2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 15-16, 2024

libtcg

Accurate lifting of executable code using QEMU

Anton Johansson, rev.ng Labs S.r.l., Malmö, Sweden (*anjo@rev.ng*)

Alessandro Di Federico, rev.ng Labs S.r.l., Milan, Italy (*ale@rev.ng*)

***Abstract*—In this work, we introduce *libtcg*[1], an easy-to-use library able to lift executable code to an architecture-independent Intermediate Representation (IR). Being based on QEMU, libtcg inherits its wide and well tested Instruction Set Architecture (ISA) support. Furthermore, the IR exposed by libtcg is independent of the input ISA and explicitly represents how each instruction affects the CPU state. This makes it particularly suitable for building general static analysis tools such as register dataflow visualizations or stack boundary identifiers. As a consequence effort can then be spent on advanced and sound analyses rather than dealing with the quirks of each supported ISA.**

Keywords—static analysis; intermediate representations; QEMU

I. Introduction

Lifting, or converting a binary from architecture-specific executable code to an architecture-independent Intermediate Representation (IR) has become a crucial preliminary step for modern tooling in software verification and reverse engineering. Everything from binary rewriters (zipr [1]), instrumentation frameworks (valgrind [2]), and decompilers (Ghidra, IDA Pro, rev.ng [3, 4, 5]), to full system emulators (QEMU [6]), require a solid foundation in the form of an accurate lifter with wide Instruction Set Architecture (ISA) support.

On the other hand, creating such a lifter for modern CISC architectures is a notoriously challenging task, and often a significant barrier to entry for the development of new tools. Looking at modern binary rewriters, few support more than x86 and ARM (Appendix A), and even mature tools such as valgrind has incomplete x86 support (no SSE4/AVX) with maintainers expressing concern over supporting multiple architectures[2] .

QEMU, in particular, functions both as a dynamic binary translator (user-mode) and full system emulator (system-mode) with support for a large set of ISAs and device models [6]. This is accomplished by (1) lifting the guest executable code to its own internal IR called Tinycode; and (2) recompiling Tinycode to the host architecture and; (3) executing the compiled code. Compared to other contemporary lifters, QEMUs ISA support is rivaled only by projects such as IDA Pro and Ghidra. However, QEMUs lifter is extremely robust, well tested, and accurate enough to facilitate booting fully fledged operating systems[3], making it an ideal base for static analysis tools to build upon.

II. libtcg

A. Design Goals

libtcg is a minimal subset of user-mode QEMU compiled as a shared library, whose goal is to expose an architecture-independent and accurate representation of the underlying ISA, decoupled from the QEMU emulation engine. Consequently, static analysis tools employing libtcg can effectively handle multiple architectures, allowing developers to focus on the actual analyses rather than modelling the semantics of each instruction in the target ISA. In terms of an API, libtcg has 3 goals in mind, users need to be able to: (1) request lifting to Tinycode of a given buffer of executable code; (2) manipulate the list of Tinycode instructions lifted by libtcg, and; (3) request lifting to Tinycode of different ISAs in the same process.

[1] Available at https://github.com/AntonJohansson/libtcg
[2] https://valgrind.org/info/platforms.html, accessed 18-04-2024.
[3] For a collection of varied bootable disk images, see the QEMU advent calendar: https://qemu-advent-calendar.org/2023/

B. API Design and Implementation

The core API of libtcg implementing design goals (1) and (2), is shown in listing 1. This small set of functions is responsible for all interaction with QEMU. Specifically, `libtcg_translate_block` takes as argument, a user-provided buffer of executable code at a specified address, and returns a `LibTcgTranslationBlock` containing the lifted Tinycode with access to source/destination operands for each instruction. "translation block" is QEMU terminology and refers to a basic block in the input ISA. Hence, multiple calls to `libtcg_translate_block` might be necessary.

Listing 1: Core API of `libtcg`

```
// LibTcgContext is opaque and stores information specific to a
// libtcg version
LibTcgContext *libtcg_context_create(LibTcgDesc *desc);
void           libtcg_context_destroy(LibTcgContext *context);

// Translate one block of code buffer[size] at a given address
LibTcgTranslationBlock libtcg_translate_block(LibTcgContext *context,
                                              const unsigned char *buffer,
                                              size_t size,
                                              uint64_t virtual_address,
                                              uint32_t translate_flags);

void libtcg_translation_block_destroy(LibTcgContext *context,
                                      LibTcgTranslationBlock block);

// Functions to retrieve information about the current instruction, or the
// architecture (offset in the cpu state of important registers
// pc, sp, bp, ...).
const char *     libtcg_get_instruction_name(LibTcgOpcode opcode);
LibTcgHelperInfo libtcg_get_helper_info(LibTcgInstruction *insn);
LibTcgArchInfo   libtcg_get_arch_info(LibTcgContext *context);
```

Additionally, flags passed to `libtcg_translate_block` control different aspects of translation, see listing 2 for possible values. In order, the presence of `LIBTCG_TRANSLATE_ARM_THUMB` indicates that the input buffer contains ARM Thumb code. Next, `LIBTCG_TRANSLATE_OPTIMIZE_TCG` tells libtcg to run the Tinycode optimizer after lifting each translation block. The optimizer performs constant propagation, liveness analysis on Tinycode variables, and dead code elimination; resulting in simplified Tinycode. Lastly, `LIBTCG_TRANSLATE_HELPER_TO_TCG` will be covered in later section when helper functions are discussed.

Listing 2: Flags accepted by `libtcg_translate_block`

```
LIBTCG_TRANSLATE_ARM_THUMB,     // Specify that input buffer contains
                                // ARM Thumb code
LIBTCG_TRANSLATE_OPTIMIZE_TCG,  // Run TCG optimizer after lifting
LIBTCG_TRANSLATE_HELPER_TO_TCG, // Use auto-translated TCG helpers,
                                // if available
```

C. Multi-Architecture API

Lifting multiple ISAs within the same process (design goal (3)) is further complicated by limitations imposed by QEMU. Due to the highly target-dependent nature of the QEMU code base, a separate libtcg library is required for each supported ISA. To facilitate working with multiple ISAs, a `dlopen`-friendly API is also exposed, wrapping the core API in function pointers. As an example consider listing 3 where libtcg is dynamically loaded for the MicroBlaze architecture.

Managing multiple instances of libtcg is still cumbersome, especially for longer running processes. As a remedy, a libtcg-loader library is provided which handles `dlopen`-ing and management of state for multiple instances of libtcg, the API is shown in listing 4.

Listing 3: Example usage of `libtcg`'s `dlopen`-based API

```c
#include <libtcg/libtcg.h>

// load libtcg for e.g. microblze */
void *handle = dlopen("libtcg-microblaze.so", RTLD_LAZY);
// A single libtcg_load function needs to be looked up */
LIBTCG_FUNC_TYPE(libtcg_load) *libtcg_load = dlsym(hanldle,
                                                   "libtcg_load");
// The libtcg_load function returns a struct of function pointers
// corresponding to the libtcg API.
LibTcgInterface libtcg = libtcg_load();

// Use as previously ...
LibTcgContext context = libtcg.context_create();
LibTcgTranslationBlock insns = libtcg.translate_block(context,
                                                      buffer, size
                                                      virtual_address,
                                                      0 /* flags */);
```

Listing 4: `libtcg-loader` API for dealing with multiple instances of libtcg within the same process

```c
// For a given LibTcgArch, return the LibTcgInterface
// and LibTcgContext. Create them if it's the first
// time opening an architecture.
void libtcg_open(LibTcgArch arch,
                 LibTcgInterface *libtcg,
                 LibTcgContext **context);

// Close a given libtcg library
void libtcg_close(LibTcgContext *context);
// Close all open libtcg libraries
void libtcg_close_all(void);
```

D. *Dealing with Helper Functions*

In general, lifters will run into trouble when encountering an instruction which cannot be cleanly modeled in its IR. Consider `idiv rcx` on x86_64 which performs integer division between `rax` and `rcx`, and stores the quotient to `eax` and remainder to `edx`. Exceptions are triggered if `ecx` is zero or the quotient overflows. In Tinycode, this corresponds to `call idivl_EAX rcx`, where `idivl_EAX` is a regular C function in QEMU, referred to as a helper.

Helper functions exists to deal with more complicated behavior like system calls, and importantly have free reign over the CPU registers and may read or write to any one of them. Unfortunately, this means very few assumptions can be made as to the state of registers when a helper call is crossed, further complicating static analysis. While QEMU does provide information on whether or not a helper function reads or writes registers, we cannot tell which registers are affected. Moreover, helper functions are especially prevalent in QEMU and are used to model floating point-, certain vector-, and 128-bit integer operations.

To mitigate the impact of helper functions, the `LIBTCG_TRANSLATE_HELPER_TO_TCG` flag may be supplied to libtcg, which enables the use of automatically generated Tinycode implementations of various helper functions. See listing 5 for the Tinycode generated for the `idivl_EAX` helper. The automatic translation is performed during QEMU build time and uses an experimental LLVM-based tool, called *helper-to-tcg* which is not yet available in the QEMU upstream repository [7]. While verbose, the generated Tinycode more accurately describes the behavior of the helper function. Notably, two paths through the code result in `call raise_exception` corresponding to the previously mentioned overflows, and the modification of `rdx`, `rax` is also clearly seen.

Listing 5: Resulting Tinycode for the `call idivl_EAX rcx` instruction when using `LIBTCG_TRANSLATE_HELPER_TO_TCG`.

```
mov_i64 loc0, rcx
mov_i32 loc9, rax
[...]
// Conditional branch checking if rcx == 0
brcond_i32 rcx, $0, ne, $L0
call raise_exception env, $0, $0x7a3c7d89ad00, $0x7a3c7da94178, $0x401
set_label $L0
[...]
// Compute quotient and remainder
div2_i64 loc16, tmp17, loc10, tmp17, loc12
add_i64 loc13, loc16, $2147483648
// Conditional branch checking if quotient overflowed
brcond_i64 loc13, $4294967296, ltu, $L2
call raise_exception env, $0, $0x7a3c7d89ad00, $0x7a3c7da94178, $0x401
set_label $L2
[...]
mov_i32 rax, loc9
mov_i32 rdx, loc11
```

III. LIBTCG-BASED STATIC ANALYSIS TOOLS

Usage of libtcg is demonstrated through a series of *simple* multi-architecture static analysis tools. In order of increasing complexity, the following analyses were implemented: an ISA-independent instruction printer; a Control Flow Graph (CFG) printer; a register dataflow visualizer, and; a maximum stack boundary indentifier. Excluding parsing of ELF headers and printing of results, the core of each analysis was implemented in 50, 140, 540, and 650 lines of C code, respectively. Each tool operates on a buffer of executable code, provided either as an ELF section, a function name, a region from a file specified with offset and size, or lastly as raw bytes from `stdin`[4]. All example programs that follow were compiled without optimization.

In general, disassembly of an ISA with variable length encoding, such as ARM+Thumb, is notoriously complex. Therefore, libtcg does not aim to solve the problem of correctly identifying the location and architecture of the code to lift, but instead focuses on making analyses easy to implement, given a buffer of code.

A. *ISA-Independent Instruction Printer*

With the goal of producing Tinycode equivalent to the executable code in the input buffer, the buffer was linearly scanned, lifting instructions to Tinycode one translation block at a time. Next, each translation block was serialized to `stdout` in order of translation. As an example, see the basic 32-bit ARM code in listing 6 and its corresponding Tinycode in listing 7.

Listing 6: Basic 32-bit ARM program

```
.text
.global _start
_start:
    mov r0, #1234
    mov r7, #0x15
    swi 0
```

Listing 7: Tinycode corresponding to listing 6

```
---- 0000000000000000
mov_i32 r0,$0x4d2

---- 0000000000000004
mov_i32 r7,$0x15

---- 0000000000000008
mov_i32 pc,$0xc
call exception_with_syndrome,$0x8,$0,env,$0x2,
                             $0x46000000
```

One drawback of the approach to code region identification taken in this analysis, is the inability to deal with ARM code that switches from 32-bit to Thumb mode, or vice versa. These regions will not be correctly indentified through a linear scan. Instead, control flow directed lifting would be needed where each lifted translation block would be scanned for possible branches to find new code regions.

B. *Control Flow Graph Printer*

Building on the instruction printer, the input buffer is again lifted to a list of translation blocks. Next, to produce a CFG, all direct jumps to code that has already been lifted are turned into edges, splitting blocks in the process if the destination address is in the middle of translation block. Finally, the CFG is serialized to the DOT language for visualization with Graphviz [8]. Listing 8 shows a simple Hexagon program for computing the sum of the natural numbers from 0 to 10, and Figure 1 shows the produced CFG, clearly reproducing the loop structure.

Note that the choice not to handle indirect jumps means that jump tables and function calls in the input architecture are not correctly identified. Similarly, direct jumps to code that has not been lifted are not handled in the CFG. Again, this is the problem of code region identification, and not an inherent limitation of libtcg.

[4] Source code availabe at: https://github.com/AntonJohansson/libtcg-examples

Listing 8: Hexagon program which sums the natural numbers from 0 to 10

```
.text
.global _start
_start:
    {
        r0 = #0
        r1 = #0
    }
.loop:
    {
        r0 = add(r0, r1)
        r1 = add(r1, #1)
    }
    {
        p0 = cmp.gt(r1, #10)
        if (!p0.new) jump:t .loop
    }

    r6 = #94
    trap0(#1)
```

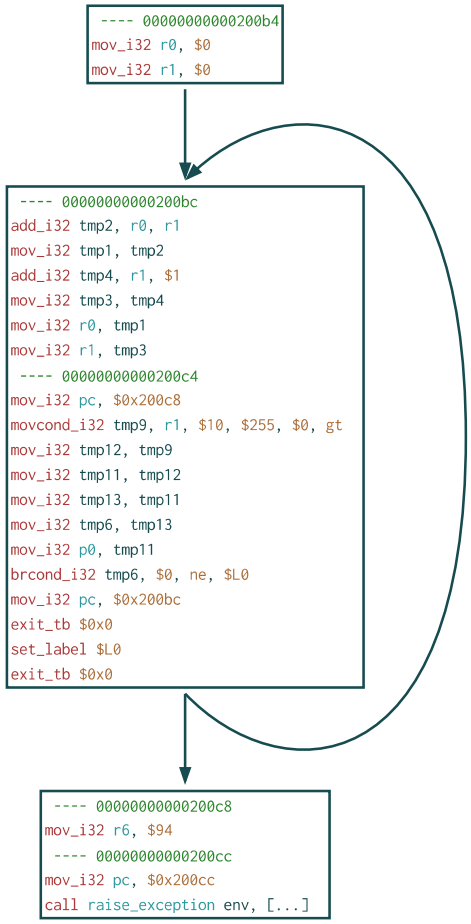

Figure 1: CFG produced for the Hexagon program in Listing 8

C. *Register Dataflow Visualization*

Implemented as a backward dataflow analysis (see Ref. [9]) over the CFG recovered by the previous analysis, the goal is to construct a tree of all possible instructions which may affect the value of a given Tinycode operand. Whenever an instruction that writes to the given operand is found, the operands of that instruction are added to a list of new values to search for; values that are found are removed from the list. This list gets propagated backwards through the CFG and across stack loads and stores until a fixed point is reached. Stack loads and stores were identified by running the same analysis on the pointer operand of the memory operations, and the resulting tree was folded to an offset from the stack register, if possible. All connected stack loads and stores could then identified on the CFG.

As an example, consider listing 9 containing a simple C program to sum the natural numbers from 0 to `n`, and Figure 2 with the resulting visualization. The program was compiled for 64-bit RISCV, and the analysis starts from the `x15/a5` operand to the `mov_i64 x14/a4, x15/a5` instruction (see instruction marked with borders). Identified source instructions are colored based on the destination operand with the exception of stores which are colored based on the pointer operand. Inspecting the figure, it becomes clear that `x15/a5` in the body of the loop corresponds to the value of `sum` that is loaded from the stack. The operand being analyzed also depends on the instructions that follow it, which loads the loop induction variable `i` and performs the addition `sum + i` before storing the result to the stack again.

In terms of limitations, the analysis will not be able to explore sources from non-stack loads, or from stack loads where the pointer operand cannot be reduced to an offset from the original stack pointer, such as stack addresses that depend on a runtime value. Additionally, sources which are not available on the CFG will be missed, including helper calls and indirect jumps. Therefore a conservative approach is taken where all indirect jumps and calls to helpers that affect the cpu state are assumed to be sources of the operand.

Listing 9: C program for testing register dataflow visualization.

```c
int test_reg_src(int n) {
    int sum = 0;
    for (int i = 0; i < n; ++i) {
        sum += i;
    }
    return sum;
}
```

Listing 10: C program for testing maximum stack size analysis.

```c
int test_max_stack(int n) {
    int arr[] = {1,2,3,4};
    if (n < 4) {
        return arr[0];
    } else {
        return arr[1];
    }
}
```

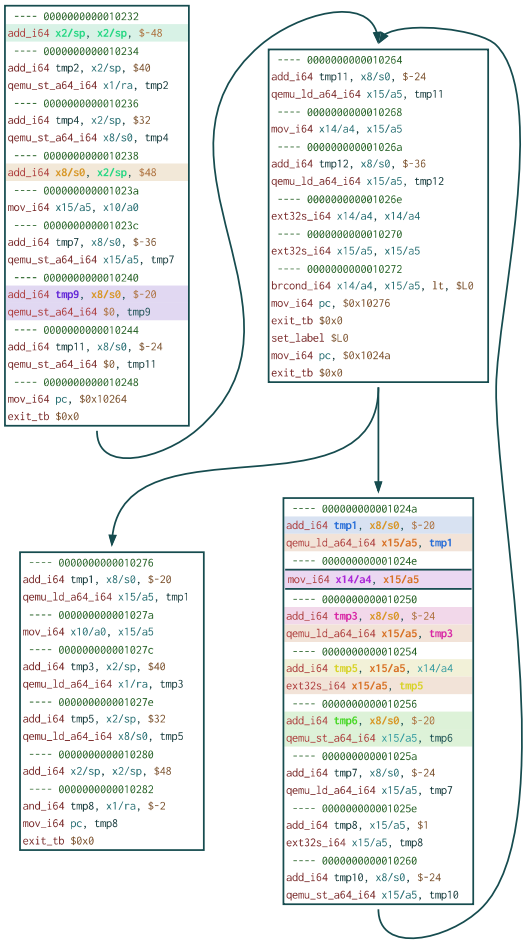

Figure 2: Register dataflow visualization of the `x15/a5` operand. Analyzed program is given in Listing 9, compiled for RISV64.

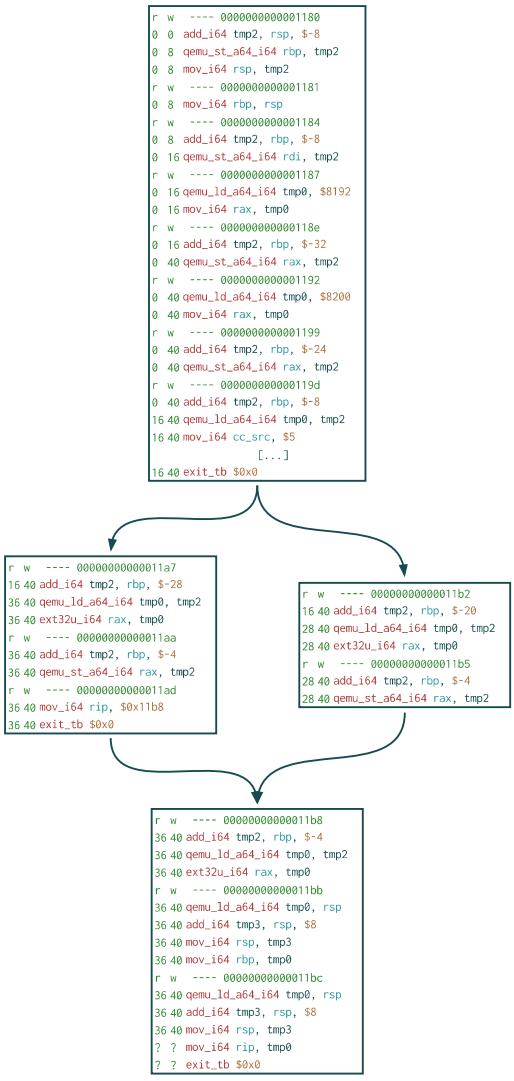

Figure 3: Maximum stack size analysis of the program in Listing 10, compiled for x86_64.

D. *Maximum Stack Size Calculator*

A forward dataflow analysis (see Ref. [9]) over the CFG was created with the goal of making a conservative estimate of the maximum stack offset that has been read or written for each instruction in the program. Iterating over the CFG, stack loads and stores were identified in the same manner as the previous analysis, hence the same limitations apply. Next, whenever a stack operation was detected, that offset would be propagated to subsequent instructions, taking the maximum along the way if a new stack offset was found. The final stack offset detected for each translation block was propagated forward through the CFG and used as a starting point for following translation blocks. This process was carried out until fixed point.

See listing 10 for a simple C program consisting of an if statement along with stack reads and writes. Figure 3 shows the CFG produced by the analysis, when ran on the C program compiled for x86_64, with the maximum stack read and write offsets annotated in the margin. Examining the figure, the program starts by pushing `rbp` and the argument `rdi` to the stack, followed by loading the initial array values from constant addresses and storing them on the stack, resulting in a maximum written stack offset of 40. Note that array values are stored as 64-bit integers. Finally, the function argument is read from offset 16, and each branch reads from offsets 28 and 36, respectively.

IV. Testing

The ISA-independent instruction printer was tested against a set of 749 QEMU v8.2.1 user-mode test binaries spread across 16 architectures[5] and produces Tinycode for 716/748 (96%) of the binaries. All of the 32 failures are due 32-bit ARM programs switching between Thumb and normal code, which is not supported when lifting with a linear scan as discussed previously. As for the more complicated analyses, they were manually tested and verified against simpler binaries for x86, ARM, Hexagon, and RISCV.

V. Future Work

As it currently stands, libtcg is split into a "core" API consisting of a shared library per ISA, and an optional common "multi-architecture" API. This is a consequence of the highly coupled nature of the QEMU codebase, and results in increased code duplication and a larger maintainence burden. As a solution, we are currently involved in an upstream effort to modularize QEMU by factoring out as much target-independent code as possible into separate libraries[6]. libtcg would then depend only on the target frontends. In addition to simplifying libtcg, this is a crucial step towards emulating heterogeneous systems consisting of multiple cores of different architectures. Heterogeneous emulation is especially important in modeling a complete system on a chip, similar to what Xilinx accomplished with their ARM/MicroBlaze QEMU fork [10].

Lastly, despite usage of helper-to-tcg to automatically translate helper functions, many of them cannot be expressed directly in Tinycode. Thankfully, most helper functions are well structured with bounded loops, making them susceptible to static analysis techniques. Therefore, all registers accessed by a helper function should be statically identifiable, and can be exposed to libtcg for usage in analyses.

VI. Conclusion

To summarize, we have introduced libtcg, an accurate and easy-to-use lifter based on QEMU with wide ISA support. This allows developers to focus on creating the actual analyses, rather than modelling ISAs. On top of libtcg, four static analyses were implemented for demonstration purposes, and tested on 16 architectures with widely different characteristics. All without writing a single line of architecture specific code. The analyses range in complexity from dumping an ISA-independent representation to `stdout`, to finding the maximum stack offset accessed at any given point in the program. Lastly, different avenues for mitigating the limitations inherited by QEMU were discussed, including efforts towards heterogeneous emulation and accurate tracking of registers modified by helper functions.

[5] Tested architectures: x86, SPARC, SH4, s390x, RISCV, PPC, Nios2, MIPS, MicroBlaze, m68k, LoongArch, HPPA, Hexagon, CRIS, Alpha, ARM

[6] Roadmap: https://wiki.qemu.org/Dynamic_machine_and_heterogeneous_emulation_roadmap

APPENDICES

A. *Architecture support among modern lifters and binary rewriters*

Table 1: Supported architectures and executable formats for a selection of binary rewriters and lifters

Tool	Format	x86_32	x86_64	ARM32	ARM64	MIPS32	MIPS64	SPARC32	SPARC64	SH4	MSP430	RISCV64	S390X
e9patch [11]	ELF, PE	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
ddisasm [12]	ELF, PE	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗
mctoll [13]	ELF	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
miasm [14]	ELF, PE	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗
rellume [15]	ELF	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗
remill [16]	ELF, PE	✓	✓	✗	✓	✗	✗	✓	✓	✗	✗	✗	✗
reopt [17]	ELF	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
retrowrite [18]	ELF	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
revng [5]	ELF, PE	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓
zipr [1]	ELF, PE	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗

REFERENCES

[1] W. H. Hawkins, J. D. Hiser, M. Co, A. Nguyen-Tuong, and J. W. Davidson, “Zipr: Efficient static binary rewriting for security,” in 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 559–566, 2017.

[2] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” in Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’07, (New York, NY, USA), p. 89–100, Association for Computing Machinery, 2007.

[3] National Security Agency (NSA), “Ghidra.” Available at https://ghidra-sre.org/.

[4] Hex-Rays, “IDA Pro.” Available at https://www.hex-rays.com/products/ida.

[5] A. Di Federico, P. Fezzardi, and G. Agosta, “rev.ng: A multi-architecture framework for reverse engineering and vulnerability discovery,” in 2018 International Carnahan Conference on Security Technology (ICCST), pp. 1–5, 2018.

[6] F. Bellard, “Qemu, a fast and portable dynamic translator,” in Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC ’05, (USA), p. 41, USENIX Association, 2005.

[7] A. Johansson and A. Di Federico, “Automatic promotion of helper functions to tcg using llvm,” Presented at KVM forum 2023. Available at https://github.com/revng/qemu-upstream/tree/feature/helper-to-tcg.

[8] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, Graphviz and Dynagraph — Static and Dynamic Graph Drawing Tools, pp. 127–148. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[9] F. Nielson, H. R. Nielson, and C. Hankin, Principles of Program Analysis. Berlin, Heidelberg: Springer-Verlag, 1999.

[10] Xilinx, “Xilinx qemu.” Available at https://github.com/Xilinx/qemu.

[11] G. J. Duck, X. Gao, and A. Roychoudhury, “Binary rewriting without control flow recovery,” in Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020, (New York, NY, USA), p. 151–163, Association for Computing Machinery, 2020.

[12] A. Flores-Montoya and E. Schulte, “Datalog disassembly,” in 29th USENIX Security Symposium (USENIX Security 20), pp. 1075–1092, USENIX Association, Aug. 2020.

[13] S. B. Yadavalli and A. Smith, “Raising binaries to llvm ir with mctoll (wip paper),” in Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2019, (New York, NY, USA), p. 213–218, Association for Computing Machinery, 2019.

[14] CEA IT Security, “miasm.” Available at https://github.com/cea-sec/miasm, accessed 18-04-2024.

[15] A. Engelke, Optimizing Performance Using Dynamic Code Generation. PhD thesis, Technical University of Munich, 2021.

[16] Trail of Bits, “remill.” Available at https://github.com/lifting-bits/remill, accessed 18-04-2024.

[17] “reopt.” Available at https://github.com/GaloisInc/reopt, accessed 18-04-2024.

[18] S. Dinesh, N. Burow, D. Xu, and M. Payer, “Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization,” in 2020 IEEE Symposium on Security and Privacy (SP), pp. 1497–1511, 2020.