Lifting to a higher-level intermediate language would abstract each instruction into multiple operations, not tied together semantically, which might be modified, reordered, and optimized independently. Code generation would become difficult, and likely diverge significantly from the input assembly. However, for low-level binary instrumentation or hardening, producing a differently-optimized version of the code is counterproductive. Instructions might have been carefully chosen to work around low-level architectural issues (e.g., Spectre [32]), or to perform additional security checks; an optimizing framework could undermine the defense. Hence, we deliberately use a lower-level IR, in order to make output code generation more predictable and more in-line with the original input.

As a result of our layout-agnostic design, binaries transformed by Egalito have excellent performance. On SPEC CPU 2006, Egalito incurs 0.46% overhead, which becomes a 1.7% performance *speedup* when we enable some simple binary-level optimizations. Sometimes, even when security hardening is applied, the transformed program may run faster than the original, e.g., we see a 1.4% performance speedup with lightweight control flow integrity. With our profile-guided optimization tool, performance can potentially become even better (the best SPEC CPU case observes 11.8% speedup). Our AFL fuzzing backend is 18-61x faster than other binary-level fuzzing. Hence, Egalito provides a realistic way to introduce binary modifications with production-level performance.

A binary rewriter that relies on completely accurate disassembly must have high confidence in its analyses. Thus, we tested Egalito thoroughly across multiple Linux distributions during its development, including Debian, Ubuntu, openSUSE, Fedora, and Gentoo. Our evaluation of 172 Debian packages containing 867 executables and libraries showed that in 866 cases (99.9%) Egalito correctly recovered all cross-references and jump table metadata (the most challenging aspect), including table bases, invocations, and bounds. After transformation, 90 of 149 Debian packages pass all tests—and fully 40 of the failures are due to a detectable binary-generation issue that can be addressed with additional engineering. As further evidence of completeness, Egalito is able to analyze and transform itself, analogous to a bootstrapping compiler. Our system is currently in use by researchers and instructors at several other institutions, and we hope that a wider userbase will give it even better robustness over time.

Our end goal is to have a full Linux distribution where every program can be readily transformed at the binary level to adapt to new attacks or architectural quirks [33]. A security-conscious user may be willing to trade off performance for security, enabling a suite of defenses on particular applications. We implemented nine binary tools atop Egalito, most of which increase security—including a JIT-Shuffling defense [61] which relocates functions periodically to random addresses, in a fully self-hosted environment with no untransformed code. A hardware designer creating new hardware might wish to remove—or add—certain sequences of instructions, without having to modify a compiler and recompile userspace. Two of our Egalito tools perform such binary-level architectural adaptation. One is a retpoline defense against Spectre [32], which replaces problematic instructions with new sequences. Another is a software implementation of Intel's Control-flow Enforcement Technology (CET) [28], which augments hardware and compiler deployment by handling partially-present instrumentation. All of our tools run on stripped binaries, built with typical flags: i.e., those used to build distribution-standard `.deb` or `.rpm` files.

Egalito has been released [59] under an open source license (GPL v3) and may be found at https://egalito.org/ [60].

Our main contributions are as follows:

- We define a framework to be *layout agnostic* if it can individually relocate or resize each binary element, without reliance on patching or address virtualization.
- We present the *Egalito recompiler*, a binary transformation framework built around a layout-agnostic IR called *EIR*. Egalito-transformed binaries achieve excellent performance: SPEC CPU has a 1.7% speedup.
- We demonstrate the usefulness of binary-level architectural adaptation with a retpoline defense against Spectre [32] and a software implementation of Intel's CET [28] to augment hardware/compiler deployment.
- We demonstrate Egalito with a total of nine transformation tools, including a continuous code randomization defense (JIT-Shuffling [61]), which operates from a fully self-hosted environment where tool code is itself defended recursively.
- We present a large-scale study of 867 Linux executables/libraries, and show that Egalito can fully analyze and recover all cross-references 99.9% of the time. Furthermore, 90 of 149 Debian packages pass all tests after binary rewriting (40 straightforward known failures).

2 Background and Related Work

Rewriting via Code Patching Statically rewriting a binary by installing hooks or trampolines in the original code (binary patching) is simple and efficient. Examples of patching-based rewriters include PEBIL [36], REINS [57], and ReVARM [30]. Patching can also be deferred until runtime as in Dyninst [12]. These systems do not attempt to find and transform all pointers in a binary, and overhead increases with more modifications, limiting the scale of the approach.

Process Virtualization In dynamic binary translation (DBT), a process virtual machine transforms each basic block, just-in-time, before it is executed. Existing DBT systems include DynamoRIO [10], Pin [38], Valgrind [42], and (on ARM64) Mambo [25]. These systems are not designed for security: for instance, a DynamoRIO tool that protects code pointers has no way to defend the code pointers in DynamoRIO's code cache or translation tables (and proof-of-concept exploits already exist [24]). Furthermore, DBT incurs substantial runtime overhead: 887% (DynamoRIO) and 3421% (Pin) on a large set of real-world x86_64 programs, and 28%-34% average (Mambo) on SPEC CPU. Valgrind, with its higher-level IR called VEX, has 330% overhead on SPEC CPU.

The PSI [64] platform is designed like a DBT system, implementing a process virtual machine and preventing control flow from escaping the instrumented version of the code. However, PSI operates through binary rewriting, statically inserting all the code necessary into an executable. It reuses a disassembler from prior work [56, 65]. PSI has a flexible architecture, and is designed for security, but does incur nontrivial virtualization overhead (53% on a large set of programs).

Multiverse [6] conservatively disassembles a binary starting from every offset within the `.text` section. This guarantees a complete but imprecise disassembly. Hence, they cannot easily identify all valid code pointers. Accordingly, Multiverse virtualizes addresses and preserves input program layout, incurring 60.42% average overhead on SPEC CPU, with 288% overhead in the worst SPEC CPU case.

Recompilation/Reassembly Binary analysis suites [11, 19, 49, 51] often include sophisticated analyses to lift binaries without metadata into a high-level IR (such as VEX or BIR). Most frameworks are not designed to regenerate code after analysis. However, the SecondWrite [3] binary rewriter lifts binaries to LLVM intermediate language and regenerates a new executable using LLVM's standard backend. SecondWrite operates on arbitrary binaries with no metadata and speculatively disassembles and lifts all parts of an executable that could be code. Because it does not necessarily find all pointers, SecondWrite includes a full copy of the original binary mapped at the original address to service read requests. The complexity of lifting to LLVM is why SecondWrite was only able to correctly transform a subset of SPEC CPU.

Two recent works, Uroboros [55] and Ramblr [54], implement binary reassembly. Their aim is to fully solve the disassembly problem, and lift a binary into a `.s` file which can be processed by a standard assembler. These are the first works to recognize the potential of layout-agnostic binary rewriting. However, since they are targeting arbitrary binaries, these systems must use complex and expensive speculative disassembly techniques, with no guarantee of success. They also provide no IR beyond flat generated assembly.

Relevant Binary Defenses We built several binary defense tools inspired by existing literature. One common late-stage attack vector is to re-enable code injection [43] (normally defended by OS protections), and we defend against this with our `W^X` Sandbox. Current code-reuse attacks repurpose instructions already present in a process's address space. For example, in Return-Oriented Programming (ROP) [48], the attacker builds exploit code piecewise using small sequences of code (gadgets) that end in return instructions. Randomization-based defenses try to foil such attacks by making the code layout unpredictable. We implemented in-place randomization [44], and a JIT-Shuffling continuous randomization technique (based on Shuffler [61] and TASR [8]). We also implemented debloating [2, 14], where unneeded code is removed from the program to improve security. Finally, control-flow integrity (CFI) is another code-reuse defense, where the target of every indirect control flow is validated [1, 31, 48]. Intel has described a new hardware extension called Control-flow Enforcement Technology [27, 28]; we implemented this CFI scheme in software.

3 Design

We designed Egalito according to the following goals:

- **Performance:** The framework should introduce near zero overhead, and therefore avoid using binary patching or address virtualization machinery.
- **Flexibility:** The framework should enable arbitrary code insertions and deletions without concern for address space layout (i.e., it should be layout-agnostic).
- **Deployability:** The framework should operate on ordinary (stripped) binaries, e.g., `.deb`/`.rpm` archives, without requiring special metadata or compiler flags.
- **Bootstrapping:** The framework should provide runtime support through (egalitarian) self-transformation.

The Binary Landscape Binary rewriting has historically been considered fragile: rewriting might work in some cases and fail in others. Traditional executables are stripped and position-dependent, and are very difficult to analyze. Recently, however, executables have begun to include more metadata by default. Most importantly, Linux distributions have migrated to using position-independent binaries over the past few years [18, 21, 39, 53]. Position-independent code (PIC) may be loaded at any base address, a feature used to implement shared libraries for decades, but enabled now for executables to strengthen Address-Space Layout Randomization (ASLR) defense [47]. Position-independent binaries contain more metadata, though not enough to make analyses straightforward—e.g., jump tables have no associated metadata. Yet, this shift to PIC is what enables our analyses.

Our analyses are sufficient to handle our target binaries, the types of ELF binaries that appear in current `.deb` or `.rpm` archives: position-independent, optimized, and stripped. However, our analyses are relatively straightforward and could become simpler over time. It is always possible for compilers to provide disassembly ground truth with extra metadata [33]. We hope Egalito will widen the demand for binary transformation and help compiler writers judge what metadata is helpful to include for binary analysis. Since we use some heuristics for jump table analysis, Egalito gives up on completeness (see Section 8). In exchange, we obtain a much more powerful and complete binary intermediate representation that represents the internals of a binary program.

Some binary rewriting techniques over-approximate the code (e.g., by disassembling from every offset in `.text` [6]). Others wait and discover the true extent of code at runtime (an under-approximation). We aim to statically uncover the precise set of code and control-flow in a program, so that we can generate standalone code without virtualization machinery. We also insist on avoiding binary patching, which means that we must find all references within a program; we cannot leave trampolines behind at original function addresses to catch errant calls through untransformed pointer values. Any virtual address in the input binary can be repurposed in the output. This recompiler output should look rather

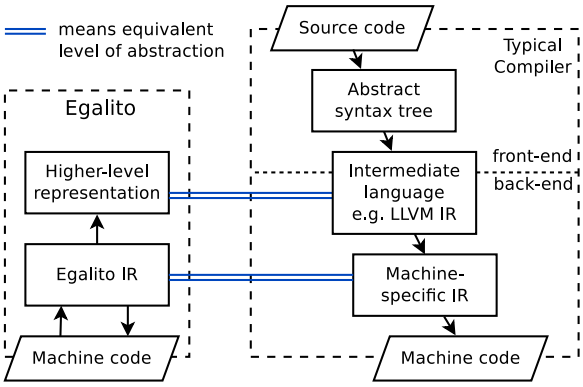


Figure 1. Egalito IR (EIR) design. Egalito reverses a compiler backend to obtain an IR similar to a machine-specific IR, and augments it with higher-level data structures.

closer to the output of a real compiler than that of a binary rewriter. There is no copy of the original `.text` to service reads and potentially provide security vulnerabilities; it is not overwritten with `hlt` instructions, but rather simply not included in our new ELF. Analysis details are in Section 5.

Egalitarian Capabilities In the tradition of bootstrapping compilers, we designed Egalito to be able to analyze and transform itself. This is possible in part because Egalito is written in a compiled language (C++). We provide a custom loader which analyzes itself, the executable, shared libraries, and Egalito, at load-time; it then bootstraps into a fully self-hosted environment where the only code present in the address space is code that Egalito has generated (and the `vDSO` kernel interface). This enables transformations that need dynamic analyses or runtime code-generation. As an example of this, we provide JIT-Shuffling (based on an earlier egalitarian defense, Shuffler [61]), which continuously generates its own code at new random addresses, with no undefended (fixed-address) code. For details, see Section 6. From a security perspective, egalitarian transformation can often enforce security isolation without requiring additional levels of privilege (e.g., kernel assistance).

Intermediate Representation The defining design decision of Egalito is its choice of intermediate representation (IR). There are many possible types of IR, which can be roughly categorized as follows, from front- to back-end in the compilation process: 1) abstract syntax trees, closely tied to the original source language; 2) intermediate languages such as three-address code or LLVM IR, used for optimization [35]; and 3) low-level machine-specific intermediate representation for code generation and peephole optimization (LLVM's `MachineInstr`, GCC's RTL). Figure 1 shows the basic structure of a compiler's intermediate representations. A recompiler consists of two pieces: first, the inverse of a compiler backend, to turn binary code into an IR; second, a forward mechanism to turn the IR back into binary code.

Intuitively, lifting machine code to the lowest-level IR is simplest, while higher-level IR provides more expressive analysis and transformative power. With Egalito, we reverse the compilation process only to the machine-specific IR level (see Figure 1), instead of all the way to an intermediate language like VEX or LLVM IR. Lifting to a higher-level intermediate language would typically turn each assembly instruction into multiple operations in SSA (single static assignment) form. The set of operations created for an instruction would not be tied together semantically, and might be modified, reordered, and optimized by existing infrastructure. Code generation would then become difficult, and likely diverge significantly from the input assembly. Hence, we deliberately chose a lower-level IR to make the output code generation more predictable and more in-line with the original input.

We do give up on reusing the substantial existing code that operates on established IRs. One of the main benefits of LLVM IR is its suite of existing analyses, optimizations, and backends. However, for low-level binary instrumentation or hardening, producing a differently-optimized version of the code is counterproductive. The transformations may be working around low-level issues (e.g., Spectre [32]) that are invisible to the optimizing framework; or transformations might add checks that the framework would rather optimize away, undermining a defense. Furthermore, intermediate languages are simply not designed for code modification and regeneration. For example, VEX (used by Valgrind [42] and angr [49]) uses a single representation for both relative and absolute references, making code regeneration difficult; inserting code in VEX also requires modifying the addresses of all subsequent instructions manually. Hence, we designed a custom IR with precisely the properties we need for layout-agnostic binary recompilation. Details follow in Section 4.

4 Intermediate Representation (EIR)

Our Egalito IR (*EIR*) is a C++ class hierarchy that can be viewed as a complete abstract syntax tree of the ELF input binary. It stores instruction encodings placed in architecture-independent categories (e.g., `ControlFlowInstruction`). It also stores semantic information that we recover about an ELF, such as jump tables and control flow. EIR has two major innovations compared to typical binary rewriters.

First, in a departure from normal parse trees, we abstract addresses into *links*, and then store both addresses and links. The addresses in each tree node allow EIR to represent an original ELF precisely, and enables minimal changes when performing ELF-to-ELF recompilation. Meanwhile, links are key to supporting layout-agnostic user modifications, since targets may be resolved even if addresses are reassigned or if new code is inserted. In the latter case, assigned addresses may overlap until they are recomputed during a "linking" step that moves nodes such as basic blocks to new non-overlapping addresses. Egalito tools can rearrange addresses directly, or rely on the framework to generate non-overlapping addresses after modifications.

Second, we extend EIR with high-level use-definition/def-use data structures. These data structures are read-only and ephemeral, meaning they are only valid for a specific EIR state but can be recreated at any point from EIR. They provide access for analyses possible only on higher-level IRs, such as our own jump table analyses, while EIR remains low-level to enable efficient code regeneration. EIR is the canonical representation on which all modifications must take place.

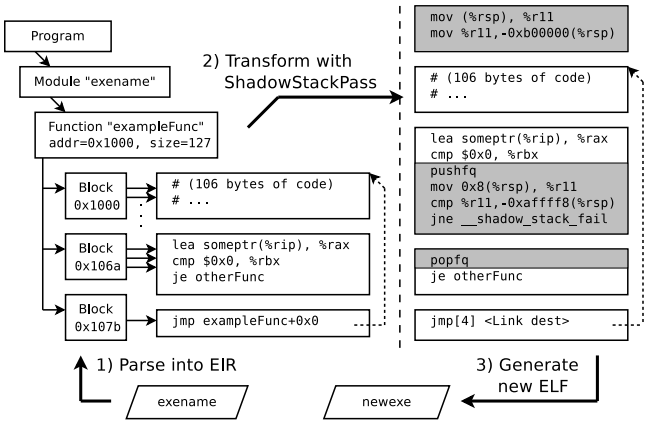
4.1 Shadow Stack Transformation Example

In Figure 2, we show an example transformation tool that adds a shadow stack to ordinary x86_64 binaries. This shadow stack is located at a constant offset (`-0xb00000`) from the real stack [13, 16]. The majority of the code is written in one recompiler pass, shown in Figure 2b. The code is simplified for brevity. Our full implementation creates `__shadow_stack_fail` with a single `hlt` instruction, and allocates the shadow stack memory region by adding a call at program start.

A recompiler pass is a `Visitor` as in the *Visitor design pattern* [22], able to access EIR nodes at any level of granularity by implementing `visit` functions. In this case, we visit each `Function` and add code in its prologue to save the return address to the shadow stack. We recurse on all (`Block`) children of the `Function`, and the default `Block` visitor recurses on all of its (`Instruction`) children. We visit each `Instruction` and look for ones that leave the function: returns, external (tail) jumps, or indirect jumps/calls. At each exit point, we insert code to verify the return address against the saved shadow stack value. The inserted code modifies the flags register (`%rflags`), so we ask Egalito to save it if necessary—`stackAdded` indicates how many bytes Egalito pushed to the stack, and we emit code appropriately.

In the `main` function, we parse an input ELF file, run two passes, and then generate an output ELF. The first pass is the one in this example, while `ReassignAddresses` chooses non-overlapping addresses for all code—necessary since we insert instructions and increase the size of blocks and functions.

Figure 2a shows an example EIR tree structure, and the code transformations that `ShadowStackPass` performs. Its seemingly simple code insertions trigger operations of significant complexity. First, notice the `forceSameBlock` flag in the code. This controls whether instructions inserted at the very beginning of a block should be part of the same block (i.e., whether incoming jumps will run the new code). The initial shadow stack save uses `forceSameBlock=false`; it should execute only once when the function is first invoked. If any jumps target the first instruction in the block, Egalito creates a new earlier basic block for inserted instructions (as happens with the `jmp` in the last Block in Figure 2a). Conversely, incoming jumps to a block containing an exit jump should run the new code to check the return address



(a) EIR transformation example of adding a shadow stack. Input code is 1) disassembled, 2) transformed, and 3) regenerated.

```
class ShadowStackPass : public RecompilerPass {
public:
  virtual void visit(Function *function) {
    Instruction *instr1 = function->getChild(0)->getChild(0);
    Mutator::insertBefore(instr1, /*forceSameBlock=*/ false,
      std::vector<Instruction *>{
        ASM("mov (%rsp), %r11"),
        ASM("mov %r11, -0xb00000(%rsp)") });
    recurse(function);  // RecompilerPass::recurse
  }

  virtual void visit(Instruction *ins) {
    if(ins->isType<Return>()
      || (ins->isType<ControlFlow>() && ins->isExternalJump())
      || (ins->isType<IndirectJump>() && !ins->forJumpTable())) {

      // We clobber the RFLAGS register; Egalito saves it if
      // necessary, pushing stackAdded bytes onto the stack.
      auto addCheckLambda = [] (int stackAdded) {
        auto failFunc = Find::function("__shadow_stack_fail");
        return std::vector<Instruction *>{
          ASM("mov " << stackAdded << "(%rsp), %r11"),
          ASM("cmp %r11, " << -0xb00000+stackAdded << "(%rsp)"),
          ASM("jne " << new RelativeLink(failFunc)) };
      };
      Mutator::insertBefore(ins, /*forceSameBlock=*/ true,
        AddRegisterSaving({X86_REG_RFLAGS}, addCheckLambda));
    }
  }
};

int main(int argc, char *argv[]) {
  assert(argc == 3);  // usage: ./shadowstackify input output
  EgalitoInterface egalito;
  egalito.parse(argv[1]);
  egalito.getRoot()->accept(ShadowStackPass());
  egalito.getRoot()->accept(ReassignAddresses());
  return egalito.generate(argv[2]);
}
```

(b) C++11 code for shadow stack transformation.

Figure 2. Adding a constant-offset x86_64 shadow stack.

before exiting. With `forceSameBlock=true`, Egalito reuses the same basic block for insertions. Links that originally targeted the first instruction in the block are automatically updated in constant time to point to the new first instruction.

The original `jmp` instruction had a one-byte displacement, since its offset was `-127` (the size of `exampleFunc`); the assembler uses the shortest possible encoding. However, after inserting some intervening code, a (signed) one-byte displacement no longer reaches the target. Egalito automatically re-encodes one-byte jumps that no longer reach their target to use 4-byte displacements in the `PromoteJumps` pass. Since one jump promotion can cascade and cause others to need promotion, this pass runs iteratively until a fixed point.

Next, consider the registers used by this example. We use `%r11` as a temporary register, but do not ask Egalito to save this register, because `%r11` is callee-saved and may be overwritten by a function call. We do ask Egalito to save the flags register `%rflags` which is overwritten by our `cmp` instruction. `%rflags` is not expected to be preserved across function calls, but it must be preserved across conditional tail recursion (Figure 2a shows such an example with `je`). Egalito finds all jumps that perform tail recursion. Often, transformations will need to handle these cases specially, e.g., to prevent tail recursion from causing two shadow-stack pushes in a row without an intervening pop. In this simple shadow stack, however, pushing (a memory write) is idempotent.

Egalito saves registers with `push`/`pop` instructions, and can analyze a function to see if register saves are really necessary (i.e., the value is used by some successor instruction). Egalito will also identify whether a function uses the red zone instead of a stack frame: leaf functions on x86_64 may access 128 bytes beyond the top of the stack in lieu of moving `%rsp`, which is slightly more efficient, but means that an inserted `push` will overwrite actual program data. Hence, Egalito may automatically add up to 128 to the stack pointer before saving registers. In our example, we save `%rflags` which involves adding 8 bytes to the stack pointer, and this information is passed to the code-generation lambda so it can adjust its offsets. Egalito, and user tools, can also inject additional global data, thread local storage, or `%gs` variables.

5 Binary Analysis

We use several analyses to recover static control-flow graphs and obtain complete and precise disassembly. We focus on modern Linux binaries, which are position-independent [18, 21, 39, 53], optimized, and stripped. See Section 8 for limitations. Our focus on PIC binaries is unlike most related works: most focus on position-dependent binaries, and conversely, some rely on additional compiler flags (e.g., `-Wl,-q` and `-ffunction-sections`) for extra metadata [46, 61].

5.1 Disassembling Code, Not Data

Binary analysis in general is undecidable [58]—e.g., classifying bytes as code or data after a loop is equivalent to the halting problem. The standard technique for binary rewriting is recursive disassembly, which gives *sound* but *incomplete* results. Many tools conservatively overapproximate—in this example, treating the bytes both as potential code (if they disassemble without errors) and also as data, leaving the original `.text` section in place. Dynamic binary translation

can delay the decision to runtime, and only treat bytes as code when when a jump to the memory is actually observed.

Modern binaries separate code and data sections, and refrain from using embedded constants. x86_64 code has no need since RIP-relative literal loads can reach +/- 2GB, and ARM64 constant pools (`-mpc-relative-literal-loads`) are disabled except under the tiny memory model [15] (address space ≤ 1MB). This is consistent with prior work which finds that while embedded constants are present in real-world x86 Windows binaries [58], all GCC- and Clang-generated binaries can be linearly disassembled on x86 [4]. Thus, linear disassembly of code sections will suffice.

5.2 Reconstructing Functions

Egalito operates on stripped binaries, where function boundaries are not specified. We approximate function boundaries with a coarse-grained heuristic based on direct call targets, which may conservatively lump functions together. We then analyze jump tables. The final control flow graph of each function is split into disjoint connected components to accurately reconstruct function boundaries. There is one further special case: non-returning functions. After a call to a function that never returns, the compiler will place the next basic block immediately afterwards, knowing execution will never fall-through. GCC tracks functions like `exit` with an attribute `noreturn`, recursively propagating it to functions that always call `exit` etc; Egalito uses a similar analysis.

Frame unwind information, created for C++ exceptions and debugging, is sometimes present. When it is, we use it to precisely identify function boundaries. On ARM64, this info is only present for functions that throw exceptions; on x86_64, stripped binaries contain frame unwind information for every function (`-funwind-tables` is enabled by default).

5.3 Identifying Code Pointers

A binary is full of values that might be constants or might be pointers. It is not sufficient to simply consider all values to be pointers if they lie within the valid range for code virtual addresses—even SPEC CPU is disassembled incorrectly under such a heuristic (§6.1 of [55]). In PIC, all absolute pointers will have relocations [7]. Relative pointers in the data section are typically used only used for jump tables, covered next. Relative pointers in x86_64 code sections are a `%rip`-relative constant in a single instruction, and are easy to identify.

The situation is more complicated on RISC architectures such as ARM64, however. A single fixed-length instruction cannot encode a PIC reference. The compiler uses: 1) an `adrp` instruction to load the upper bits; and 2) an `add`, `ldr`, or `str` instruction to load the page offset. These instructions form a PC-relative load, so there is no relocation metadata. Furthermore, the compiler's optimizer may place these logically related instructions far away from one another, in different basic blocks, hoisted outside loops, etc. We leveraged data flow analysis to find such split-pointer loads.

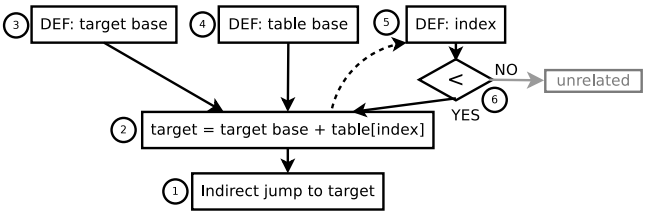


Figure 3. Jump table reconstruction steps.

5.4 Reconstructing Jump Tables

There is no metadata which can assist in locating jump tables (no relocations in PIC). Standard ARM64 jump tables can use 4-, 2- or even 1-byte offsets—but ARM64 does not even define standard relocation types for 1-byte values. Nevertheless, we aim to recover all jump table addresses, invocation locations (indirect jumps), and table bounds (number of entries). Although bounds checks may be optimized away by the compiler, determining the bound is essential: if it is underestimated, some edges in the function's control flow graph will be unidentified, while if it is overestimated, arbitrary data will be corrupted during recompilation.

Detection Procedure Our solution leverages sophisticated data analysis techniques including use-def chains. We will use Figure 3 as a running example. First, we consider every indirect jump in the program ①. We look for expressions that flow into the jump computation ②, pattern-matching against structures that the compiler uses to implement jump tables. These patterns are independent of exact instructions, registers, operand order, flow through repeated `mov`s, basic block structure, etc. We extract the address of the sequence of table entries (the table base ③), and the value added to each table entry to compute its destination (the target base ④, same as table base on x86_64). Finally, we look for the table bound. We extract the indexing register or memory expression, find the definition of its value ⑤, and iterate over all uses of this value. One or more uses will flow back to the jump instruction, and there may be bounds checks along those paths ⑥. We select the tightest comparison bound.

Bounds Not all bounds are enforced with a straightforward comparison instruction, however. We implemented many special cases, including: 1) subtraction/bitwise test, then check flags against zero; and 2) bitwise `and` with a constant bound (e.g., for hash computations). We also encountered sophisticated tables which we call multistage jump tables: one table is indexed into to determine an index within a second table, which finally contains a target address. We handle this by parsing the first table as usual, with the correct striding. We examine every value in the table and find the maximum index—thus deducing the bound for the second-stage table.

Adjacency Heuristic In some cases, we simply cannot determine the bound, and we use the following heuristic. We observed that current compilers (GCC and Clang) place all

jump table contents sequentially. We expand each table as much as possible without including entries whose computed target lies outside the source function, and stop at the end of the section or when another link occurs. This heuristic can fail in real-world cases such as tables that partially overlap each others' data (e.g., `glibc` hand-coded assembly, which luckily contains explicit bounds checks), but it is a useful fallback. Our evaluation shows that across thousands of jump tables, the adjacency heuristic is only used 6.52% of the time. Considering that not all jump tables even include a bounds check, we believe Egalito does very well in its analyses.

6 Egalito Tools

Nine tools follow, ordered from simple to sophisticated.

Counter-Based Profiling This tool instruments EIR (functions or basic blocks) with counter increments. Each `Chunk` is given a separate global variable, and its Egalito name is written into a data section. Counter values are appended to a binary file at program exit. We provide a `gprof` look-alike, which prints accumulated statistics from past runs.

Profile-Guided Optimization We implemented a profile-guided optimization tool which modifies a program's layout for the best performance given knowledge about the input. Given function-call counts from a representative execution, recorded by our profiling tool, this tool arranges functions from most common to least for better caching performance.

Debloating We implemented a function-level debloating [2, 14] tool for x86_64 and ARM64. Starting from the program entry point and from every function whose address is taken, this tool iterates over our control-flow graph and finds all reachable code. Any unreachable functions are removed, as they represent bloat and cannot be called (barring reflection).

Instruction Reordering In-place randomization [44] consists of four techniques, of which we implemented two: instruction reordering within basic blocks, and reordering register saves/restores. We use dataflow information to create a graph of all instruction dependencies within each basic block. We then choose a new order for the entire block, maintaining a set of valid next instructions at each point and selecting one at random. Note that this algorithm, while simple, does not guarantee uniformly random permutation selection. Within function prologues and epilogues, we reorder register saves/restores by eliminating the ordering constraint between `push`/`pop` instructions that would otherwise hold (due to contention on `%rsp`). In the prologue, we choose a new register save order; in each epilogue, we restrict the ordering between pops to follow the reverse of the save order. Non-`push`/`pop` instructions may still be randomly intermingled as dependencies allow.

Data Execution Prevention On x86_64, we wrote a sandbox to enforce `W^X` memory: no memory page may be writable and later executable. We find all `syscall` instructions, leveraging dataflow analyses to deduce the system call number (`%rax`). We instrument `mmap`, `mprotect`, and `munmap` to track whether each mapped memory page has ever been writable. If the program tries to make such a page executable, this constitutes a sandbox violation and the program is terminated. We combine this sandbox with our control flow integrity from CET. The control flow integrity prevents an attacker from jumping over the system call instrumentation, while the data structure that tracks writable pages cannot be reliably corrupted (located at a random address, only referred to by `%gs`). Hence, this sandbox prevents any code-injection attack, even if the attacker can corrupt data and call `mprotect`.

Retpolines The recent Spectre [32] vulnerability exploits a hardware bug, an inconsistency in the way current CPUs (Intel, AMD, and ARM) implement speculative execution. One available software fix for Spectre Variant 2 is to transform all indirect jumps into *retpolines* [52], which force speculative execution into safely contained infinite loops. We created an x86_64 Egalito tool that transforms every indirect jump into a retpoline. We outline retpolines to avoid duplicating their code, generating a new function for each unique indirect target expression (e.g., `%rax`, `0x10(%rax,%rbx,2)`, etc).

Software Implementation of Intel's CET Intel has recently announced an x86_64 hardware extension called CET or Control-flow Enforcement Technology [27, 28]. This extension specifies a set of hardware instructions that will be made available in future CPUs, and map to no-ops on current processors. CET consists of a) control-flow integrity (CFI) for indirect branches, and b) a hardware shadow stack to protect return statements. We implemented this defense in software in August 2018 when the `libstdc++.so` in Ubuntu 18.04 began to include `endbr64` instructions (but other libraries did not). Our tool may be applied comprehensively across a system directly to the binary code, and when hardware support becomes available, the instrumentation can be removed.

Under CET's CFI scheme, the target of every indirect control flow (call or jump) is marked with a specific instruction `endbr64`. Our CFI pass iterates over each function whose address is taken and adds `endbr64` to its prologue (if not already present). Finally, we instrument each indirect call with a runtime check to verify that its target is an `endbr64`. If not, our error handler raises a fatal `SIGILL` signal.

We developed two shadow stack implementations. The first is the simple constant-offset shadow stack used as an example in Section 4.1. The second, a more faithful reproduction of CET, stores the shadow region at `%gs` with a shadow stack pointer at `%gs:0x0`. Pushing and popping involves incrementing/decrementing this pointer, which means that a push without a corresponding pop will cause a detectable fault. CET also specifies hardware instructions that modify the top-of-stack pointer directly, for stack unwinding and exception handling. We leave these for future work.

Egalito-AFL Egalito-AFL is a binary-level backend for the AFL fuzzing framework [63]. It adds the instrumentation necessary for AFL to determine coverage (i.e., the set of branches that are taken). We integrate with the AFL forkserver, which forks new targets to avoid `exec` calls. The forkserver creates a System V shared memory segment and passes it to the initial target via the `__AFL_SHM_ID` environment variable. We inject code at program start to detect this and map a 0x10000-size memory region with `shmat`.

We instrument every basic block with a coverage-recording snippet based on `bin_coverage.c` from drAFL [50]. Each block is assigned a random (constant) ID, and an accumulator tracks history of the past few branches, hashing into the shared memory region [26]. Each time, the accumulator is right-shifted by one, and XOR'd with the random block ID; the index in the shared memory region corresponding to the lower 16 bits of the accumulator is incremented. We perform this update using only a single register (plus `%rflags`).

Just-In-Time Shuffling JIT-Shuffling is a novel continuous code randomization defense, based on prior work (Shuffler [61], TASR [8]). Like Shuffler, JIT-Shuffling is x86_64-only and transforms every code pointer into an index within a runtime dispatch table. Return addresses become a pair of numbers, a function index and a byte offset into that function. The table is stored using the unused `%gs` segment register [5, 61], to prevent an attacker from performing memory disclosure on the table. Direct function calls, tail recursive calls, indirect function calls, returns, etc, are replaced with `%gs`-relative jumps, while pointer initializations are changed into indices—addresses are never used as code pointers.

In a departure from Shuffler, JIT-Shuffling operates synchronously. Function `%gs`-table entries initially point to the address of an Egalito *resolver* function that instantiates the function at a new address, similar to the way a lazy PLT resolver computes addresses on the fly. Periodically, a "reset" callback erases all functions, and points their table entries back to the Egalito resolver. If control flow returns to a function which has been erased, it will be reinstantiated. As in Shuffler, JIT-Shuffling makes use of two code sandboxes during execution, and migrates between them while leaving no fixed code in the address space—even Egalito code. JIT-Shuffling supports `fork` and multiple threads: each execution context uses its own sandboxes (threads share EIR).

7 Evaluation

Our evaluation uses the machines in Figure 4. Unless otherwise noted, we used M1 for x86_64, and M8 for ARM64.

7.1 Correctness of Binary Analysis

Function Boundaries On ARM64, we transformed all 105 GNU Coreutils binaries (stripped). Egalito identified all function boundaries; however, in 11 cases we split the code into additional functions, when the (non-returning) `error()` function is called with a constant argument. (Separating such functions is an accurate representation of control flow.)

ID	Arch	Linux Distribution	Machine	RAM	GCC
M1	x86_64	Debian buster	4c/8t i7-4770	32GB	7.2.0
M2	x86_64	Debian stretch 9.6	8c/16t X5550x2	24GB	6.3.0
M3	x86_64	Debian testing	4c/8t i7-2600	16GB	8.2.0
M4	x86_64	Devuan ascii	8c/8t W-2145	64GB	6.3.0
M5	x86_64	openSUSE*	4c/8t i7-4770	32GB	7.3.0
M6	x86_64	Ubuntu 18.04.1†	6c/12t X5550x2	10GB	7.3.0
M7	x86_64	Fedora 31	8c/16t X5550x2	24GB	9.2.1
M8	ARM64	openSUSE Leap†	8c/8t ThunderX	8GB	7.1.1
M9	ARM64	openSUSE*	Raspberry PI 3	1GB	7.2.1

Figure 4. Machines used for Egalito testing and evaluation. *=openSUSE Tumbleweed rolling release. †=Virtual Machine.

Code and Data Pointers We validated that Egalito can detect all pointers using relocation ground truth. We compiled GNU Coreutils and `glibc` on both x86_64 and ARM64 with `-ffunction-sections` and `-Wl,--emit-relocs` (`-Wl,-q`), to include as many relocations as possible in the output. Egalito creates links for precisely the set of code and data pointers in the ground truth (plus additional links for jump table entries, which have no relocations).

Inline Assembly Egalito handles many assembly functions correctly, e.g. those in `glibc`. However, some hand-coded assembly (`libffi`, crypto code) embeds jump table values into `.text` symbols. By design, Egalito trusts function boundary metadata, but to handle non-standard cases we provide an override settings file for users to define code/data boundaries. Our evaluation does not rely on any such parse overrides.

Jump Table Study We verified Egalito's detected jump tables against ground truth obtained from the compiler, using GCC's `-fdump-final-insns`, which outputs the RTL intermediate language while producing the corresponding object files. This data includes the number of jump tables per function as well as the number of entries in each table. On both x86_64 and ARM64, we programmatically verified jump tables in `glibc` and Coreutils. We manually verified `glibc` jump tables written purely in assembly.

To test jump table detection at scale, we ran a large-scale experiment on x86_64 Debian packages. We built each package from source with the `dpkg-buildpackage` option `DEB_CFLAGS_APPEND=-fdump-final-insns`. We preserved every `.o` object file created during the build process by overriding `rm` and various variants of `gcc`. Finally, we extracted the built package (and its debug package) and found all executables and libraries therein. We used `FILE` symbols to map functions back to object files and hence to jump table information. For functions not within a `FILE`, we searched for functions with the same name in all object files. We also considered function aliases and `*.lto_priv.NN` symbols generated by link-time optimizations. We analyzed each case with Egalito and compared our list of recovered jump tables with the

ground truth. In case of multiple ground-truth options (multiple functions of the same name), we considered Egalito to be correct if it matched one option.

Overall, we ran the experiment on 172 packages from Debian's popcon list [45] on M3. We excluded large packages like `systemd` and packages that would likely not contain C/C++ executables. These 172 packages produced 867 executables and shared libraries, and in 866 cases, Egalito successfully reproduced the ground truth jump table list. (Parsing these 867 ELFs took Egalito 55 minutes on a single core.) The single failure is `sftp` which contains heavily nested jump tables that our control-flow graph does not yet capture. Hence, Egalito correctly recovered jump tables 99.9% of the time. Nearly half of all executables included complex stack-variable bounds checks that required tracking data flow through memory, not just registers. Of the 3970 recovered jump tables, Egalito analytically determined table bounds in 93.48% of cases (relying on the table adjacency heuristic in the remaining 6.52%). Since not all jump tables even include a bounds check, Egalito's analyses do very well.

7.2 Correctness of Binary Transformation

To gain confidence in Egalito's generated code, we executed and successfully passed the test suites for Coreutils, `ffmpeg`, and `sqlite` on M2 (`sqlite` includes 2,583,067 tests). We manually tested 13 arbitrary programs; many succeeded, including `dpkg`, `make`, `tmux`, and `vim`, while 4 failed. Two failed due to some position-dependent code linked in with PIC, one failed due to Egalito features not yet implemented (`aptitude` throws an exception), and Google's V8 contains embedded data after every function (within symbol boundaries).

Debian Package Tests We ran a second large-scale analysis on Debian binaries, running test suites associated with Debian packages on Egalito-transformed binaries. We found that some 9904 Debian packages are registered in the Debian continuous integration system, and we targeted the 308 packages that have the tag `implemented-in` set to `c` or `c++`. We transformed all executables in the distribution's `.deb` files with Egalito, then ran the package tests in a `chroot` with `autopkgtest`. 149 packages contain tests and build correctly in the `chroot`. 90 out of 149 (60%) packages have *all* tests pass. 38 failures can be detected by Egalito, they are because we do not yet support generating symbol versions (this merely requires additional engineering effort). At least one throws an exception (not supported), and one has a too-strict test that looks for sequences of zeros in the executable. If we take this into consideration, 90/109 (82.6%) of the packages pass all tests (4 additional packages pass at least one).

Compiler Versions To test binaries from different compilers, we built many versions of GCC from source. With each GCC, we verified that `sqlite` (compiled position-independent) still passed all tests. Specifically, we tested the following versions of GCC (earlier compilers no longer built cleanly): 4.9.4, 5.3.0, 5.4.0, 5.5.0, 6.1.0, 6.2.0, 6.3.0, 6.4.0, 6.5.0, 7.1.0, 7.2.0, 7.3.0, 7.4.0, 8.1.0, 8.2.0. We tested Clang 5.0.1 by transforming and successfully running SPEC CPU ref on M5 (loader mode).

Linux Distributions We verified that Egalito-transformed SPEC CPU ref runs correctly on four different distributions: Debian (M1), Ubuntu (M6), openSUSE (M5) and Fedora (M7). (All in 1-1 ELF mode except M5, where we used loader mode).

Go Binaries Egalito cannot transform Go binaries correctly, because they contain `vtable`-like structures in `go.itab.*` without relocations. We would need to represent these data structures in EIR in order to transform Go programs.

Chromium Libraries V8 and Chromium contain an unusual form of DWARF that we do not support. However, we transformed Chromium's shared libraries in 1-1 mode. Of 177 libraries, 59 use symbol versions in a way we do not support (a low-level ELF generation, solvable with engineering effort). Of the 118 remaining, 12 cause Chromium to fail, and `libffi` fails disassembly due to embedded jump tables in assembly code. 105 of 118 (89%) transform correctly and using `LD_LIBRARY_PATH` to load all transformed libraries, Chromium can browse to news sites and display videos.

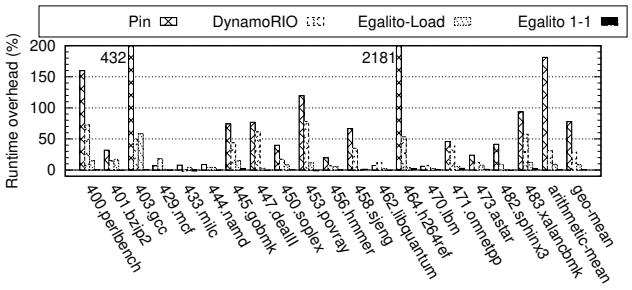
`/usr/bin/` smoke test We tried to transform all the executables in `/usr/bin/` on M4 (1-1 mode), then ran each program with `--help`, comparing against baseline. There were 1379 executables; 90 (6.5%) failed during transformation because they were position-dependent, and 11 failed transformation for other reasons (Egalito was aware of a problem). Of the remaining 1278 executables, 1256 (96.6%) produced identical output and exit code. Of the remaining 33 failures, 25 were due to lack of `RUNPATH` support, and 8 from fatal signals.

Kernels We transformed a Fuchsia [23] microkernel (for ARM64) with Egalito. In lieu of PIC, we leveraged full relocations with the `-q` linker flag. We added function call logging into a ring buffer, generated a flat binary image, and successfully booted it on a bare-metal Raspberry PI 3 (M9).

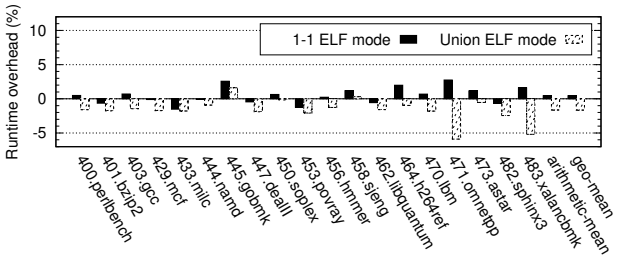
7.3 SPEC CPU Performance Evaluation

Egalito supports three major execution modes: 1-1 ELF, union ELF, and loader mode. We present Egalito's performance on SPEC CPU in each mode, utilizing binary optimizations discussed in Section 7.4. In 1-1 ELF mode, we transformed only the main executable and not shared libraries; in union ELF and loader mode, we transformed all code. Since Egalito does not yet support C++ exceptions, we modified SPEC CPU by replacing exceptions with conventional control flow in `omnetpp` (20-line change) and `povray` (15 lines). We also fixed a compile error in `soplex` (1 line) in recent GCC versions.

Comparison with DynamoRIO and Pin As shown in Figure 5a, Egalito has much better performance than existing DBT-based tools DynamoRIO and Pin (measured on M1). DynamoRIO geo mean overhead is 28.8%, Pin is 77.7%,



(a) Comparison of DynamoRIO, Pin, Egalito (loader and 1-1 mode).



(b) Performance of Egalito-generated ELF outputs.

Figure 5. Egalito runtime overhead on SPEC CPU 2006, in different modes, compared with DynamoRIO and Pin.

while Egalito in 1-1 mode is only 0.46% (executables pre-transformed). For a fairer comparison, Egalito in loader mode also parses all executables at load-time, incurring 8.7% slowdown (1.1% slowdown with caching; see loader mode below).

1-1 ELF Mode This is Egalito's default mode, reading in a single ELF and outputting a single ELF. Since EIR represents and recreates each input instruction, we can expect near zero performance overhead (given a no-op transformation tool). We measured this baseline overhead in 1-1 ELF generation mode by running SPEC CPU 2006 on machine M1, and results are shown in Figure 5b. Egalito incurs only 0.46% overhead.

Union ELF Mode In union ELF mode, Egalito combines the input executable and all its shared libraries into one output ELF, essentially transforming a dynamically linked program into a statically linked one. We measured the baseline overhead in union ELF mode (also on M1), and Egalito achieves a 1.7% geo mean *speedup*. We use original function order and 2-byte alignment as in 1-1 ELF mode; the speedup is because union ELF mode collapses PLT calls into direct calls.

The memory and space overhead of Egalito-generated ELF files is minimal. On SPEC CPU, union ELF outputs were on average only 0.44% larger (low variance) than the sum of input ELFs. At program entry, the mapped memory of union ELFs is 79%-95% (average 88.4%) of baseline. At program exit, file-backed resident memory use is 152KB-1.25MB less (598KB average). So union ELFs are memory efficient, but will not share library code pages with different programs. Code sections, augmented with library code, are 540KB-2.56MB larger (1.7MB average)—an upper bound on memory wasted.

Egalito Tool	Mode	Overhead	Worst case
No-op 1-1 mode	1-1	0.46%	2.8% omnetpp
No-op union mode	union	-1.7%†	1.6% gobmk
No-op loader mode	loader	-1.4%†	2.6% povray
Profiling	1-1	0.16%, 1 fail	3.3% povray
Profile-guided opt	1-1	-1.0%†, C only	3.2% h264ref
Retpolines	1-1	6.9%	63.0% povray
CET CFI only	union	-1.4%†	-0.57%† gobmk
CET CFI + const stack	union	4.4%	22.2% povray
CET CFI + %gs stack	union	9.8%, 1 fail	40.7% povray
Instruction reordering	loader	-2.7%†	3.8% povray

Figure 6. Geometric mean overhead of Egalito tools (loader mode has load time excluded). †=performance speedup.

Loader Mode Next, we investigated Egalito's baseline overhead in loader mode. The unoptimized raw overhead of the Egalito loader is 8.7% geo mean; using HOBBIT files (a binary serialization of EIR: Hierarchical Object Built for Binary Transformation) gives 1.1% geo mean overhead. Much of this is load-time overhead, spent parsing ELF files (9.4 seconds average, max 15.9) or loading HOBBIT files (2.8 seconds, max 4.1). Also, the SPEC CPU harness invokes the target more frequently in certain cases—e.g., gcc is invoked nine times per run while some binaries are only invoked once. By subtracting all load-time cost (amortized in long-running programs), we see a 1.4% geo mean speedup. (Some speedup is again expected due to collapsed PLTs.)

7.4 Binary Optimizations

Function Padding The compiler uses 16-byte padding between functions, but we try 16-, 8-, 4, and 2-byte padding. (1-byte function padding interferes with exception handlers; the least significant bit is set to indicate thrown exceptions.) Compact code may utilize instruction caches better, so Egalito defaults to 2-byte padding. The best padding varies according to test case and machine; for in-depth analysis see [62]. On M1 2-byte padding is 0.56% slowdown but 8-byte is best at 0.53% speedup. Selecting the best case for each SPEC program gives 1.2% geo mean speedup with 9.6% speedup in the best case. On M2, 2-byte is fastest at 0.59% speedup.

PLT Collapsing To support dynamically linked libraries, calls from one library to another go through indirection via the Procedure Linkage Table (PLT). But in union ELF or loader mode, Egalito knows all the code that will be executed. So we collapse PLT calls into direct calls, and place code from all libraries into a single text section, essentially transforming a dynamically linked program into a statically linked one.

7.5 Egalito Tool Performance

Egalito is 51759 lines of code (determined by sloccount [17]); each tool is between 157-306, except JIT-Shuffling with 1510.

Tools on SPEC CPU Figure 6 shows the performance of several Egalito tools. Retpolines on M1 has geo mean overhead

6.9%, in line with other published numbers (e.g., Bullet [34]). However, it is prohibitively expensive for `povray` and `xalan`, which use large numbers of virtual function calls and hence incur the indirect-jump overhead repeatedly. The instruction reordering tool observed a speedup over the baseline, likely noise from loader mode. In our CET tool, the simpler constant-offset shadow stack is more efficient than `%gs`. The `%gs` shadow stack fails in one case (`gcc`) due to a bug with our conditional tail recursion detection. Profile-guided optimization, trained with call counts on "test" and evaluated on "ref", shows a 1.0% speedup (best: 11.8% speedup for dealII) [62].

Egalito-AFL We measured the fuzzing throughput of Egalito-AFL, in comparison with a DynamoRIO-based fuzzer called drAFL [50]. Both tools operate directly on binaries, unlike the original AFL which requires source-level instrumentation with an appropriate compiler. We fuzzed `readelf` and `libpng` on M2 for 5 minutes, and Egalito-AFL obtained a 15.6x and 64.2x speedup respectively over drAFL; when run for 10 minutes, these speedups became 18.0x (60060 vs 3353 executions) and 61.4x (526149 vs 8566). Egalito-AFL is 18-61x faster because 1) Egalito-AFL outputs one ELF binary, while drAFL runs DynamoRIO to rebuild the code cache for each execution; and 2) Egalito-AFL integrates with AFL's forkserver (drAFL does not), allowing minimal `exec` syscalls.

Just-In-Time Shuffling To evaluate JIT-Shuffling, we defended Nginx 1.11.3 on x86_64 (M5). We tested four workers serving ten concurrent clients from the `wrk` tool over one minute, for 612B, 100KB, and 1MB requests. Results in multithreaded and multiprocess mode are nearly identical.

We first erase all functions after each Nginx HTTP request, as in TASR [8], but this is prohibitively expensive (5-50x). TASR advocates this design but does not measure its performance (Shuffler [61] would have 1.5% throughput with 100KB requests). We can trade performance for security by resetting less frequently; instead of after every request, we can reset after every 10, or every 100, etc. Nginx achieves 50-90% of the original throughput when resetting after every 100 requests. This policy is still orders of magnitude ahead of leakage attacks like Blind ROP, which requires a total of 11070 Nginx requests [9]. It also results in re-randomizing every 1.8-4.9 ms, much faster than Shuffler's 50 ms interval.

7.6 Comparison with Other Frameworks

Ramblr [54] We installed Ramblr on M2, and tried to transform `/bin/ls` with the simplest `stackretencryption` reassembler backend. It failed during code generation, as did a) hello world, b) hello with `-no-pie`, c) hello with `-static`, d) programs without `glibc` (complained about empty input).

Multiverse [6] On a 32-bit VM, `pwntools` dies (dependency). We tested further on a 64-bit system (M3). Multiverse does not work on position-independent executables (generates invalid ELF headers), statically linked executables, or `glibc` (jump mapping table contains an invalid offset). On hello world compiled with `-no-pie`, `_start` is transformed, but the RIP-relative `__libc_start_main` pointer is unmodified and the code runs the original `main`. By mapping the original `.text` with executable permissions, Multiverse allows the original code to execute. There are also several `RWX` memory regions in which Multiverse writes and then runs code, opening the door to code injection attacks.

PSI [64] PSI is distributed on a 32-bit Ubuntu 12.04 VirtualBox VM. We measured the runtime of Egalito (M2) and PSI (VM) against baselines on their respective machines. On `zip`, Egalito had -3.07% overhead vs PSI's 8.26%; on `python`, 3.33% vs 44.6%; on `perl`, -13.6% vs 108%; on `vim`, success vs crash. PSI's higher overhead is due to its address virtualization.

8 Limitations and Discussion

Egalito works across many binaries and architectures, but it:

1. requires inputs to be position-independent code (PIC) for sufficient metadata (discussed in Section 3);
2. uses data-flow analysis techniques that infer missing metadata (see Section 5.4), and so is not complete and cannot guarantee full disassembly;
3. cannot handle obfuscated code, nor inline assembly which embeds data—like jump table values—into `.text` symbols (discussed in Section 7.1);
4. does not yet support some implementation-level features (described in Section 7.2), such as C++ exceptions and atypical metadata (e.g., Go binaries or V8).

64-bit RISC-V support is well underway in Egalito: we parse into EIR with a different disassembly library, and successfully analyze jump tables, but do not generate code. We are extending Egalito to Windows, where newer executables include relocations for base address randomization [40].

9 Conclusion

We have presented the Egalito recompiler, a layout-agnostic binary rewriting framework. We implemented nine tools with Egalito including a novel defense JIT-Shuffling, an AFL backend, and a software version of Intel's CET. Egalito is very efficient, observing a substantial performance speedup of 1.7% on SPEC CPU thanks to binary optimizations. It successfully runs on programs from hundreds of Debian packages. We open-sourced Egalito [59, 60] to aid researchers in creating robust and efficient binary transformations.

Acknowledgments

We thank the reviewers for their valuable comments. This work was supported in part by the Office of Naval Research (ONR) under awards N00014-16-1-2263 and N00014-17-1-2788. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US government or ONR.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity. In *Proc. of ACM CCS*. 340–353.

[2] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2019. Nibbler: Debloating Binary Shared Libraries. In *Proc. of ACSAC*. 70–83.

[3] 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 *Proc. of ACM EuroSys*. 295–308.

[4] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries.. In *Proc. of USENIX SEC*. 583–600.

[5] Michael Backes and Stefan Nürnberger. 2014. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *Proc. of USENIX SEC*. 433–447.

[6] Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *Proc. of NDSS*. 40–47.

[7] Eli Bendersky. 2011. Position Independent Code (PIC) in shared libraries on x64. https://eli.thegreenplace.net/2011/11/11/position-independent-code-pic-in-shared-libraries-on-x64.

[8] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely Rerandomization for Mitigating Memory Disclosures. In *Proc. of ACM CCS*. 268–279.

[9] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. 2014. Hacking Blind. In *Proc. of IEEE S&P*. 227–242.

[10] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An Infrastructure for Adaptive Dynamic Optimization. In *Proc. of CGO*. 265–275.

[11] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Proc. of CAV*. 463–469.

[12] Bryan Buck and Jeffrey K. Hollingsworth. 2000. An API for Runtime Code Patching. *IJHPCA* 14, 4 (2000), 317–329.

[13] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining Light on Shadow Stacks. In *Proc. of IEEE S&P*. 985–999.

[14] Yurong Chen, Tian Lan, and Guru Venkataramani. 2017. DamGate: Dynamic Adaptive Multi-feature Gating in Program Binaries. In *Proc. of ACM FEAST*. 23–29.

[15] GNU Compiler Collection. 2017. Using the GNU Compiler Collection (GCC): AArch64 Options. https://gcc.gnu.org/onlinedocs/gcc/AArch64-Options.html.

[16] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2015. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proc. of ACM CCS*. 555–566.

[17] Al Danial. 2017. AlDanial/cloc. https://github.com/AlDanial/cloc.

[18] Debian. 2015. Hardening - Debian Wiki. https://wiki.debian.org/Hardening.

[19] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. REV.NG: A Unified Binary Analysis Framework to Recover CFGs and Function Boundaries. In *Proc. of CC*. 131–141.

[20] Chris Eagle. 2011. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press.

[21] Fedora. 2016. Harden All Packages - Fedora Project. https://fedoraproject.org/wiki/Changes/Harden_All_Packages.

[22] Erich Gamma. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, India.

[23] Google. 2018. fuchsia Git repositories. https://fuchsia.googlesource.com/.

[24] Cosmin Gorgovan. 2016. Escaping DynamoRIO and Pin - or why it's a worse-than-you-think idea to run untrusted code or to input untrusted data. https://github.com/lgeek/dynamorio_pin_escape.

[25] Cosmin Gorgovan, Amanieu D'antras, and Mikel Luján. 2016. MAMBO: A Low-Overhead Dynamic Binary Modification Tool for ARM. *ACM TACO* 13, 1 (2016), 14.

[26] Thomas Huet. 2017. AFL. https://github.com/mirrorer/afl/blob/master/docs/technical_details.txt.

[27] Intel. 2016. Intel is innovating to stop cyber attacks. https://blogs.intel.com/blog/intel-innovating-stop-cyber-attacks/.

[28] Intel. 2017. Control-flow Enforcement Technology Preview. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[29] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. 2012. kGuard: Lightweight Kernel Protection against Return-to-user Attacks. In *Proc. of USENIX SEC*. 459–474.

[30] Taegyu Kim, Chung Hwan Kim, Hongjun Choi, Yonghwi Kwon, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2017. RevARM: A Platform-Agnostic ARM Binary Rewriter for Security Applications. In *Proc. of ACSAC*. 412–424.

[31] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. 2002. Secure Execution via Program Shepherding. In *Proc. of USENIX SEC*. 191–206.

[32] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwartz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Proc. of IEEE S&P*. 1–19.

[33] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P. Kemerlis, and Michalis Polychronakis. 2018. Compiler-assisted Code Randomization. In *Proc. of IEEE S&P*. 461–477.

[34] Michael Larabel. 2018. Benchmarking Retpoline-Enabled GCC 8 With -mindirect-branch=thunk. https://www.phoronix.com/scan.php?page=article&item=gcc8-mindirect-thunk&num=2.

[35] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of CGO*. 75–86.

[36] Michael A. Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. 2010. PEBIL: Efficient Static Binary Instrumentation for Linux. In *Proc. of ISPASS*. 175–183.

[37] LLVM. 2019. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html.

[38] 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 *Proc. of ACM SIGPLAN PLDI*. 190–200.

[39] Marcus Meissner. 2017. openSUSE Tumbleweed now full of PIE. https://lists.opensuse.org/opensuse-factory/2017-06/msg00403.html.

[40] Microsoft. 2016. -DYNAMICBASE (Use address space layout randomization). https://docs.microsoft.com/en-us/cpp/build/reference/dynamicbase-use-address-space\protect\discretionary{\char\hyphenchar\font}{}{}layout-randomization.

[41] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Exploring Multiple Execution Paths for Malware Analysis. In *Proc. of IEEE S&P*. 231–245.

[42] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM SIGPLAN Notices*, Vol. 42. 89–100.

[43] Aleph One. 1996. Smashing The Stack For Fun And Profit. *Phrack* 7, 49 (Nov 1996).

[44] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2012. Smashing the Gadgets: Hindering Return-Oriented Programming using In-Place Code Randomization. In *Proc. of IEEE S&P*. 601–615.

[45] Avery Pennarun, Bill Allombert, and Petter Reinholdtsen. 2019. Debian Popularity Contest. https://popcon.debian.org/.

[46] Ashwin Ramaswamy, Sergey Bratus, Sean W. Smith, and Michael E. Locasto. 2010. Katana: A Hot Patching Framework for ELF Executables. In *Proc. of ARES*. 507–512.

[47] Martin Richtarsky. 2017. Hardening C/C++ Programs Part II - Executable-Space Protection and ASLR. https://www.productive-cpp.com/hardening-cpp-programs-executable-space-protection-address-space-layout-randomization-aslr/.
[48] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proc. of ACM CCS*. 552–61.
[49] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proc. of IEEE S&P*. 138–157.
[50] Maksim Shudrak. 2019. drAFL. https://github.com/mxmssh/drAFL.
[51] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proc. of ICISS*. 1–25.
[52] Paul Turner. 2018. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886.
[53] Ubuntu. 2016. Security/features - Ubuntu Wiki. https://wiki.ubuntu.com/Security/Features#Userspace_Hardening.
[54] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again. In *Proc. of NDSS*.
[55] Shuai Wang, Pei Wang, and Dinghao Wu. 2016. UROBOROS: Instrumenting Stripped Binaries with Static Reassembling. In *Proc. of IEEE SANER*. 236–247.
[56] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proc. of ACM CCS*. 157–168.
[57] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Securing Untrusted Code via Compiler-Agnostic Binary Rewriting. In *Proc. of ACSAC*. 299–308.
[58] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. 2011. Differentiating Code from Data in x86 Binaries. In *Proc. of ECML PKDD*. 522–536.
[59] David Williams-King et al. 2020. columbia/egalito. https://github.com/columbia/egalito.
[60] David Williams-King et al. 2020. Egalito. https://egalito.org.
[61] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *Proc. of USENIX OSDI*. 367–382.
[62] David Williams-King and Junfeng Yang. 2019. CodeMason: Binary-Level Profile-Guided Optimization. In *Proc. of ACM FEAST*. 47–53.
[63] Michal Zalewski. 2019. AFL. http://lcamtuf.coredump.cx/afl/.
[64] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R Sekar. 2014. A Platform for Secure Static Binary Instrumentation. *ACM SIGPLAN Notices* 49, 7 (2014), 129–140.
[65] Mingwei Zhang and R Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proc. of USENIX SEC*. 337–352.

A Artifact appendix

Submission and reviewing guidelines and methodology: *http://cTuning.org/ae/submission.html*

A.1 Abstract

We provide a virtual machine image which contains data needed to replicate some Egalito experiments. The machine contains the following:

- Egalito source repository (pre-built).
- Instructions to build Egalito from scratch.
- Scripts for several SPEC CPU 2006 experiments.
- Scripts to run Egalito and DynamoRIO AFL fuzzing.
- Large-scale jump table analysis for Debian packages.
- Large-scale Debian package tests.

Some experiments need internet access; in particular, the large-scale experiments rely on access to a Debian mirror. At http://doi.org/10.17605/OSF.IO/KDUZG we provide the virtual machine image and a `README.txt` file (which includes credentials).

A.2 Artifact check-list (meta-information)

- **Algorithm: binary recompilation**
- **Program: SPEC CPU 2006 v1.1 (must be obtained separately)**
- **Compilation: GCC 6.3.0**
- **Transformations: binary-to-binary recompiler**
- **Run-time environment: Debian stretch 9.11 + internet connection**
- **Hardware: 20GB disk, 8GB RAM, 4 core virtual machine**
- **Execution: 24+ hour runtime for complete experiments**
- **Output: performance numbers and transformation pass/fail**
- **Experiments: via scripts in virtual machine**
- **How much disk space required? 20GB (fixed)**
- **Publicly available?: Yes**
- **Code licenses?: GNU GPL v3**
- **Archived? Yes**
- **Artifacts publicly available?: Yes**
- **Artifacts functional?: Yes**
- **Artifacts reusable?: Yes**
- **Results validated?: No (as per conference policy)**

A.3 Description

A.3.1 How delivered. Please go to http://doi.org/10.17605/OSF.IO/KDUZG and download `egalito-artefact.tar.gz`. The archive is a 1.5GB download and requires 20GB of space once extracted. It contains the QEMU/KVM-compatible virtual machine image, a KVM machine definition XML file, and a copy of the `README.txt` with username/password and basic instructions. Further instructions on each experiment are included in additional `README` files in the home directory of the VM.

A.3.2 Hardware dependencies. The VM requires: 20GB disk space, 8GB RAM, 4 CPU cores. It should run on any x86_64 system (tested on Debian Linux only).

A.3.3 Software dependencies. We recommend using KVM with hardware acceleration enabled to run the virtual machine with optimal performance.

A.3.4 Data sets. SPEC CPU 2006 v1.1 (`SPECcpu2006-1.1.iso`) is required to replicate the SPEC CPU experiments.

A.4 Installation

Extract the `.tar.gz` (needs 20GB disk) and obtain the following files: `egalito-artefact.qcow2`, `machine.xml`, `README.txt`. Then,

create a new virtual machine with the qcow2 disk image, or import the existing machine. To import the existing machine, update the disk image path in `machine.xml` and import it into KVM with: `virsh define machine.xml`. You may get errors about unsupported CPU features depending on your CPU; the machine was created for an Intel i7-4770 host (M1).

After you've booted the machine, log in with the credentials in the `README.txt`. You can get a `TTY` console with `virsh console egalito-artefact`. You can also find the IP address of the virtual machine by running `ip addr show` on the guest, and then `ssh` in. If you created a new machine, you may need to run `sudo dhclient DEVICE` on the guest to get network access, where `DEVICE` is the name Linux chooses for your new network card.

If you wish to replicate the SPEC CPU results, find the IP address of the virtual machine, then `scp` your `SPECcpu2006-1.1.iso` to the VM's home directory from your host machine. Continue to follow the instructions in `README-speccpu.txt`.

A.5 Experiment workflow

Described in individual `README` files in the VM's home directory: `README-manual.txt`, `README-speccpu.txt`, `README-afl.txt`, and `README-largescale.txt`.

We recommend running experiments from within `tmux` because they can take a while. Also, you may wish to delete past experiments before running new ones to avoid running out of disk space.

A.6 Evaluation and expected result

Our SPEC CPU experiments should be able to successfully run `all_c` and `all_cpp` targets on `ref` size. We provide `mirrorgen` (1-1), `uniongen`, `retpolines`, `endbr` (Intel CET), `ss-const` (Intel CET + const shadow stack), and baseline configurations. Performance numbers collected in a virtual machine cannot be relied upon, but we observed similar results to our baremetal experiments: 4.4% slowdown for retpolines and 3.3% speedup for uniongen in the VM.

The AFL experiment should run approximately 25x faster for Egalito-AFL than for DynamoRIO-based fuzzing. The target is `/usr/bin/readelf`; we are not aware of any bugs in this program.

The large-scale experiments rely on accessing packages, sources, and build dependencies from a Debian mirror. These will necessarily change over time, so different numbers are to be expected. We successfully analyzed 1207 executables in the jump table analysis with 17 skipped and 0 failures; full details are included in the `README`. In the large-scale package tests, we saw 90 pass, 59 fail, and 140 skipped (due to compilation errors and/or lack of tests).

A.7 Experiment customization

The user can manually invoke Egalito on any executable, with various transformations. Any binary can be fuzzed with our AFL tool. Individual SPEC CPU targets can be run. We provide a SPEC diffing script to compare multiple runs. The large-scale tests are reusable, using a `chroot` environment to build and test packages. We believe the scripts may be useful in other contexts as well.