BinRec: Dynamic Binary Lifting and Recompilation

Anil Altinay*
University of California, Irvine
aaltinay@uci.edu

Joseph Nash*
University of California, Irvine
jmnash@uci.edu

Taddeus Kroes*
Vrije Universiteit Amsterdam
t.kroes@vu.nl

Prabhu Rajasekaran
University of California, Irvine
rajasekp@uci.edu

Dixin Zhou
University of California, Irvine
dixinz@uci.edu

Adrian Dabrowski
University of California, Irvine
a.dabrowski@uci.edu

David Gens
University of California, Irvine
dgens@uci.edu

Yeoul Na
University of California, Irvine
yeouln@uci.edu

Stijn Volckaert
imec-DistriNet, KU Leuven
stijn.volckaert@cs.kuleuven.be

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

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

Abstract

Binary lifting and recompilation allow a wide range of install-time program transformations, such as security hardening, deobfuscation, and reoptimization. Existing binary lifting tools are based on static disassembly and thus have to rely on heuristics to disassemble binaries.

In this paper, we present BinRec, a new approach to heuristic-free binary recompilation which lifts dynamic traces of a binary to a compiler-level intermediate representation (IR) and lowers the IR back to a "recovered" binary. This enables BinRec to apply rich program transformations, such as compiler-based optimization passes, on top of the recovered representation. We identify and address a number of challenges in binary lifting, including unique challenges posed by our dynamic approach. In contrast to existing frameworks, our dynamic frontend can accurately disassemble and lift binaries without heuristics, and we can successfully recover obfuscated code and all SPEC INT 2006 benchmarks including C++ applications. We evaluate BinRec in three application domains: i) binary reoptimization, ii) deobfuscation (by recovering partial program semantics from virtualization-obfuscated code), and iii) binary hardening (by applying existing compiler-level passes such as AddressSanitizer and SafeStack on binary code).

ACM Reference Format:
Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, Herbert Bos, and Michael Franz. 2020. BinRec: Dynamic Binary Lifting and Recompilation. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3342195.3387550

1 Introduction

Binary rewriting [27, 61, 62] has many applications such as post-installation program hardening [14, 40, 45, 58, 59, 68], (de)obfuscation [20, 64, 65], and reoptimization [22]. However, its effectiveness is limited in practice by the complexity of analysis and transformation in the absence of source code.

To overcome the limited expressiveness of assembly code, researchers introduced "binary lifting" which raises machine instructions to higher-level intermediate representations (IR) such as LLVM bitcode [4, 25, 26]. Binary lifting has the potential to capitalize on powerful compiler-level analysis and transformations already available in production compilers such as binary reoptimization. Despite its benefits, binary lifting has not seen widespread adoption in practice because existing approaches rely on static disassembly, which is fundamentally unable to accurately model indirect control-flow targets, differentiate between code pointers and data constants, or identify the boundary between data and instruction bytes [6, 33].

While heuristics have been used to successfully circumvent these limitations for certain binaries that adhere to specific assumptions [4, 62], binaries that are the target of analysis are typically release builds, stripped of symbols

*Equal Contribution Joint-First Authors

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387550

and debug information, and sometimes even intentionally obfuscated by vendors or malware authors. Code patterns found in such binaries easily violate these assumptions, e.g., handwritten assembly, highly optimized code, code produced by non-standard compilers, obfuscated or packed code, and even position-independent code, which is commonly used in shared libraries [7].

In contrast to static translation methods, dynamic binary translation (DBT) tools such as Pin [39], DynamoRIO [2] and Valgrind [43] analyze concrete executions of a target program, and thus can seamlessly handle all statically unknown components such as mixed code and data, and indirect control-flow targets. Unfortunately, the usability of existing DBT tools is limited for two reasons: first, they operate on the level of machine code, limiting the availability of complex analysis tools. Second, the rewritten code in their output is tailored to the tool's runtime environment, and can not be reused for subsequent executions. In other words, any transformation on the binary has to be done again each time the program runs. This introduces performance and portability problems for instrumented applications.

We present BinRec—a framework that employs dynamic analysis to lift binary code to LLVM IR in order to apply complex transformations, and subsequently *lowers* it back to machine code, producing a standalone executable binary which we call the *recovered* binary. To the best of our knowledge, BinRec is the first binary lifting framework based on dynamic disassembly, enabling lifting of statically unknown code for the first time. Additionally, BinRec is the first dynamic binary rewriting tool that persists its transformations in a standalone output binary.

Our main goal is to recover code that is opaque to static analysis. While our use of dynamic analysis solves this issue, it brings with it the problem of covering code that is not exercised during lifting: when dynamically lifting a program from a single trace, one only observes one out of many possible code paths. Hence, the recovered binary only supports code paths for which all control flow edges are present in the code path observed during lifting. A *control flow miss* occurs when the recovered binary reacahes a code path that was not covered during lifting. Much like page faults are handled by a page fault handler in modern operating systems [21], BinRec handles control flow misses by means of customizable handlers that may disallow the unknown control flow transfer by stopping execution. Alternatively, the handler may be configured to apply *incremental lifting*, allowing unknown edges and retrofitting the binary with the newly found code path. For optimization scenarios, the handler may even be left empty to allow for aggressive branch pruning, specializing the binary for a specific input format. Applications of our framework may select a handler that best suits their needs, for instance depending on whether unknown control flow is assumed to be malicious or not. The use of dynamic tracing enables us to produce recovered binaries with precise control-flow integrity (CFI). The allowable targets for any indirect control-flow are hence limited to the ones observed during (optionally incremental) lifting. We show that BinRec produces recovered binaries hardened with control flow integrity (CFI) with slowdowns of 0.98x – 1.29x, depending on the optimization level of the binary.

Crucially, BinRec allows us to harness the power of existing IR-level compiler analyses and transformations on binaries where static lifting fails. Our evaluation on SPEC CPU2006 shows that BinRec successfully lifts code patterns in optimized input binaries that state-of-the-art static lifters such as McSema [26] and Rev.ng [25] cannot. To demonstrate the immediate benefits of lifting binary code to compiler IR, we show that BinRec improves performance of some of our non-optimized input binaries and successfully applies two security transformations available in LLVM—SafeStack [36] and AddressSanitizer [51]—to our lifted IR. In contrast to previous binary rewriting approaches, BinRec naturally enables these compiler transformations without any additional engineering effort. We also show that trace-based lifting enables us to recover partial program semantics of virtualization-obfuscated binaries, by combining IR-level analysis with readily available compiler optimizations.

In summary, our contributions are the following:

- We present BinRec, the first dynamic binary lifting framework. BinRec uses dynamic program analysis, trace merging, and incremental recovery to lift programs to a compiler-level intermediate representation. Our prototype successfully handles stripped, real-world release binaries. It is available at https://github.com/securesystemslab/BinRec.
- We show that BinRec robustly recovers all SPEC INT 2006 benchmarks without heuristics, the first lifting framework to do so. We also show that these recovered binaries outperform those that are successfully lifted by state-of-the-art lifting tools.
- We evaluate BinRec in three application domains: i) Binary reoptimization, leveraging alias analysis tailored to the lifted IR resulting in improved performance in non-optimized binaries. ii) Binary hardening through CFI and compiler-level transformations such as AddressSanitizer and SafeStack. iii) Binary deobfuscation through successful recovery of partial program semantics in virtualization-obfuscated binaries.

2 Current Limitations in Binary Lifting

Analyzing binary code – or translating it to an accurate high-level representation that is better for analysis, transformation, and recompilation – is a challenging problem. The problem is compounded in cases where the binary code is encrypted or obfuscated. While many general problems of (static) disassembly have been well documented in the literature [6, 33], in this section we reiterate in detail some

of the current unsolved challenges in the context of binary lifting and program transformation through static methods. We describe these challenges below and motivate our new dynamic approach by explaining why static, heuristic-driven approaches are inherently insufficient for lifting arbitrary binaries.

2.1 (C1) Code vs Data, and Reference Ambiguity

By default, stock compilers do not attach labels to the data and references they embed into a program. To distinguish code from data and references from constants, the appropriate labels must be inferred through program analysis. This problem is undecidable in the general case [33], so state-of-the-art analysis tools employ heuristics to approximate the correct label set [61, 62, 68]. A data value, for example, can be considered a code reference if it is aligned correctly, and if it represents a valid code address in the binary. However, value collisions occur frequently [61] and alignment is not mandatory on many platforms. A dynamic tool can accurately assign labels by observing how the CPU interprets the values it reads from memory.

2.2 (C2) Indirect Control Flow

Indirect Control Flow transfers (iCFTs) may transfer control to one or more target locations depending on their execution context. Indirect calls are used to implement calls to function pointers in C code, which are even more prevalent in C++ code in the form of virtual functions. Indirect branches often implement switch statements and position-independent code (PIC). In PIC, all direct branches are replaced with indirect branches that add the offset at which the binary/library is mapped in memory, to the branch target.

Statically identifying all potential targets of iCFTs is, again, undecidable in the general case [33]. Static approaches do achieve high accuracy when identifying the potential targets of iCFT instructions that load their destination address from jump tables [24, 68]. Resolving indirect function calls and returns, on the other hand, remains a challenge. Wang et al. [62] argue handling iCFTs can be supported through their approach, but their prototype Uroboros does not handle iCFTs. The underlying analyses [24] used in Rev.Ng [25] claim 90-95% jump target recovery depending on architecture.

Meanwhile, Qian et al. [47] as well as Zhang and Sekar [68] use a lookup-table that translates original target addresses to the new addresses at run time, effectively resulting in a hybrid approach between static and dynamic rewriting. The table contains potential targets collected based on heuristics.

Dynamic tracing can reliably identify control flow targets as it follows the CPU to any jump target regardless of how the target address is computed.

2.3 (C3) External Entry Points

Dynamic linking is prevalent in real world software, and it presents additional hurdles to binary analysis and rewriting. Analyzing and rewriting external libraries at a binary level is generally infeasible; this requires static linking for all the library code [25] and incurs significant overhead [7]. Without visibility of all the code, however, the control and data flow between program modules is only partially observable to binary analysis through the interface of external modules.

Such partial visibility can be a problem when a code pointer of the main module is passed as an argument to an external module and is used to re-enter the main module, e.g., callbacks. After binary rewriting, this code pointer will become invalid because the code layout changes. Some existing binary rewriters attempt to support such callbacks by implementing special case handlers for the interface of known libraries [4, 63]. However, they cannot correctly handle external callbacks through unknown interfaces. Multiverse instead implements run-time lookup tables to handle callbacks [7] as a generic but heavyweight solution to support unknown external entry points.

Dynamic tracing can easily capture such entry points by recording control flow transfers going in and out of the targeted code space, which enables performant, surgical control and data modification at these points.

2.4 (C4) Ill-formed code

Manually written assembly code is not only used for optimization, but as an anti-debugging and anti-disassembly technique as well. While generated code is somewhat predictable, aggressive compiler optimizations can lead to similar ill-formed instruction constructs [6].

Overlapping instructions are a classic anti-disassembly technique [38] but occasionally appear in highly-optimized libraries too [6]. Selection control structures (e.g. switch/-case) are lowered as *Inline data and jump tables* by some compilers. *Overlapping basic blocks, multi-entry functions, and tail calls* obscure the detection of function boundaries.

Dynamic tracing bypasses handling of ill-formed code during disassembly by observing the actual instructions executed by the CPU instead.

2.5 (C5) Obfuscation

In addition to naturally occurring technical challenges, binary lifting approaches may have to deal with binaries that have explicitly been modified with the intent to obstruct analysis. While these obfuscation techniques are well documented [1, 3, 18], they still pose significant challenges in practice.

For instance, virtualizing obfuscators transform executable code stored in code sections into bytecode stored in data sections, and embed a virtual machine into the program to interpret the bytecode [1, 5]. In a program

protected by such an obfuscator, the static code sections reveal little to no information about the behavior of the program. Other problematic obfuscation techniques include opaque predicates [19], control-flow flattening [18], and aliasing [60]. All these transformations can be used to artificially inflate the size and complexity of the program's control-flow graph to a point where static disassembly becomes intractable.

Dynamic lifting can revert all of these obfuscating transformations to some extent. In the case of virtualizing obfuscation, a dynamic lifting tool can capture the run-time semantics of the program in the form of executable code, which can then be transformed into an equivalent deobfuscated trace [20, 49, 52, 65]. In the other cases, a dynamic tool can remove dead code and spurious aliases.

3 Design

Our design for BinRec overcomes the fundamental limitations identified in Section 2. We achieve this by leveraging *dynamic program analysis* to recover accurate disassembly of binaries which is then translated into a transformable, high-level intermediate representation (IR).

Figure 1 shows a high-level overview of our approach, consisting of three logical components: an extensible dynamic lifting engine and data collector, a transformation component that rewrites the IR code in a canonical way, and a back-end that compiles the transformed IR back to machine code and produces an executable binary. The lifting engine is extensible to support different execution driving paradigms. After running the canonicalization component, the full range of existing LLVM-based transformations can be applied to the client program.

3.1 Key Considerations for Dynamic Lifting

While our dynamic approach naturally sidesteps the limitations of static disassembly, it comes with its own set of challenges that need to be carefully addressed.

Coverage A fundamental challenge for any dynamic analysis is to drive execution through all desirable code paths [41]. Which code paths are desirable, however, depends on the type and goal of the analysis. To optimize binaries, for example, it is sufficient to explore the most frequently executed paths. For security hardening, it might be acceptable to explore only those paths reachable through trusted inputs and to prune all unexplored paths. For testing, the execution may need to cover all the code paths in the binary.

The paths BinRec covers depend on the set of inputs that drive execution, as is the case for any other dynamic analysis. We designed BinRec with configurable execution driving paradigms to accommodate a wide spectrum of applications (Section 3.2). BinRec can also merge multiple traces into a single transformable IR module, thereby recovering multiple sets of code paths (Section 3.3).

However, even with an ideal execution driver, the desirable control flow paths may not be fully exercised. This can lead the execution of the recovered binary to flow to code that was not covered during lifting, an event we refer to as a *control flow miss*. Lack of coverage can occur because the control flow of the program depends on implicit program inputs such as timing information, random numbers, and literal memory addresses. The coverage may also be incomplete because the concrete or symbolic inputs that achieve full coverage cannot be feasibly calculated. BinRec therefore handles control flow misses by means of customizable miss handlers, again, based on the application scenarios: The handler may be configured to disallow or ignore an unknown control flow transfer, or to incrementally recover the binary with the newly found code path (Section 3.5).

Scalability To dynamically disassemble or lift binary programs, they must be executed with concrete or symbolic inputs according to coverage considerations. Generating inputs to achieve maximum coverage is not only difficult, but may lead to path explosion for complex programs. To address this, we designed BinRec such that it can record multiple, independent, traces of the binary (resulting from multiple executions of the binary with different inputs). BinRec can merge the resulting traces at a later stage to increase global coverage. This design splits the analysis of complex, large binaries into smaller manageable chunks which can be lifted in a distributed and/or parallel infrastructure (Section 4.1).

3.2 Dynamic Lifting Engine

Execution Driver BinRec takes a multi-pronged approach, using several complementary methods, to drive dynamic execution. These methods use different types and sources of inputs. The first source of input to drive a program for dynamic lifting should be a test corpus exercising desired features. The more closely this corpus matches the real workload on the rewritten binary, the better. However, user-specified tests alone are unlikely to fully exercise all the code paths that should be lifted. Besides the obvious sources of explicit input to a program (command line, stdin), there can be implicit inputs that are much less obvious to users but still need to be accounted for. These can include address layout, timers, random number generators, interrupts and network packets. Even if users can specify the explicit inputs for every conceivable desired behavior of their specialized program, it is highly unlikely they would be aware of all the implicit inputs. We therefore turn to alternative techniques to produce specialized programs that are robust enough to function correctly in the presence of implicit input.

One potential solution to this problem is to drive execution through all or most of the program paths that depend on implicit input. Towards this solution, we drive some of the implicit inputs that cannot be triggered merely through explicit inputs. For example, we found an interesting case in the Perl

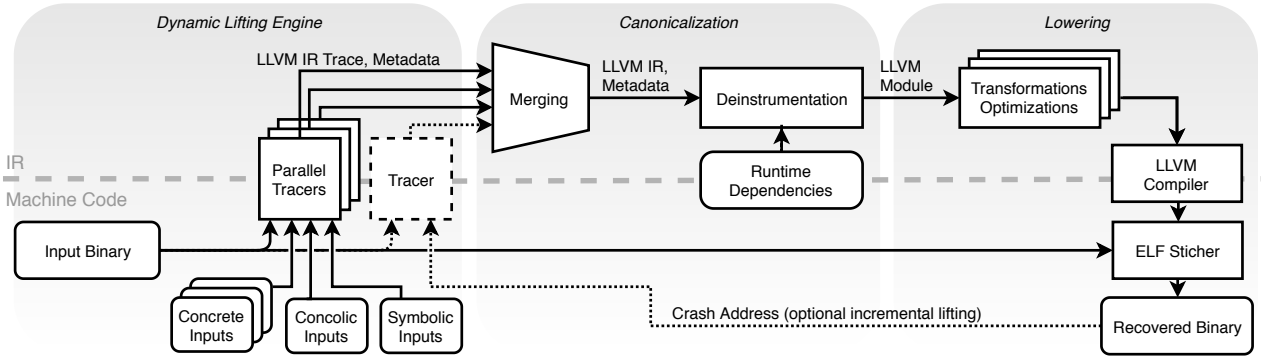


Figure 1. The steps of binary recovery: lifting to compiler IR, transformation on the IR, and lowering back to machine code.

interpreter where the control-flow depends on the virtual address space layout (specifically on the alignment of `argv` strings). We exercise this implicit input source by controlling the lengths of environment strings in a way that results in different `argv` alignments. Similarly, we enable address space randomization (ASLR) during tracing, to exercise more code paths dependent on the address space layout.

While achieving complete code coverage is not our objective, the users may still require nearly complete code coverage depending on the application. For this use case, BinRec supports concolic execution [16] to explore more code paths.

Alternatively, BinRec can take fuzzer-generated, concrete inputs to drive the binary lifting frontend. Concolic execution and fuzzing have complementary strengths and weaknesses [29, 67]. Fuzzing scales well to large programs, but has difficulty exploring all branches of complex conditional statements. Concolic execution is useful to drive execution through such conditionals. We found concolic execution to become untenable on programs with cryptography or hashing, such as SHA2. In those programs, the SMT solver becomes a bottleneck. The input generation interface is flexible and extensible, which allows the dynamic driver to be customized for a particular client application, and so explore program paths using the best methodology for the target.

Dynamic Data Recording We record dynamic data about the execution of each program path specified by the driver. This data is key to overcome the fundamental limitations of static binary lifting as explained in Section 2. We currently record which instructions were executed, where the function boundaries are, and the observed targets of each branch instruction. We use this information to accurately disassemble binaries and produce canonical IR, as explained in Section 3.3.1. Our framework is extensible, so other data can be recorded to fill the needs of downstream transformations. The recorded data is fully accurate on paths which are exercised by the dynamic lifting engine, but we cannot reason about data that is not covered by the dynamic traces.

The BinRec front-end decodes and records each instruction executed by the client binary using the program counter. This procedure is agnostic to the static representation of the executable code and is therefore not affected by any intentional or unintentional differences between the static and dynamic (actual) instruction trace. Such differences would arise in the presence of unaligned, packed, or encrypted code. We therefore address (C4) and some aspects of obfuscation (C5). A tradeoff incurred by this design choice is a potentially slower lifting front-end. A scheme that dynamically records control flow, but that lifts disassembled basic blocks statically would occupy another point in the design space, and would sacrifice compatibility with non-standard binaries for faster lifting.

3.3 Canonicalization

Merging Traces BinRec can compose program traces generated over different runs using different execution driving paradigms. We implemented a technique to merge distinct traces into one specialized program which will behave correctly on all covered paths. In concrete terms, merging proceeds by lifting N instances of the target program in parallel. The different execution paths can be driven by fuzzing, concolic execution, or a chosen corpus of inputs. Then, we create one LLVM IR module from N LLVM IR modules using metadata we collected during lifting.

Merging depends on the ability to correlate the code and data addresses of one dynamic trace with another. In the case of position independent code, the addresses change from trace to trace, but are correlated by the section base addresses. Traces from programs using fine grained code and/or data layout randomization (at load or run-time) could be merged using a specific mapping function taking the randomization seed as input. We leave the implementation of such correlation techniques as future work.

The code of the combined program is the union of all basic blocks observed in the merged traces. The allowed targets of each control flow statement in the combined program are

the union of the observed targets for each observation of that branch in the merged traces.

It may be observed that this is a path-insensitive procedure. The resulting control-flow graph, before optimization, resembles the original program's CFG but lacks the nodes that were not executed while lifting. One could imagine an alternative, path-sensitive, reassembly technique, where only control flow paths exactly following one of the recorded traces are allowed. However, it is likely unprofitable to construct such a recovered program, as in effect this would be a tree traversal of the original program's control flow graph, and the resulting program would have a code size explosion.

Deinstrumentation BinRec uses an emulation-based dynamic lifting engine, which allows us to lift programs compiled for a different instruction set architecture than the host system. IR generated from such an emulation-based engine, however, is heavily instrumented to facilitate execution in a virtualized environment. This code cannot be used as a standalone program, unless we remove the instrumentation code. Our framework contains a *deinstrumentation* component that eliminates dependencies on the run-time environment from lifted code, and merges all captured code together into a single LLVM module that is suitable for use in subsequent transformation passes and compilation into a standalone binary.

Whereas a program binary can explicitly use physical CPU registers and memory references, the lifted IR of a recovered program has an abstract representation of the memory model in the original binary. To handle this abstraction gap, we represent physical registers, stack and memory locations as objects in the high-level IR. This enables us to generate programs which contain two stacks and register sets. The native stack contains data such as register spills and return addresses, as well as any data we add while transforming the lifted program. The emulated stack and register set contain the data of the original binary. Generated code interacts with this emulated environment to reproduce the functionality of the original program. The emulated state cannot be fully optimized into native state due to the lack of semantic information about the size and lifetime of stack allocations.

3.3.1 Control-Flow Canonicalization

Indirect Control Flow Resolution Our lifting front-end produces a collection of executed basic blocks, and a list of control-flow graph edges. We use this data to emit control-flow transfers with sound and precise lists of allowed targets. Direct control flow transfers have a one-to-one correspondence between nodes and edges in the observed control-flow graph of the client binary, and the recovered binary. We therefore represent them in a straightforward way in recovered code, using the original semantics.

Even the most precise static analysis allows more control flow targets than necessary due to analysis imprecision

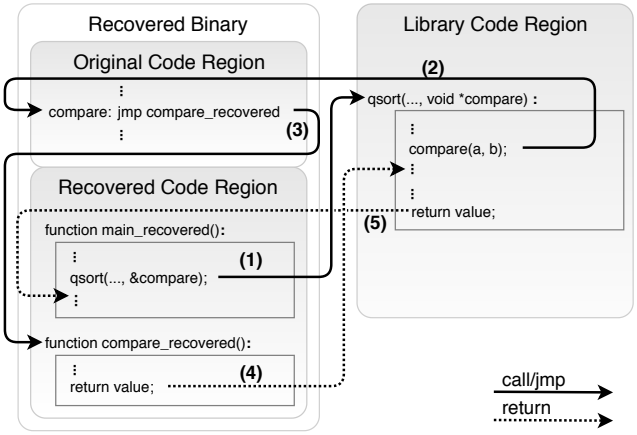


Figure 2. The address space of a recovered program that calls the `qsort` library function. Control flows as follows: (1) Call to library with original function pointer; (2) Callback via function pointer; (3) Original function was replaced with jump to recovered code; (4)(5) Returns.

(see challenge C2). In contrast, we simply record the exact dynamic targets of each indirect control flow transfer in a client binary in the lifting engine. To execute the corresponding control flow in the recovered binary, we determine the address that original code would jump to, then use that address as a key to look up the recovered code target. This is represented as a switch table in LLVM IR. We emit the minimal set of dynamic targets, which can enable further optimization by limiting the lifetime of values. Static lifting can only receive these benefits to the extent that indirect branch targets can be statically determined. This has been extensively explored in the program analysis [6] and CFI literature [11, 13], and previous work has found even the most precise static analysis overapproximates the set of possible targets.

Library Calls BinRec supports calls to external (i.e., non-recovered) libraries. The principal step necessary to execute such a library call is to marshall the emulated program state into concrete state before the call. Marshalling is necessary to match the ABI of linked libraries. Upon return from the library, the concrete state is reloaded into the emulated state. The maximum amount of state that may need to be transferred is the full register set, including the stack pointer. When possible, we can use the function signatures of external library calls to optimize the state marshalling. With signature information, only caller saved registers which are actually read or written need to be marshalled from emulated state to concrete state. Our prototype implementation of BinRec uses signature information to optimize calls to the C library.

External Callbacks Our approach to solving the external callback challenge C3 is both sound and performant. Only a dynamic lifting approach can achieve both these properties

at once. In the lifting front-end, BinRec detects execution of the binary under analysis, and records call targets where the caller is outside the analysis region (i.e., callback functions). There is no need to track callback pointers at any other time because we detect when they are actually invoked. We also record the instruction pointer values when the called-back code exits to external library code via a `call` or `ret` instruction. We insert entry stubs for the external code to recovered code transitions, and exit stubs at recovered code to external code transitions. These stubs also perform the state marshalling mentioned in the previous paragraph. During the ELF stitching phase (Section 3.4), we insert code trampolines at the original virtual addresses of the called-back functions. Figure 2 visualizes the resulting control flow for a call to `qsort` which includes a simple callback.

If a static binary lifter attempted to use trampolines to handle callbacks as we have, they would lack the dynamic information about which functions are actually executed via callback. Without the dynamic information, the only sound approach would be to mark every function as a potential entry point. Creating many entry points to recovered code is deleterious to performance, as it increases code size and forces variables to be stored and reloaded.

3.3.2 Data Canonicalization

Accurately lifting data structures from binaries is a hard problem and the focus of orthogonal research [56]. Some architectures allow interleaving of code and data. This is true for ARM, but also for x86 where compilers often embed jump tables into code sections. In BinRec, we take a conservative approach by including data from the client binary as global variables in the IR, as well as copying any code sections in the binary that may contain data. We preserve their base mapping addresses in order not to invalidate existing references in the lifted code. We leave the task of applying existing analysis methods to split up the data into variables and creating typed references in the lifted code to future research. Thanks to our lifting engine, such analysis methods can benefit from strong data flow analysis at the level of compiler IR.

3.4 Lowering

After the client program IR has been transformed as desired, we produce a functional recovered binary. We use an unmodified LLVM compiler (`llc`) to generate a temporary ELF binary from the recovered IR. Then, our lowering toolchain stitches together ELF sections from the temporary binary and the original binary into one combined binary. We use the majority of sections from the temporary binary, and data sections from the original. Finally, we execute binary patching to insert the trampolines to support external callbacks (Section 3.3.1), and update dynamic linking structures (Section 3.4).

Dynamic Linking We lift all dynamic data and code references into canonical LLVM IR, and then lower this IR using LLVM's code generation infrastructure. This functionality requires us to redirect references to external functions and data used by the client binary. In addition to static references, we collect the dynamic addresses of every indirect load, which enables us to redirect those references to external symbols as well. We then ensure the dynamic linker operates on only lifted data structures, which is necessary given our atypical ELF layout. We utilize the ELF dynamic symbols section to determine the address of data symbols which will be filled by the dynamic linker. Even stripped binaries must retain this information. This approach could be extended to non-ELF binaries with minimal effort by implementing the API of the platform-specific dynamic linker. The real world benefit of dynamic linking support is that BinRec can support any off-the-shelf instrumentation scheme that acts via inserted calls to an external library. We use this functionality to enable the AddressSanitizer and SafeStack applications in Section 6.

3.5 Control Flow Miss Handling

Binaries recovered with BinRec may encounter unrecovered paths during testing or after deployment due to the coverage limitation of dynamic analysis (see Section 3.1). BinRec handles these control flow misses by forcing the recovered binary to invoke a *control flow miss handler* whenever it encounters an unrecovered path. Several control flow miss handlers are available.

The **log** hander logs the instruction pointer value that is missing from the recovered binary, and then aborts execution. This mode is useful when divergence between the recovered binary and the original is more dangerous than program termination.

The **fallback** handler diverts execution from the recovered code into the original code of the input binary. This involves marshalling of the emulated CPU state in the recovered code into the physical state of the original binary (see also Section 3.3.1), and then jumping to the original binary at the intended address. This miss handler is only available when the original binary and recovered binary target the same architecture. It is ideal for use cases that require program instrumentation without unexpected termination. Note that in a mitigation scenario, in which BinRec is used to augment lifted code with security instrumentation, this requires a binary-level mitigation for the remaining binary code. The binary mitigation may be heavyweight and hence inefficient. However, the fallback code is not expected to be on the hot path since it is not exercised by the lifting workload.

The **incremental lifting** handler feeds back the logged missing instruction pointers into the dynamic lifting engine, where we capture a trace covering the new control-flow edge, and merge it with the existing traces. Using this *incremental lifting* paradigm, the recovered binary can be continuously

updated. Our current incremental lifting prototype lifts instructions until the next conditional control-flow transfer.

The recovered program can invoke the fallback miss handler, or the log handler. Meanwhile, the dynamic lifting engine can generate one or more new program traces via the logged instruction pointers in an asynchronous background process. We incorporate the new and existing traces to generate a new recovered binary.

An advantage of incremental lifting is it directly lifts new code without the need to reproduce the (explicit or implicit) input that triggers the miss during lifting. Consider a program feature that is only exercised due to unconstrained system randomness on the test system. There is no need to isolate and constrain the source of randomness to replicate it on the lifting system. Alternatively, there is no need to wait for non-deterministic fuzzing or concolic execution techniques to drive execution through the new paths.

Finally, when it is known that the tracing stage has already covered all paths that implement the features of interest, the miss handler can be optimized out completely. This is useful for aggressive optimization scenarios in which the lifting input is known to cover all necessary code, and eliminating a branch leads to new optimization opportunities.

4 Implementation

We implemented a prototype of BinRec, spanning 13,338 SLOC of which 9,709 are C++ code that implements lifting and canonicalization. The implementation targets single threaded 32-bit x86 binaries on Linux.

Our dynamic lifting engine is built on top of S^{2}E [16], a framework that facilitates symbolic execution of a single process running in the QEMU virtual machine [8]. Code is translated to LLVM IR in order to be symbolically executed by the KLEE symbolic executor [12]. S^{2}E automatically provides multi-architecture support and sandboxing of input binaries, since it is based on QEMU. This flexibility comes at the cost of a relatively long lifting time, which we discuss in Section 5.4.

4.1 Parallel Tracing

To address the scalability challenge (see Section 3.1), we architected BinRec with high parallelism. Dynamic tracing is expensive in time (due to dynamic binary translation) and disk usage (due to virtual machine images). We implemented a flexible run configuration scheme that allows operators to describe test cases to saturate a server's CPU and memory resources. Multiple traces through the same binary are lifted in parallel, and we can also lift different binaries in parallel.

The dynamic traces do not all have to be conducted at one time, so a lifted binary can be produced and used while more paths are being explored for the next version of the lifted binary. A dynamic trace is a stable artifact on disk that can be copied, shared, and reused. This allows the coverage of a binary to continuously be improved, and traces will not have to be regenerated.

4.2 Optimization

S^{2}E represents all instructions as modifications to a `struct` which stores the complete state of the original binary. This hinders existing LLVM passes from precisely analyzing and optimizing code. To address this issue, we optimize lifted code in several ways. First, our deinstrumentation described in Section 3.3 brings the code into a state where LLVM can perform existing optimizations including aggressive constant propagation and dead code elimination. Next, we guide the alias analysis with the fact that pointers to non-overlapping registers in the emulated register state cannot alias [23]. Third, we aggressively promote global variables representing the client binary state to equivalent local variables; even inlining functions that use them if it is favorable. Figure 3 shows the performance benefit obtained by applying our custom alias analysis and global variable promotion.

Stack unwinding optimization Client binaries often utilize error handling mechanisms such as setjmp and longjmp which save and restore the program state. BinRec programs have two contexts, the physical context of the recovered program, and the emulated context of the original program. Setjmp and longjmp calls in the original program should be translated to a save and restore of the emulated context in the recovered program. It would be possible to copy the emulated state to physical state, the same way we do for library calls, and thereby use the native setjmp/longjmp handlers. Instead, we implemented our own handlers to avoid the extra state copy by directly operating on emulated state.

5 Evaluation

In this section, we first compare our prototype against state-of-the-art static lifting approaches. We then assess the performance of programs lifted by BinRec in terms of run time and code coverage, as well as the lifting speed of our BinRec prototype. We use the SPEC CPU2006 benchmark suite, which is standard in the binary lifting literature [4, 7, 25], because it contains CPU-bound benchmarks, providing us with a pessimistic view of run-time overheads (as opposed to I/O-bound programs whose I/O performance is unaffected by lifting). We conducted our lifted binary run-time experiments on a system with 8GB RAM and an Intel i5-3210M running at 2.5GHz, with frequency scaling turned off to ensure stable performance. Lifting time experiments were conducted on an Intel Xeon E7-4870 @ 2.40GHz with 188 GB RAM. We used gcc 4.8.4 to compile all programs with optimization levels O0 and O3 (see Table 1). Our prototype is based on S^{2}E, which emulates floating-point instructions using integer instructions for portability. In this prototype implementation, we do not aim to optimize floating-point performance, so we limit our evaluation to the CINT subset of SPEC CPU2006.

```
1  void callback_func(j_common_ptr cinfo) {
2    printf(".");
3  }
4
5  int main (int argc, char **argv) {
6      struct jpeg_decompress_struct info; //jpeg info
7      struct jpeg_progress_mgr  progress;
8      ...
9      //After some initialization code
10     progress.progress_monitor = callback_func;
11     progress.pass_limit = 0x8048860;
12     progress.pass_counter = 0L;
13
14     info.progress = &progress;
15     jpeg_start_decompress( &info );
16
17     char *data = (char *) malloc(dataSize);
18     readData(info, data);
19     ...
20 }
```

Listing 1. Excerpt of *decompress.c*: libjpeg example in C.

5.1 Comparison with static lifters

BinRec reliably lifts and recompiles a large number of real-world binaries. In addition to the qualitative benefits of our dynamic technique as discussed in Section 3, we investigated quantitative advantages of our approach. We compared BinRec to McSema [26] and Rev.ng [25], popular state-of-the-art binary lifting frameworks.[1] We limit our comparative study to active, open source binary lifters which, like BinRec, aim to be compiler-agnostic.

We found McSema [26] could only recover a limited number of binaries correctly in our tests. While trying to lift binaries compiled without optimization, we encountered errors with McSema's handling of double-precision floating point operations in 32-bit applications, unsupported xmm instructions (xmm xorpd, xmm andpd) on 64-bit, and segmentation faults in the C++ delete operator. In addition, some binaries lifted from compiler-optimized code caused segmentation faults upon launch or produced incorrect output.

We also identified cases where binaries generated by McSema interpreted data as code pointers, illustrating (C1) in real-world code. McSema uses IDA Pro for control flow graph recovery and analysis. Hence, it is limited by IDA's inability to correctly identify function pointers in real-world code. This can lead to problems as illustrated by Listing 1: a structure type in libjpeg contains a member field that holds the address of a callback function (line 10), while another holds an integer that represents a loop bound (line 11) which happens to be in a similar value range. IDA is closed source, but we suspect it uses heuristics to identify integers with values in the executable segment as code pointers, which fails in this case. The recovered binary McSema generates from this program mistakenly changes the integer, thereby changing program semantics. Similarly, failure to identify code pointers correctly could cause mishandling of callbacks in this program. Unfortunately, the authors do not provide any performance numbers for correctly lifted binaries using McSema.

[1] Code snapshot on July 25th, 2019

Table 1. Measured execution time normalized to the original binaries. Rev.ng results are reported from publication [31].

	BinRec		McSema		Rev.ng
Benchmark	**O0**	**O3**	**O0**	**O3**	**reported**
400.perlbench	1.25	1.48	–	–	3.7
401.bzip2	0.76	1.05	2.84	–	2.2
403.gcc	1.26	1.37	–	–	2.1
429.mcf	0.83	1.00	2.31	1.41	1.5
445.gobmk	1.04	1.56	–	–	3.3
456.hmmer	0.77	0.74	–	–	2.2
458.sjeng	0.77	1.08	3.43	–	2.6
462.libquantum	0.95	1.30	2.07	1.04	1.1
464.h264ref	0.80	1.24	–	–	2.7
471.omnetpp	1.92	3.09	–	–	2.8
473.astar	0.80	0.94	–	–	1.5
483.xalancbmk	1.12	1.66	–	–	2.8
geomean	0.98	1.29	–	–	2.25

We were unable to recover most of the dynamically linked SPEC INT2006 binaries with Rev.ng [25]. While we managed to get some of the binaries running by reducing the optimization level to O0 (a classic example of (C4)—due to aggressive optimization), this still yielded mixed results. For instance, the tool was able to produce a lifted version of *libquantum* but its output differed from the output of the original program. The only test that was correctly recovered was *mcf*. Some tests failed completely (even at O0), e.g., *gcc*, *gobmk*, *perlbench*, and *xalancbmk*.[2] Table 1 compares the performance of BinRec to Rev.ng using the most recent published results [31]. The authors note that these were all statically linked. Although their client binaries' optimization level is not specified, BinRec's performance (0.98x for O0, 1.29x for O3) exceeds Rev.ng's (2.25x) in either case.

In summary, both state-of-the-art tools we looked at were unable to reproducibly recover even standard binaries, despite being actively developed and widely used open-source frameworks for binary lifting. We would like to stress that this does not reflect a lack of sophistication behind those tools (or the developers), but instead highlights the tremendous difficulty faced by static lifting approaches. Crucially, we found our dynamic tracing technique to aid the lifting process within BinRec significantly: we are able to recover all of the test binaries in question while the recovered binaries performed favorably by comparison and produced correct output.

5.2 Performance

Table 1 presents the performance of binaries lifted with BinRec. For every input program we compiled both optimized (O3) and unoptimized (O0) binaries which produce correct

[2] The error message indicated failed assertions in the IsolateFunctionsImpl class upon replacement of indirect branch targets, strongly hinting towards an instance of (C2). We contacted the developers but did not get any detailed feedback in time for the submission.

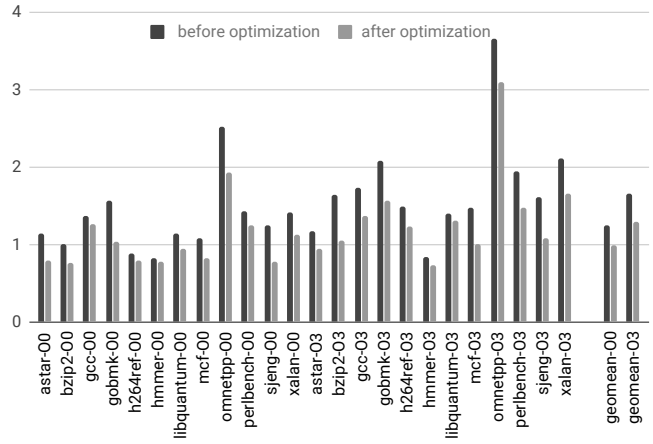


Figure 3. Execution time improvement from CPU state variable de-aliasing and global variable promotion.

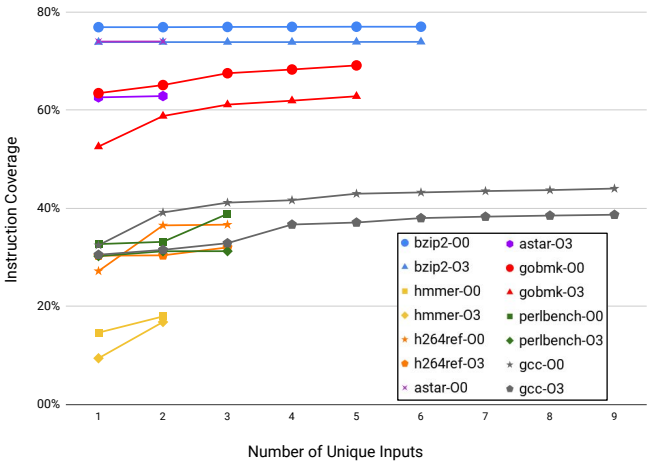


Figure 4. Coverage with respect to the original binaries. The input set is the *ref* workload of SPEC CPU2006.

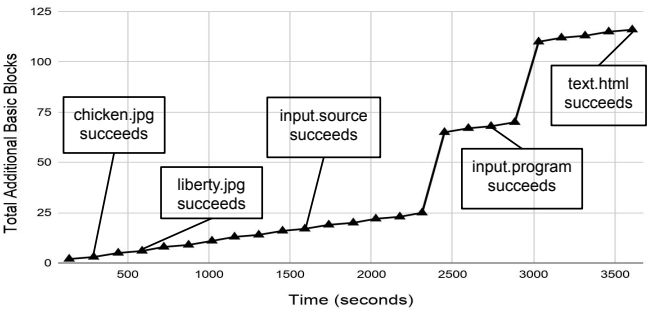


Figure 5. Incremental lifting progression of *bzip2*.

output in the test cases. Our results show that there is a potential for performance improvement by using BinRec as a post-release optimizer—particularly, if the original was not optimized at the source level. With BinRec, six benchmarks – *bzip2*, *mcf*, *hmmer*, *sjeng*, *h264ref*, and *astar* – run faster than the unoptimized client binaries. In some cases, BinRec can re-optimize release builds to be faster than even the optimized binaries (e.g., *hmmer* and *astar*). Compared to the *optimized* client binary, the *hmmer* "nph3.hmm swiss41" workload finished in 0.62x the time. Interestingly, *hmmer* is the only SPEC binary to be faster when re-optimized from an optimized (0.62x) rather than an unoptimized client binary (0.85x).

There are factors that accelerate and factors that slow down programs recovered by BinRec. We discussed several of the accelerating factors in Section 4.2 and show their benefit in Figure 3. Floating point instructions are emulated in the lifted binaries, which incurs a performance penalty (e.g., we found this to be one of the main factors for the slowdown of *omnetpp*). Further, the IR of recovered programs contains less accurate information about the size and lifetime of stack allocations compared to source code, which impedes optimization. The geometric mean run time factor of BinRec binaries compared to unoptimized and optimized input binaries is 0.98x and 1.29x, respectively.

5.3 Code Coverage

Figure 4 shows the instruction coverage of lifted binaries as we increase the number of supported input workloads. The rate of coverage change is substantially different between binaries, and reflects both the number of unrelated features in the binary and the similarity of the test cases. *bzip2*, for instance, exercises nearly the same code path for each input. In contrast, *gobmk* and *gcc* see a steady increase in code coverage for each added input. The level of instruction coverage should therefore be dependent on the application, lifted feature set, and use case. Users of our framework may aim to increase coverage or to keep it low, limiting the attack surface for attackers. In both cases, BinRec's ability to report code coverage provides the user with a practical metric to determine if incremental lifting is effective; either in maintaining low coverage or in increasing coverage.

Incremental Lifting To show the effectiveness of incremental lifting, we conducted an experiment with *bzip2* as illustrated in Figure 5. We first lifted the binary with SPEC `training` inputs, which is the origin point of the graph. Then, we ran the lifted binary with `reference` inputs and incrementally lifted code to support each new input. The callouts on Figure 5 indicate when each additional input became functional in the recovered binary. Each triangle represents one cycle through the lifting frontend, and each cycle took approximately 140 seconds.

5.4 Lifting Time

BinRec's ability to robustly lift binaries without relying on heuristics comes at the cost of lifting time. As a dynamic lifting tool, BinRec's lifting time depends on the execution time of its input programs. Table 2 shows BinRec's lifting times for each input binary. In order to show the worst case lifting time, we used a SPEC reference input—which fully excercises loop iterations—for lifting. The lifting could be

Table 2. Time in seconds to capture LLVM IR from input binaries with BinRec and McSema toolchains, alongside execution time for S^{2}E without BinRec instrumentation. For BinRec and S^{2}E we report the maximum time among the reference workloads from SPEC CPU2006.

	BinRec		S^{2}E		McSema	
Benchmark	**O0**	**O3**	**O0**	**O3**	**O0**	**O3**
400.perlbench	425,619	321,078	62,482	49,221	3,375	3,385
401.bzip2	86,181	69,389	27,614	18,311	117	122
403.gcc	37,276	28,468	6,156	4,929	6,996	7,378
429.mcf	283,413	227,999	209,914	197,910	11	8
445.gobmk	84,214	72,307	15,496	8,721	1,332	1,063
456.hmmer	179,127	144,529	87,911	28,159	204	189
458.sjeng	727,675	548,432	95,936	86,153	294	368
462.libquantum	421,269	176,536	–	49,334	21	16
464.h264ref	86,433	65,202	31,012	15,233	336	586
471.omnetpp	–	312,665	–	105,015	258	224
473.astar	211,782	119,436	80,613	66,201	22	18
483.xalancbmk	–	–	–	–	74,948	17,103
geomean	178,480	138,379	44,810	35,021	371	320

much faster with an optimized trace input which is designed to reach more paths and minimize loop counts, but such an optimization would not reflect real world workloads. Since the current prototype of BinRec uses S^{2}E [16] as its tracing frontend, we also present the time to execute those workloads without instrumentation in S^{2}E. The lifting time of static lifting toolchains, such as McSema, does not depend on the input, and is in general faster than our dynamic approach. We present the lifting time which we collected with McSema here for comparision.

Binary lifting is a one-time, offline process and thus it does not affect performance of actual binary execution. If fast lifting times were in fact desired, it could be accomplished using a faster tracing frontend such as Pin, KVM-enabled QEMU, or native execution with a hardware control-flow tracing feature (e.g., Intel PT). In that case, however, we may miss the flexibility of disassembly in S^{2}E, and its ability to explore multiple code paths through concolic execution.

6 Applications

One of BinRec's main goals is to enable complex transformations on real-world program binaries. Since BinRec can lift binaries to a compiler-level IR and supports dynamic linking, this enables us to make use of a large ecosystem of off-the-shelf compiler-based transformations and analysis tools. In addition to compiler transformations, existing, black box binary utilities such as `readelf` or `LD_PRELOAD` remain usable on BinRec binaries. In this section, we showcase some applications that demonstrate this ability: deobfuscation, AddressSanitizer and SafeStack through compiler transformations, and control-flow hijacking mitigation. Developers who are familiar with these transformations do not need any knowledge of binary analysis to use them within our framework. While we only provide a limited set of example applications in this section, we note that BinRec reliably enables—for the first time—a large number of interesting, feature-rich program analyses and transformations through extensive compiler-based tooling for binary programs.

6.1 Control-flow Hijacking Mitigation

Even without additional compiler-based transformations, BinRec has an endogenous ability to mitigate memory-corruption vulnerabilities in the original program. A recovered program emulates the execution of the original program. Because of the emulation, what was control flow in the original program becomes data flow in the recovered program. BinRec does not natively mitigate data-only attacks [34], though they may be mitigated using additional transformation on the IR.

A control-flow hijacking attack typically subverts control flow by overwriting a code pointer. This pointer could be used by an indirect jump, indirect call, or return instruction. When tracing indirect control flow in the original program within BinRec, we observe actual control flow targets. The recovered program then contains switch statements where cases are jumps to these observed targets. The value of the instruction pointer (`%rip`) in the original program is emulated by the recovered program, and it is used as the index into the switch statement. The switch statements are lowered into assembly consisting of trees of compare and direct jump instructions, so no new attack surface is introduced by this dispatch mechanism. This is functionally equivalent to what is commonly known as context-insensitive control-flow integrity on forward and backward edges [11].

Consider an original binary with a vulnerable stack buffer overflow using an unsanitized `strcpy` call, that can be used to overwrite a return address. In the recovered program, that buffer is located in a `@memory` array which emulates the memory of the original program. The `strcpy` call will proceed in the same way in the recovered program, allowing

Table 3. Number of allowed targets for indirect branches/-calls in SPEC CPU2006 binaries lifted by BinRec, compared to the number statically found by BinCFI [68].

	O0				O3			
	BinRec CFI			BinCFI	BinRec CFI			BinCFI
Benchmark	**Median**	**IQR**	**Max**		**Median**	**IQR**	**Max**	
400.perlbench	5	7.5	176	2,101	4	7	176	1,916
401.bzip2	3	0	22	151	3	0	22	117
403.gcc	4	3	212	6,593	3	3	212	5,407
429.mcf	3	1	7	68	3	0	7	66
445.gobmk	3	0	492	2,780	3	0	492	2,590
456.hmmer	3	0	8	671	3	0	7	620
458.sjeng	4.5	3	12	223	5.5	3.3	12	215
462.libquantum	3	0	2	177	3	0	5	161
464.h264ref	3	0	10	686	3	0	10	617
471.omnetpp	3	4	168	3,167	3	1.3	168	2,482
473.astar	3	0	3	213	3	0	4	139
483.xalancbmk	3	4	38	35,106	4	3	38	15,950

IQR: inter-quartile range

the attacker to overwrite the emulated return address of the vulnerable function. The original return instruction is emulated using a switch statement, loading an attacker-provided value via the emulated register `@RIP`. If the target is not one of the traced return sites of the vulnerable function, the error case of the switch statement will abort execution. Otherwise, execution will proceed in the style of a control-flow bending attack [13], since the target address represents a valid execution under context-sensitive (but not path-sensitive) analysis of the original program. If optimization is applied to the recovered binary and the error case is deleted from this switch statement, one of the observed return targets of the vulnerable function will be chosen in a compiler-specified manner. In this case, an attacker aware of BinRec could still perform control-flow bending. However, any attempt to hijack control flow via writing code pointers (*vtable* overwrite, indirect code pointer write via heap overflow, etc.) is mitigated.

To evaluate the security properties of the resulting solution, we measured the number of allowed targets across all the recovered edges. Though our approach also protects returns, we only present forward edges in Table 3 for easier comparison with other approaches. Our results show that BinRec can enforce a median number of around 3 indirect callees on a nontrivial fraction of the target programs. The table also shows these results for binCFI [68], a static binary-level CFI solution. Because it can not statically predict valid branch targets with precision, binCFI's policy must allow transfers to any address-taken function, increasing the number of allowed branch targets by orders of magnitude when compared to BinRec.

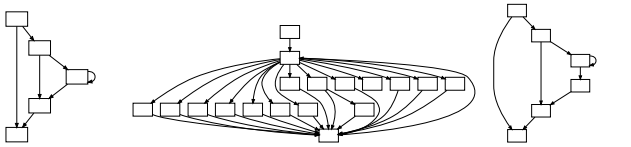
6.2 Virtualization-deobfuscation

We used BinRec to lift programs obfuscated by virtualization (cf., Section 2.5). Figure 6 illustrates our deobfuscation approach. For this use case, we detect the Virtual Program Counter (VPC) and virtual interpreter loop through known techniques [52]. We instrument the recovered IR to log the value of the VPC at the entry point of the interpreter loop, then produce a binary. We exercise the instrumented binary to obtain a graph of VPC nodes. We create a new program from this graph by copying the body of the virtual interperter loop into the VPC nodes. After applying standard compiler optimizations (most notably constant propagation and dead code elimination), only one virtual opcode handler remains for each duplicated interpreter. The result is a program with the semantics and static structure of the original program; the virtualiztion obfuscation has been removed.

Table 4. The number of LLVM instructions: after lifting, after optimization without deobfuscation, and after deobfuscation and optimization. The baseline is the number of LLVM instructions obtained by compiling the unobfuscated program with `clang`.

	Lifted	Optimized	Deobfuscated	Baseline
`eq`	2,362	152	35	38
`fib`	3,163	210	63	43

To evaluate our deobfuscation approach, we implemented a virtualizer that supports a set of bytecode instructions. We then created a source-to-source virtualization-obfuscated version of two simple programs: `eq` checks if two arguments match, and `fib` computes the n-th Fibonacci number. Table 4 shows how deobfuscation affects the size of the recovered code. We attain a code size close to that of IR recovered from the unobfuscated binary. Figure 7 depicts the `fib` program, showing its control flow graph obfuscation and subsequent deobfuscation.



(a) Original **(b)** Obfuscated **(c)** Deobfuscated

Figure 7. Deobfuscation of the `fib` program. The control flow graph structure of the deobfuscated binary matches that of the original bytecode, rather than that of the interpreter, which indicates the control flow obfuscation was successfully removed by the analysis implemented in BinRec.

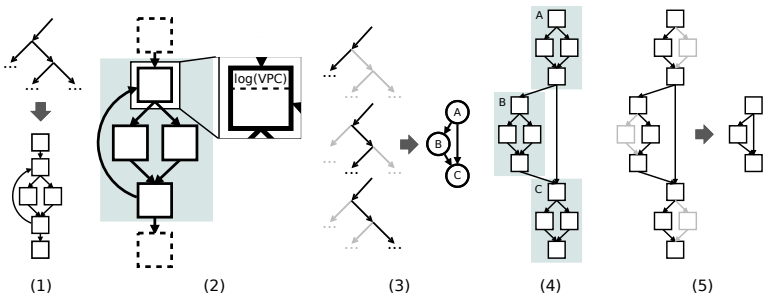


Figure 6. Our deobfuscation approach. (1) We lift the binary using symbolic execution or high-coverage inputs. (2) We identify the lifted interpreter loop and instrument it to log the virtual program counter (VPC) at the entry. (3) The instrumented binary is exercised for all uncovered code paths, yielding a control-flow graph of VPC nodes. (4) The interpreter loop is copied into each VPC node. (5) Standard optimizations eliminate non-taken paths in each VPC node.

6.3 AddressSanitizer

AddressSanitizer (ASan) is a widely deployed bug finding tool that detects spatial and temporal memory errors [51]. It consists of an LLVM instrumentation pass and a run-time monitor. The ASan instrumentation pass identifies and registers memory allocations, and inserts checks for memory accesses. For binaries lifted by BinRec, all memory reads and writes are identified and instrumented automatically using the unmodified ASan instrumentation pass. Heap allocations (e.g., `malloc` or `new`) are recorded in the BinRec lifting frontend and rewritten in the recovered IR, making them visible to ASan. We leave the identification of stack and global allocations for future work as the problem is currently unsolved for binaries. While ASan has been applied to binaries recently [27], we note that this required a reimplementation of both the analysis and instrumentation passes—a substantial disadvantage in maintainability compared to BinRec. Our recovered IR enables the use of ASan to detect spatial and temporal heap access violations. We used two test programs containing (1) a heap use-after-free error and (2) an out-of-bounds write and lifted both test programs in BinRec before applying ASan, successfully discovering these errors.

6.4 SafeStack

SafeStack is a compiler-based transformation pass that separates sensitive data, such as return addresses, and potentially insecure data, such as large application buffers, into separate stacks [36]. If memory isolation features such as x86 segmentation or Intel Memory Protection Keys are available, they are used to isolate the two stacks. If hardware features are unavailable, SafeStack leverages ASLR to hide the safe stack, requiring attackers to bypass ASLR in order to corrupt sensitive data.

By default, BinRec generates programs which contain two stacks with SafeStack-like security properties. The native stack contains sensitive data such as register spills and return addresses, as well as any new instrumentation and library code frames. The emulated stack, which contains the stack data of the original binary, resides at an ASLR-randomized location.

We were additionally able to apply SafeStack's transformations to recovered programs without requiring any modifications to its analysis or transformation passes, since BinRec lifts programs to well-formed LLVM IR. After the SafeStack transformation, recovered programs therefore contain three stack-like memory regions. The native stack contains library frames and newly added safe variables. The emulated stack, at an ASLR-randomized offset, emulates the original binary stack. A third stack in a separate x86 memory segment contains new, potentially insecure buffers. We do not identify stack variables within the original binary, which impedes the transformation's ability to move unsafe buffers from the emulated stack to the third, segmented stack (see Section 7).

7 Limitations

Our prototype implementation of BinRec can only handle single threaded x86 ELF binaries. There are no theoretical limitations on threaded-ness or architecture; the constraint comes from the engineering effort required to implement inline assembly snippets, mostly for library code interfacing. Supporting other binary formats such as PE is no fundamental problem, but requires reimplementing binary stitching in the new format. Additionally, it would require a modest amount of engineering effort to implement per-thread, thread-safe (for multi-processing) data structures to collect separate dynamic traces from each thread in our dynamic lifting engine.

We did not implement handling of self-modifying code. To support it, we would need to add 'version labels' to each recovered code address. This would take some additional lifting time (because code cannot be cached), and complexity while merging traces into one CFG.

BinRec does not recover a mapping between stack slots and variables. Such a mapping would improve optimization and allow more fine grained instrumentation by transformations such as SafeStack and ASan. SecondWrite [4] determined such a mapping for a limited set of input binaries using heuristics, but we leave the determination of a general procedure as future work.

8 Related Work

Low-level binary analysis and rewriting Many projects target the problem of low-level binary analysis and rewriting. PEBIL [37], UQBT [17] and Uroboros [62] all statically rewrite binary programs either at the machine code level or using a custom low-level IR. Their main aim is to support the insertion of simple instrumentation, where efficiency is more important than the ability to perform complex code transformations (such as altering the CFG). angr [55] supports static and dynamic analysis techniques, including symbolic execution, but does not target code rewriting (unlike BinRec). Earlier work such as ATOM [28], PLTO [50], Diablo [46], and Vulcan [57] are powerful tools, but to our knowledge they do not work well without debug symbols. Also, they typically do not support a generic compiler-level IR.

Bauman et al. [7] disassemble instructions from every offset of code sections, creating a superset of all possible disassemblies. They statically rewrite binaries without heuristics by preserving the superset of disassemblies, such that only the legal part of the rewritten binary will be executed at run time. However, deferring correct disassembly until runtime adversely affects rewritten binary performance. Yardimci and Franz [66] use a mostly static approach to automatically vectorize loops in stripped binaries. The approaches

of both Yardimci and Bauman both use an indirect branch table which maps original program addresses to rewritten program addresses to support indirect control flow. BinRec uses a similar indirect branch table for external callback support, but it generates more optimal code because only those callbacks which are actually invoked need branch table entries and control flow graph entry points in rewritten code.

Code transformations using dynamic traces Dynamic instrumentation tools such as PIN [39], Dyninst [10], DynamoRIO [2] and Valgrind [43] are dynamic binary translation (DBT) tools, providing runtime APIs to analyze and instrument code at run time. These tools do not support saving the changes to an output binary with the intent of replacing the original binary. They can have substantial runtime overhead [44], and require specific assembly-level transformation passes for each application, whereas BinRec leverages existing techniques present in production compilers.

Just-in-time (JIT) compilers such as V8 [30] and SpiderMonkey [42] collect dynamic traces to determine which code to optimize and to speculate dynamic data types. Sulong [48] is a frontend for the Graal compiler that effectively creates an LLVM bitcode execution engine. Similar to BinRec, Sulong optimizes LLVM bitcode using dynamic traces and applies instrumentation such as bounds checks to detect safety violations. However, such source-level JIT compilation approaches leverage language semantics and thus do not address the problem of binary lifting or analysis. Instead, they focus on solving a different set of problems such as how to optimize dynamic type checks or when to trigger different tiers of execution.

Binary code lifting LLBT [53, 54] statically retargets binaries to different ISAs after lifting them to LLVM IR. McSema [26], Dagger [9], Rev.ng [25] and RevNIC [15] (based on S^2E) and SecondWrite [4] lift machine code for the purpose of high-level static binary translation on LLVM IR.

HQEMU [32] extends QEMU's back-end to lift code to LLVM IR similarly to S^2E, for the purpose of optimization. It does not decouple lifted code from the QEMU runtime to produce a standalone executable binary, like BinRec.

This paper extends our own prior work, a short workshop paper [35] that presents a high-level idea of dynamic binary lifting. This prior work constructs a rewritten binary from a single dynamic trace, which in turn fails to produce a binary that covers a whole targeted input corpus. The extended version of BinRec presented in this paper addresses this issue i) by stitching multiple, parallel traces into a single executable binary; ii) by incrementally recovering missing basic blocks and control flow edges from the original binary. Compared to our previous work, BinRec shows evaluation results with the complete set of SPEC CINT2006 benchmarks, with significant performance improvement due to our new optimized alias analysis. Furthermore, the prior work was solely targeted for attack surface reduction and it does not present a mechanism to modify the dynamic linkage of their input binaries, limiting code instrumentation. This extended work, on the other hand, shows effectiveness with a rich set of applications including virtualization-deobfuscation, AddressSanitizer, SafeStack, and a control-flow hijacking defense. Moreover, it outlines unsolved challenges of static disassembly in the context of binary lifting.

9 Conclusion

We presented BinRec, a new solution for binary lifting based on dynamic analysis. BinRec lifts a program to compiler-level intermediate code for ease of analysis, while ensuring that it can still compile the result to executable code. Compared to existing static analysis-based techniques, BinRec can seamlessly handle indirect control flow transfers, handwritten assembly and obfuscations. We designed BinRec to overcome the coverage issue of dynamic analysis by using trace merging and incremental recovery. We demonstrated the powerful applications made possible by BinRec: recovering program semantics of virtualization-obfuscated binaries, and applying compiler-level optimizations and hardening transformations to stripped binaries.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their feedback. This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124 and FA8750-15-C-0085, by the United States Office of Naval Research (ONR) under contract N00014-17-1-2782, by the National Science Foundation under awards CNS-1619211 and CNS-1513837, and by the Netherlands Organisation for Scientific Research through grant NWO 639.023.309 VICI "Dowsing". Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA) or its Contracting Agents, the Office of Naval Research or its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government. The authors also gratefully acknowledge a gift from Oracle Corporation.

References

[1] CodeVirtualizer. https://www.oreans.com/codevirtualizer.php.
[2] DynamoRIO. https://dynamorio.org.
[3] VMProtect. https://vmpsoft.com/.
[4] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *Eurosys*, 2013.
[5] Bertrand Anckaert, Mariusz Jakubowski, and Ramarathnam Venkatesan. Proteus: virtualization for diversified tamper-resistance. In *ACM DRM*, pages 47–58, 2006.

[6] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *USENIX SEC*, 2016.

[7] Erick Bauman, Zhiqiang Lin, and Kevin W Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *NDSS*, 2018.

[8] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX ATC*, 2005.

[9] Ahmed Bougacha, Geoffroy Aubey, Pierre Collet, Thomas Coudray, Jonathan Salwan, and Amaury de la Vieuville. Dagger: Decompiling to IR. https://llvm.org/devmtg/2013-04/bougacha-slides.pdf, April 2013.

[10] Bryan Buck and Jeffrey K Hollingsworth. An API for runtime code patching. *IJHPCA*, 2000.

[11] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Comput. Surv.*, 2017.

[12] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[13] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX SEC*, 2015.

[14] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *ISCC*, 2006.

[15] Vitaly Chipounov and George Candea. Reverse engineering of binary device drivers with RevNIC. In *EuroSys*, 2010.

[16] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. *S2E: a platform for in-vivo multi-path analysis of software systems.* 2012.

[17] Cristina Cifuentes and Mike Van Emmerik. UQBT: Adaptable binary translation at low cost. *Computer*, 2000.

[18] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[19] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *ACM POPL*, pages 184–196, 1998.

[20] Kevin Coogan, Gen Lu, and Saumya Debray. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *CCS*, 2011.

[21] Jonathan Corbet. User-space page fault handling. https://lwn.net/Articles/636226/, 2015.

[22] Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. Link-time binary rewriting techniques for program compaction. *ACM TOPLAS*, 2005.

[23] Saumya Debray, Robert Muth, and Matthew Weippert. Alias analysis of executable code. In *POPL*, 1998.

[24] Alessandro Di Federico and Giovanni Agosta. A jump-target identification method for multi-architecture static binary translation. In *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*, pages 1–10. IEEE, 2016.

[25] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. Rev.Ng: A unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, pages 131–141, New York, NY, USA, 2017. ACM.

[26] Artem Dinaburg and Andrew Ruef. McSema: Static translation of x86 instructions to LLVM. In *ReCon 2014 Conference, Montreal, Canada*, 2014.

[27] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *S&P*, 2020.

[28] Alan Eustace and Amitabh Srivastava. ATOM: A flexible interface for building high performance program analysis tools. In *USENIX TCON*, 1995.

[29] Patrice Godefroid, Michael Levin, and David Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[30] Google. V8. https://v8.dev.

[31] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. Performance, correctness, exceptions: Pick three. In *Binary Analysis Research Workshop*, 2019.

[32] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. Hqemu: A multi-threaded and retargetable dynamic binary translator on multicores. In *CGO*, 2012.

[33] R. Nigel Horspool and Nenad Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 1980.

[34] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In *USENIX SEC*, 2015.

[35] Taddeus Kroes, Anil Altinay, Joseph Nash, Yeoul Na, Stijn Volckaert, Herbert Bos, Michael Franz, and Cristiano Giuffrida. Binrec: Attack surface reduction through dynamic binary recovery. In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*, 2018.

[36] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *OSDI*, 2014.

[37] Michael A Laurenzano, Mustafa M Tikir, Laura Carrington, and Allan Snavely. Pebil: Efficient static binary instrumentation for linux. In *ISPASS*, 2010.

[38] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *ACM CCS*, pages 290–299, 2003.

[39] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[40] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. Ccfi: Cryptographically enforced control flow integrity. In *CCS*, 2015.

[41] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *2007 IEEE Symposium on Security and Privacy (SP '07)*, 2007.

[42] Mozilla. Spidermonkey. https://ftp.mozilla.org/pub/spidermonkey/prereleases/60/pre3, 2018.

[43] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[44] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: a practical binary optimizer for data centers and beyond. In *CGO*, 2019.

[45] Manish Prasad and Tzi-cker Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *USENIX ATC*, 2003.

[46] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *ISSPIT*, 2005.

[47] Chenxiong Qian, Hong Hu, Mansour A. Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. RAZOR: A framework for post-deployment software debloating. In *USENIX SEC*, 2019.

[48] Manuel Rigger, Roland Schatz, Rene Mayrhofer, Matthias Grimmer, and Hanspeter Mossenbock. Sulong, and thanks for all the bugs: Finding errors in c programs by abstracting from the native execution model. In *ASPLOS*, 2018.

[49] Rolf Rolles. Unpacking virtualization obfuscators. In *WOOT*, 2009.

[50] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. PLTO: A link-time optimizer for the intel ia-32 architecture. In *WBT*, 2001.

[51] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX ATC*, 2012.

[52] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic reverse engineering of malware emulators. In *S&P*, 2009.

[53] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. LLBT: an LLVM-based static binary translator. In *CASES*, 2012.

[54] Bor-Yeh Shen, Wei-Chung Hsu, and Wuu Yang. A retargetable static binary translator for the ARM architecture. *TACO*, 2014.

[55] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *S&P*, 2016.

[56] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS*, 2011.

[57] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical report, Microsoft Research, 2001.

[58] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive cfi. In *CCS*, 2015.

[59] Victor van der Veen, Enes Goktas, Moritz Contag, Andre Pawlowski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *S&P*, 2016.

[60] Chenxi Wang, Jack Davidson, Jonathan Hill, and John Knight. Protection of software-based survivability mechanisms. In *IEEE/IFIP DSN*, pages 193–202, 2001.

[61] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *NDSS*, 2017.

[62] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *USENIX SEC*, 2015.

[63] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012.

[64] Zhenyu Wu, Steven Gianvecchio, Mengjun Xie, and Haining Wang. Mimimorphism: A new approach to binary code obfuscation. In *CCS*, 2010.

[65] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. In *S&P*, 2015.

[66] Efe Yardimci and Michael Franz. Mostly static program partitioning of binary executables. In *ACM TOPLAS*, 2009.

[67] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX SEC*, 2018.

[68] Mingwei Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *USENIX SEC*, 2013.