Running a 64-bit ARM Application on 32-bit Hardware: Static Recompilation of a Real-Time DJ Player from AArch64 to ARMv7

Erhan

draft, November 2026

Abstract

We describe a64port, an ahead-of-time binary translator that turns AArch64 Linux programs into C and compiles that C for 32-bit ARMv7 processors. We built it to run the user-space application of a commercial DJ media player, a dynamically linked C++ program with 266 threads, a 45.9 MB executable and 24 shared libraries, on a different DJ player whose Rockchip RK3288 has four 32-bit Cortex-A17 cores. No interpreter or JIT runs on the device. We translate the guest's own dynamic linker, C library and C++ runtime along with the application, and a host loader plays the part of the kernel. QEMU removed support for 64-bit guests on 32-bit hosts in version 10.0, citing the memory space and atomicity requirements of 64-bit guests; we show how a static translator handles both on a 32-bit host, where it falls short, and which bugs only appeared on real hardware. We replace hot functions with native "leaves" and test each one differentially against the original machine code, using generated states, states captured on the device, and deliberately broken leaves that the harness has to reject. On the RK3288 the translated application draws at its own 54.4 Hz frame rate with no audio underruns in a 20-minute soak and 4.8 ms from announced playhead to audible output, and DJs have played on it. The same toolchain runs the applications of two other players of the same product family.

1 Introduction

A DJ media player is an appliance whose software is tied to one piece of hardware: the manufacturer ships a Linux system on a 64-bit ARM chip, and its one important program, a C++ GUI application with a real-time audio engine, runs nowhere else. We wanted to run that program on a different DJ player, an older device with comparable controls whose Rockchip RK3288 only implements the 32-bit ARMv7 architecture. Both devices run Linux on ARM, so the operating system interface is close. The instruction set differs, and so does the word size, which turned out to be the harder problem.

The existing tools do not cover this direction. Box64 and FEX translate onto 64-bit hosts only. QEMU's user mode emulator could run AArch64 programs on 32-bit ARM hosts for many years, until QEMU 10.0 removed 64-bit guests on 32-bit hosts with the explanation that "32-bit hosts cannot support the memory space or atomicity requirements of 64-bit guests" [1]. The static translator Lasagne [6], the dynamic translator Risotto [7] and the static recompilers of the console preservation community [2, 3] all target hosts at least as wide as the guest. A dynamic translator would also cost CPU time on a chip that is already much slower than the original.

a64port translates every AArch64 module of the program to C once, on a workstation, and compiles the result for ARMv7 with clang. The translation engine comes from the `arm64_to_c.h` emitter of suyu's mk8-recomp project [2], which we use as an instruction decoder and C emitter; our changes grew the header from 7,897 to 10,195 lines. Our contributions lie in the runtime, loader and verification around that engine, and in applying it to a complete commercial application:

- A complete static translation of a large, closed-source, dynamically linked and heavily multi-threaded AArch64 application, including its dynamic linker, glibc and libstdc++, onto a 32-bit host, with real-time requirements met on the target hardware (Sections 3 and 8).
- A systematic account of fitting a 64-bit Linux process into a 32-bit one: address space, system call and structure translation, threads, atomics and floating point, with the defects that only showed up on real hardware (Section 4).
- A measured sequence of code generation changes that took the translated program from 95 % to 31 % of an x86-64 core on a fixed workload, with ARMv7 instruction counts for the same steps and the changes that made it slower (Section 5).
- A workflow for replacing hot functions with native code and testing each replacement differentially against the original machine code, with a lifter for whole functions and call trees, replay of states captured on the device, and deliberately broken leaves that the harness has to reject (Section 6).

The translated application first ran on the host hardware on 15 October 2026, played a track there on 17 October, and played it at the correct speed on 18 October; the two other players of the family followed by early November. DJs have since played sets on the translated players.

2 Setting

The guest. The program we translate (we call it App A) is the user interface and audio engine of a pro-

fessional DJ media player. It is a non-PIE AArch64 executable of 45.9 MB with 27.1 MB of code, linked against glibc 2.29 and 18 directly needed libraries; their closure is 24 shared objects plus the dynamic linker. It is written in C++ on the JUCE framework, draws its interface through X11 into a shared-memory image, runs its audio engine at 96 kHz, decodes compressed audio ahead of the playhead in background threads, applies a time-stretching algorithm when the tempo changes with key lock enabled, and talks to other players over Ethernet with the vendor's link protocol. Buttons, the jog wheel and the LEDs sit behind a sub-CPU that the application reads over an SPI character device in fixed 64-byte frames (62 bytes of state and a CRC-16). The track database on a USB stick is an encrypted SQLite database. During playback the process has 266 threads.

We also translated the applications of two other players of the same family: App B (the successor of App A's player) and App C (a smaller model). We introduce them where they matter and summarise them in Section 8.

Hardware. The original player uses an RK3399-class 64-bit SoC with two Cortex-A72 and four Cortex-A53 cores. The host is another manufacturer's DJ player built on a Rockchip RK3288: four Cortex-A17 cores at up to 1.6 GHz with NEON and VFPv4, 2 GB of RAM, a 32-bit ARMv7 Linux kernel and userland. Its kernel boots with `isolcpus=1,2,3`, so every thread that is not pinned explicitly runs on CPU 0. Its vDSO provides no clock (Section 7), so every `clock_gettime` is a system call. We replace its own player software for the duration of a session and reuse its display, touch panel, audio output, buttons and network.

Goal. The translated application must behave like the original: correct audio at the right speed with no underruns, a smooth waveform display, low audio latency, and working link features such as tempo synchronisation with other players. The work is an interoperability project. We do not distribute any vendor code; a user supplies the firmware of a player they own (Section 11).

3 Overview

Figure 1 shows the pipeline. A profile names the executable, the dynamic linker and every library in the closure. The driver `a64recomp` translates each module into its own C project: one C file per group of functions, the module's sections as binary segments at their link addresses, and a layout header with sections, relocations and program headers. A generator reads all layouts and writes one module table for the host. clang compiles everything for the Cortex-A17 together with our runtime and loader into one statically linked ARMv7 executable.

3.1 Translation

The engine decodes one instruction at a time and emits C that operates on a guest context: 31 general registers, 32 vector registers of 128 bits, the flags and the floating-point control and status registers. Loads and stores go through memory helpers. A basic block becomes a C function, or part of one. In the engine's original design every branch returned to a dispatcher that maps a guest address to its block; Section 5 describes how the product build avoids most of these round trips.

Upstream, the engine skips instructions it does not handle and leaves them to a JIT. We have no fallback on the device, so we made every unhandled instruction fatal and set the engine's "translate all" flag unconditionally. That flag alone removed 1,107 of the 1,245 genuine decoding gaps on App A (among 5,287 untranslated words), all of them shifts by immediate that upstream had left to the JIT.

Emitted functions carry names. For App A we recovered 36,739 function names, 3,238 from the ELF's own symbols and the rest from an automated analysis of the binary, and the emitter uses them in the C identifiers (Figure 3). Profiles and crash reports then name guest functions instead of addresses.

3.2 Finding the code

A static translator has to find every block the program can reach. We build one code image from all executable sections (`.init`, `.plt`, `.text`, `.fini` and glibc's `__libc_freeres_fn`), because a static binary routes its IFUNC-resolved calls through the PLT. Discovery starts from the entry point, every `.eh_frame` function start and every dynamic symbol, and we added three sources of roots: switch tables found in `.rodata` (2,022 roots, 84 new block starts on App A), relocation addends that point into code, and absolute code pointers in the data sections of non-PIE images (2,409 roots, 467 new block starts). Switch tables matter because 87 of 727 switch targets in App A are not otherwise block starts.

On App A the 64,799 roots from `.eh_frame` and the dynamic symbols alone lead to 1,918,629 blocks and 6,756,944 instructions; with the extra roots, all executable bytes that are not zero padding are covered (99.77 % of the code image). The engine leaves 4,180 instruction words untranslated (0.062 %). Of these, 2,233 are padding, 1,809 are data that discovery walked into, and 138 decode as named instructions. Almost all of the 138 are SVE encodings of data; only a handful of sites (`FMOV` between general and vector registers, `LDNP` and `STNP`) are instructions the emitter lacks. With `RBLIFT_GAP_DUMP` set, the runtime records every address that execution reaches but discovery missed, and every untranslated instruction it reaches, and writes a root file that the next emit reads as extra roots.

3.3 Loading and dynamic linking

Our first loader pre-bound symbols itself instead of running the guest's dynamic linker. It failed on ordinary code paths: glibc's `_dl_addr` calls through a lock pointer that libpthread's initialiser installs, and libpthread is marked to initialise first for that reason. We replaced it with a loader that does what a kernel

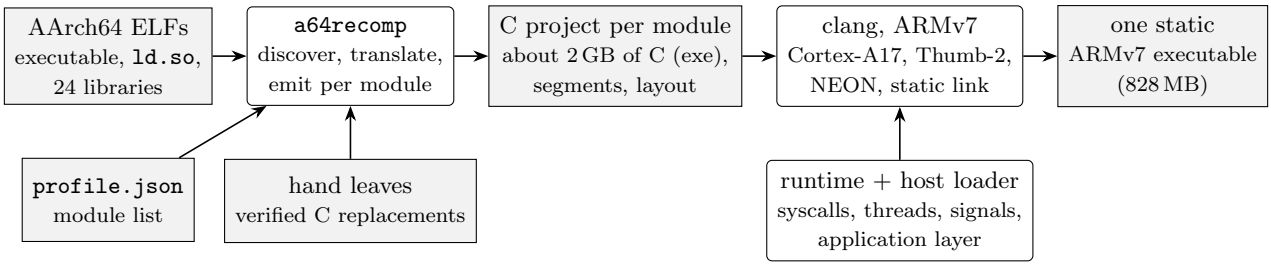

Figure 1: The a64port pipeline. Translation happens once per module on a workstation; only the C compiler step depends on the host. On the device the executable receives the original root file system as its argument.

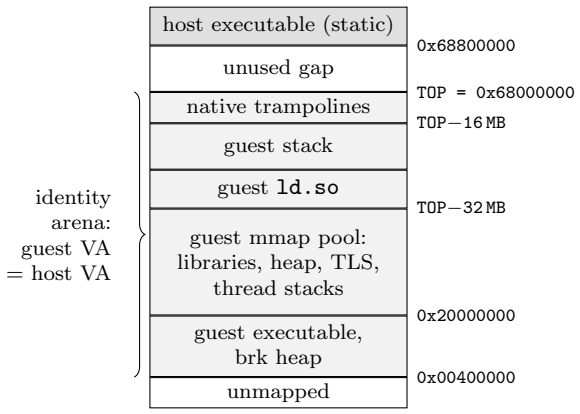

Figure 2: Host address space on the RK3288. The guest occupies one identity-mapped arena below the host executable, so a guest address is a host address.

does. It maps the executable and the translated dynamic linker, builds the initial stack with arguments, environment and an auxiliary vector (program headers, a baseline `AT_HWCAP` of FP and ASIMD so that glibc selects no optional-extension code paths, no vDSO), and starts the guest's own `ld.so`. The guest dynamic linker then maps the libraries, applies their relocations, resolves IFUNCs and runs the initialisers, all as translated code. Each module's translated blocks are registered under its load base, so a guest call into a library lands in that library's translation. The loader also puts `LD_BIND_NOW=1` into the guest environment: every PLT slot is then resolved before the first thread starts, and lazy binding never runs under the dynamic linker's lock while 266 threads compete for it.

4 Fitting a 64-bit process into 32 bits

4.1 Address space

The guest believes it has a 64-bit address space; the host has 4 GB, of which the kernel keeps one. We give the guest an identity-mapped arena from `0x400000` to `0x68000000` (Figure 2). The executable is non-PIE and loads at its link address, and the guest's `ld.so` sits near the top of the arena. The runtime's `mmap` hands out addresses for libraries, heap and thread stacks top-down from a pool between the interpreter and `0x20000000`. The host executable is linked at `0x68800000`, above the arena, and linked statically. Under qemu-user, which we use for testing, a dynamically linked host binary gets its own libraries mapped inside the arena range before `main`; on the device the static link removes any dependence on the board's C library. The armhf sysroot has no `libgcc` and our clang has no ARM compiler runtime, so we supply the few helpers the static link needs (integer division, 64-bit conversions and unwinder stubs).

Guest registers stay 64 bits wide in the context, and guest pointers stay 64-bit values. Because every guest address lies below 4 GB, a load or store truncates the address to a host pointer and accesses memory directly, without a page table. This "fastmem" arrangement cut CPU time on our fixed workload from 95 % to 62 % of a core (Table 1), the largest single step in Table 1. Fastmem has two costs. Nothing checks guest addresses any more, so an access outside the arena faults or reaches host memory; a build that routes every access through checking helpers remains our debugging build. Plain stores also no longer pass through a helper that could clear exclusive reservations, which shaped the monitor design in Section 4.3.

The 1.6 GB arena forced some economies. Guest file mappings become real host file mappings at the same address (on x86-64 this alone reduced the resident size from 2.94 GB to 1.00 GB). We report a 1 MB `RLIMIT_STACK` to the guest, which glibc uses as its default thread stack size, and give the host threads 512 KB stacks. The loader limits glibc to four malloc arenas, each of which reserves 64 MB of address space, and the application layer shrinks a 2 GB stream cache to 256 MB on 32-bit hosts. With these, App A runs at a flat 1.14 GB resident set over a 20-minute session.

4.2 System calls

A 64-bit AArch64 process and a 32-bit ARM EABI process use different system call numbers and, for many calls, different structure layouts. The runtime implements about 146 AArch64 system calls. Each one converts every structure whose layout differs between the two ABIs, field by field, and calls the host C library or the host kernel. A guest pointer is an address in the arena, so buffers whose layout matches pass through without copying. Calls the runtime does not know return `ENOSYS`.

Most of the defects we found in this layer were silent. Two of them show the pattern:

- The guest's 64-bit `struct flock` has the layout of the 32-bit ABI's `flock64`, so the runtime passed it through, but with the plain `F_GETLK`, `F_SETLK` and `F_SETLKW` commands. A 32-bit kernel reads those as the older 16-byte structure, so the lock offset came partly from a padding word. The kernel returned

EINVAL at random, SQLite reported "disk I/O error" (19 times in one run), and track titles and artwork went missing now and then. For several days we took this for a bug of the application itself. The fix maps the commands to their 64-bit variants through fcntl64.

- gettid had the x86-64 system call number, so on ARM every guest TID was −1 and priority-inheritance futexes failed with ESRCH; a futex timeout's timespec was also truncated.

The application also asks the hardware for things it does not have on the host. We keep everything specific to one application in an *application layer*, a directory of C fragments on the include path (Section 7). Without it the runtime builds as a generic AArch64 Linux runtime.

4.3 Threads and synchronisation

Each guest thread runs on its own host thread, and the guest sees the host thread ID as its TID. The runtime honours the CLONE_PARENT_SETTID, CLONE_CHILD_SETTID and CLONE_CHILD_CLEARTID flags and a process-wide exit_group, and passes futex wait, wake, requeue and priority-inheritance operations to the host kernel. Priority-inheritance futexes needed the host kernel's real implementation: our earlier emulation answered a contended FUTEX_LOCK_PI with EAGAIN, which glibc 2.29 treats as success, so two threads could hold the same mutex. After the fix we count more than 10,000 contended PI locks in a 58-second run.

The AArch64 memory model is weaker than x86's and comparable to ARMv7's. We emit barriers and acquire/release accesses as C11 atomics, which clang lowers to dmb on ARMv7, and implement LSE atomics and compare-and-swap as host atomic operations. Exclusive loads and stores (LDXR/STXR) went through several designs:

1. the engine's single global reservation raced between threads; we replaced it with a per-thread reservation checked by a value compare-and-swap;
2. our instruction fuzzer (Section 6.3) showed that a store to the reserved address must clear the reservation even when it writes the same value;
3. with 266 threads, we built a shared monitor: a slot table, per-granule stripe locks held across check and store, and a version counter that plain stores bump. In the same step the engine's no-op barrier helper and its unfenced LDAR/STLR became real fences;
4. the shared monitor completed STXR with a plain store, which could overwrite a PI futex word the kernel had just updated, so the store became a host compare-and-swap;
5. fastmem removed the store helpers that bumped versions, so the shipped build uses a per-thread value-CAS monitor again, the model qemu-user uses. It gives up the rule from step 2 and loses only an intervening store of the same value (ABA); the shared monitor remains for the checked build;
6. the stripe locks that remain, for pair and misaligned exclusives, use priority inheritance, because audio threads run at real-time priority.

64-bit exclusives become 64-bit host compare-and-swap operations, which clang emits as LDREXD/STREXD loops. The 128-bit pair forms (LDXP/STXP) have no ARMv7 equivalent. STXP compares both halves and stores them under one of 256 hashed stripe locks, which is atomic against other pair exclusives on the same address but not against plain stores or single-word atomics on the same bytes. We have not seen the application depend on that case.

The standalone runtime that runs our test programs delivers guest signal handlers at the next system call return of the receiving thread. The multi-module host that runs App A honours only the default and ignore dispositions, because the application never needed a handler to run, and keeps the fault signals (SIGSEGV, SIGBUS, SIGABRT and others) for its own crash reporting. Neither runtime delivers synchronous faults to guest handlers or supports sigaltstack.

4.4 Floating point

The engine's default floating-point code computes every operation with integer arithmetic, bit for bit as AArch64 does, including NaN propagation, rounding and the status flags. This makes results identical across hosts but costs time, and the audio engine and the time stretcher are mostly floating-point code.

We use a fast path that runs an operation on the host's VFPv4 unit and keeps the result only when it provably equals the architectural one (default rounding mode and no trap enables, finite normal operands and results, and only once the sticky inexact flag is already set; almost every operation raises inexact, so once it is set the fast path never has to compute a status flag); otherwise the exact body runs. QEMU's hardfloat [13, 14] follows the same principle. Two details were specific to this application. JUCE sets flush-to-zero around all audio processing, so the fast path refuses subnormal values instead of requiring FZ to be clear. On App C, the audio thread spent 20 % of its time in the exact fused multiply-add: the state of an audio filter decayed into subnormal numbers, and 99.9 % of the slow calls had a subnormal input. An opt-in variant flushes subnormal inputs and results as FZ prescribes and uses the host's fused multiply-add for every finite result. Values stay bit-exact; only the bookkeeping of the input-denormal and underflow flags is approximate, and nothing in the program reads them. We measured the effect before we found that App C's engine runs at 44.1 kHz, with the card wrongly at 48 kHz and the audio thread overloaded: there the variant lowered the audio thread from 114 % to 96 % of a core and removed the underruns (67 in 40 seconds). At the correct rate the audio thread needs 47 %.

5 Code generation

The engine's original output gives every basic block its own C function and returns to the dispatcher after every branch. That shape is easy to verify, but every branch costs a return and a table lookup. We changed it step by step and measured each step on a fixed work-

load: boot, load a track, play for 61 seconds, browse. Table 1 shows the cumulative result on x86-64, where we could measure quickly, and in ARMv7 instructions retired for small benchmark programs under `qemu-arm`, since x86-64 hides much of what the later steps save. Table 2 shows where the time went at four of these steps. In the starting point, the memory helpers cost almost as much as the translated code itself; fastmem removed nearly all of that. The next step, which made guest calls host calls and added the global resolver and the value-CAS monitor, removed the dispatcher and most of the remaining helper cost, and the later steps shrank the translated code by a further sixth.

Calls and chaining. With chaining, a direct branch becomes a C tail call into the target block instead of a return to the dispatcher (3.8× on a dispatcher-bound loop; 8 % on a screen switch of the real application). With `--callret`, a guest `BL` becomes a host call, bounded by a call-depth counter, and `RET` becomes a plain C `return`. After the call returns, the caller compares the guest program counter with the expected return address: if they match it continues directly in the same C function, otherwise it returns to the dispatcher.

The host's return predictor then works for guest code, and the C stack is only a cache of the guest call stack: `longjmp`, C++ unwinding and thread exit leave a frame whose return address no longer matches, and execution falls back to the dispatcher. Direct tail calls use clang's `musttail`, so chains never grow the host stack. Computed targets (`BR`, `BLR` and the resume after a system call) go through one global table over all modules, indexed by the guest address in three levels (4 MB region, 4 KB page, one slot per instruction) and filled on the dispatcher's first miss, so a call through the PLT into another library stays a host call. Figure 3 shows the result.

Functions. We group the blocks of each guest function, using `.eh_frame` and symbol boundaries, into one C function in which internal branches become `goto`. App A has 64,793 functions. Our first version allowed entry at any block of the group through a switch at the top. That made many loops irreducible, clang then kept no guest register in a host register across iterations, and grouped code ran up to 2.5× slower than per-block code. The final form gives a grouped function exactly one entry, its start; every block also keeps its own per-block function for the rare entries in the middle (a dispatcher lookup, an unwind). The cold copy costs size: the emitted C grows from about 1.3 GB to about 2 GB. Functions above 1,024 instructions or 512 blocks stay per-block, because the compiler's optimiser and the assembler are super-linear in function size (we measured this with GCC at `-O1`). A later version of the emitter raises the cap for an explicit list of hot functions; on App B the waveform column builder (1,785 instructions) then dropped from 26 % to 14 % of the GUI thread.

Flags. AArch64's calling convention does not preserve the condition flags across calls, so flag values are dead at `BL`, `BLR` and `RET`. A module-wide backward liveness analysis over direct successors, in which an unknown target counts as reading every flag, removes 727,901 of App A's 995,131 flag computations. Figure 3 shows a comparison that computes only the two flags its branch reads.

Smaller changes. Marking the context pointer `restrict` tells clang that guest memory never aliases the context, so a guest store does not force registers to be reloaded. A block function begins with an interior-entry test that sends an entry in the middle of the block to the right instruction; on a 32-bit host that test can compare program counters in 32-bit arithmetic. For a hand-picked set of hot blocks, vector instructions operate on real 128-bit vector types instead of per-lane loops (2.25× on a pixel-blending block). Leaving out the per-block self-modification check and its alternate entry functions removed 15.7 % of the emitted text; App A never rewrites its code.

Changes that did not pay. Our first grouping also kept guest registers in C local variables within a function. We verified it differentially, and it made a screen switch 17 % slower than the chained per-block build (180.4 against 154.3 ms, median of 10). Writing back only changed registers recovered 4 %; we attribute the rest to grouping itself, most likely the irreducible-loop effect described above, but we never measured grouping without locals on the application. On an FP-heavy micro-benchmark, locals alone cost 16 %, because the temporaries of the exact FP bodies spilled. The first version of the FP fast path was slower on x86-64 at `-O1` on an FP-only micro-benchmark (154 against 108 ms). Removing the interior-entry checks altogether broke the application.

Build. The App A emit produces about 2 GB of C (Table 6), about 1.3 GB before single-entry functions. A full ARMv7 compile of the 1.3 GB emit took about nine minutes on a workstation, and a build including the emit about fifteen; the emitter writes a file only if its content changed, so an engine change recompiles only the units it affects. The final executable is 828 MB. Sampling an earlier build on the device showed 99 % of the time in 3,586 blocks occupying 2.93 MB of the 637 MB of code, so we link hot functions first, in profile order.

6 Verified native leaves

After the code generation work, a handful of functions dominated each busy thread: pixel blending and waveform drawing in the GUI, the time-stretch analysis, the sample-rate converter, and the MP3 decoder. We replace such functions with native C we call *leaves*. A leaf is marked in its source with the module and guest address it replaces, and the emitter wires it in; a leaf that matches no block stops the emit. A full leaf replaces the block. A filter leaf may decline, and

Step (cumulative)	x86-64 CPU %	ARMv7 instructions retired (10^9) calls	chain	load	fp
Starting point: chained per-block code, clang -O2	95	1.61	2.41	4.91	3.82
Fastmem: identity arena, direct loads and stores	62				
Guest calls as host calls, global resolver, value-CAS monitor	46	1.52	2.20	4.82	3.63
FZ-safe FP fast paths, shared exact FP bodies	41				
Flag dead-code elimination, batched vector FP	40	1.48	1.70	4.58	5.96
`restrict` context, 32-bit PC arithmetic	39	1.44	0.55	4.44	5.92
One C function per guest function, single entry	31	1.28	0.55	4.34	5.86

Table 1: Cumulative effect of code generation changes. The x86-64 column is CPU time of the whole application over a 61-second play window. The ARMv7 columns are instruction counts of four benchmark guests (calls through function pointers, a one-block loop, a memory loop, an FP loop whose values overflow to infinity) under `qemu-arm`; blank cells were not measured at that step. The starting point already used block chaining and the FP fast path. The FP benchmark is a worst case for shared exact bodies, because every operation takes the exact path and the shared body is a call. In the session that measured the last row, the previous row re-measured at 41 %.

```
18426b0: stp  x29, x30, [sp, #-0x80]!
...
18426c4: mov  x19, x0
...
18426e4: str  x1, [x19]
18426e8: bl   0x19324b0
18426ec: str  x21, [x19, #0xc50]
...
18426fc: add  x0, x19, #0xc70
1842700: cmp  x20, x0
1842704: b.hi 0x1842710
```

```
void fn_a_18426b0(GuestContext* __restrict c){
L_18426b0:;
    { uint64_t _b=c->x[31]; int64_t _o=-128;
      recomp_stp64(c,(_b+_o),c->x[29],c->x[30]); c->x[31]=_b+_o; }
    ...
_L18426b0_5:;  { c->x[19] = c->x[0]; }
    ...
_L18426b0_13:; recomp_store64(c,c->x[19] + 0,c->x[1]);
_L18426b0_14:; c->x[30]=GB+0x18426ec;
    { c->pc=GB+0x19324b0;
      if (c->call_depth < RECOMP_MAX_DEPTH) {
        ++c->call_depth;
        fn_a_19324b0(c);
        --c->call_depth;
        if (c->pc != GB+0x18426ec) return;  /* else: dispatcher */
        goto L_18426ec; }
      if (RECOMP_BUDGET(c)) return;
      RECOMP_MUSTTAIL return fn_a_19324b0(c); }
L_18426ec:;    recomp_store64(c,c->x[19] + 3152,c->x[21]);
    ...
_L18426ec_5:;  { uint64_t _a=c->x[20],_b=c->x[0],_r=_a-_b;
                 c->z=(_r==0); c->c=(_a>=_b); }  /* b.hi: Z, C */
_L18426ec_6:;  if (c->c && !c->z) { c->pc=GB+0x1842710; goto L_1842710; }
               else { c->pc=GB+0x1842708; goto L_1842708; }
```

Figure 3: A guest function prologue (left, AArch64) and the C that `a64recomp` emits for it (right), abridged from the App A product emit; the comments are ours and the flag temporaries are simplified. The emitter normally adds a recovered name to each identifier; here we show only the address and shorten the module name to `a`. `GB` is the module base, 0 for the non-PIE executable. With fastmem, `recomp_store64` is a plain host store.

CPU (s)	start	fastmem	calls	restrict
Translated code	38.2	39.8	32.6	27.2
Memory helpers	35.5	4.4	0.6	0.5
Dispatcher	5.1	4.6	0.0	0.0
Leaves	7.5	7.4	7.0	6.6
Host libc, libm	5.3	5.2	4.3	4.1
Runtime, other	2.7	2.1	0.6	0.7
Total	94.4	63.5	45.2	39.0

Table 2: Where the time went, from the host-PC sampler, for four rows of Table 1 (whole x86-64 benchmark run: boot, load, play, browse). Columns are the rows of Table 1 named Starting point, Fastmem, Guest calls as host calls, and the row before single-entry functions. These are CPU-seconds over the whole run, including boot and browse; Table 1's percentages cover only the 61-second play window. with them the sampler files grouped functions under the wrong category, so we omit that column.

the translated body runs. A third kind, which we call a continuation leaf (`FILTER_TO`), finishes its work and continues at a fixed guest address, by a `goto` when the target lies in the same C function, which lets a leaf replace one inner loop of a large function; a filter that instead returned a new program counter would unwind every caller to the dispatcher. App A carries 55 leaves: 1 full leaf, 39 filters and 15 of the third kind. Because the emitter finds leaves by scanning source text, `#if 0` does not disable one, and a leaf added without re-emitting the module is silently ignored.

Each leaf checks its preconditions before it changes any state. A typical leaf checks that the floating-point control register has its default value, that the object it works on carries the expected function pointer, that its buffers lie inside the arena and do not overlap, and that the loop bounds are sane; otherwise it declines. Leaves that replace a loop first run the loop's index arithmetic without side effects, so that they either complete the whole loop or decline before changing any state. Figure 4 shows the start of such a leaf.

6.1 Finding candidates

A host-PC sampler driven by per-thread CPU timers splits each thread's time into translated code, memory helpers, dispatch, leaves, floating point, runtime and system calls. A guest-PC sampler attributes time to

```
RBLIFT_LEAF_FILTER_TO(a, 0x18f7b90, 0x18f7d5c) {
  const uint64_t S = c->x[19], sp = c->x[31];
  unsigned char* s = rb_hostspan_id(c, S, 2592);
  if (!s) return 0;                       /* decline */
  if (c->fpcr & UINT64_C(0x07C8FF07)) return 0;
  { uint64_t f; memcpy(&f, s + 2584, 8);
    if (f != 0x18f6070) return 0; }     /* object kind */
  const int64_t rem0 = (int64_t)c->x[20];
  if (rem0 <= 0 || rem0 > (1 << 22)) return 0;
  ...   /* table, coefficients, output:
           in the arena, pairwise disjoint */
  /* dry run of the index arithmetic: inputs
     consumed, ring offsets read */
  ...
```

Figure 4: The guard prelude of App A's resampler loop leaf (abridged from our leaf sources). It returns 0 to let the translated loop run, and after the dry run it never declines.

guest functions. On App B and App C both stopped working: with the host-PC sampler the GUI never painted, and the guest-PC sampler saw only parked threads. We then wrote an external sampler that attaches to one thread with `ptrace` at about 250 Hz, reads its program counter and detaches, and later a `perf_event` sampler with context-switch records that splits time per audio callback. Profiles can mislead: the host-PC sampler put 7 % on a leaf helper, but forcing that helper inline made the thread slower, and only a cheaper variant of it helped.

6.2 Writing leaves

We transcribed most leaves by hand from the machine code and ported some library routines, such as libX11's `_XGetPixel`, from their source code. mpg123's DCT routines, written in AArch64 NEON assembly, are regenerated from that assembly by a small generator as scalar VFP code, because ARMv7 NEON flushes subnormal values to zero and the VFPv4 unit does not; we wrote its synthesis filters by hand. We port leaves to another program of the family by matching machine code (Section 6.5) and lift whole functions or call trees with `a64nat` (Section 6.4).

We also tried twelve leaves that a script generated automatically. All twelve passed our verifier at the time. Six of them caused a heap corruption at start-up and were found by bisection. Of the six that shipped, one, the routine that checks and copies received frames from the button sub-CPU, zeroed bytes 5 to 7 of every frame, where the PLAY, CUE and loop buttons are encoded. Those buttons did nothing on the device for two working sessions until we found it. The verifier had run on the x86-64 host from random register states, and the function only does its work when a length field equals `0x40`, the 64-byte frame size, which random states never hit. We removed all automatically generated leaves and changed how we verify.

6.3 Verification

We have two kinds of verifier. The first, `verify-leaf.sh`, emits the module with and without the leaf and runs the block from the same random guest context in both builds; this is the verifier that passed the automatically generated leaves. The second, which we used for every leaf written after the button incident, runs the original machine code under `qemu-aarch64` as the reference. Its simplest form cuts the function's bytes into a blob at a fixed address and patches its exits; later forms map the module's real code and data. Generators shaped to the function produce the inputs (lengths, flags and object fields in their legal ranges, edge values included); the harness runs the original and the leaf from the same state and compares the registers, the flags and the touched memory, or only the AAPCS64-visible state for a whole function. FP leaves are compared without the cumulative exception flags in FPSR, which they do not reproduce. Each leaf is also built for ARMv7 and run under `qemu-arm`. As a rule, for each leaf we also check that the harness rejects deliberately broken copies (a changed constant, a flipped condition, an unfused multiply-add, a wrong store offset); the scripts report these mutants but do not enforce the rule. A few shipped leaves never went through the qemu harness: a clock leaf on App B and a colour lookup on App C, whose output we checked on the screen, and the system call wrappers in libc. Table 3 lists all layers.

Random states are a poor input for functions that dispatch on object fields, as the button incident showed. For the large time-stretch and drawing functions we therefore capture real entry states on the device: registers, the stack frame, and memory reachable from the arguments up to a configurable pointer depth. For the overlap-add routine (WSOLA, waveform-similarity overlap-add) of the App B time stretcher, 135 captured states and 945 perturbed variants gave no mismatch and covered 88 of 101 labels of the lifted code; the uncovered labels are wrap-around cases of ring indices that playback did not reach. A later harness maps the real executable with its 218 MB of zero-initialised data, plus the math library, relocates the captured regions, and runs whole call trees. Coverage is not always that good: playback reached only 31 of 101 labels of App C's overlap-add routine, and the first states of a track are silence, on which a mutant with an unfused multiply-add passes. Perturbation and coverage counts tell us how much of a function a run exercised, which plain pass counts do not.

`qemu-aarch64` and `qemu-arm` produce default NaNs with different sign bits. Where a leaf can create a NaN, the comparator accepts any NaN in place of a NaN.

6.4 Lifting whole functions

`a64nat` translates one AArch64 function into C with the guest registers as C locals, the floating point as host `float` and `double` (fused instructions become `fmaf` and `fma`), the flags as C locals set by every flag-setting instruction (the compiler removes the unused ones), and branches as `goto`. Its input is the text output of `llvm-objdump`, not the emitter's decoder. A call can go to a listed helper, to another lifted function, or back into translated guest code through the runtime's resolver; with `--abi`, only the argument, result and callee-saved registers are synchronised at calls and returns. The generated leaf declines at run time if the

Layer	Method	What it found
Test programs	11 freestanding and static-glibc programs (files, sockets, epoll, timers, threads, signals, arguments) compared with `qemu-aarch64`, native and as ARMv7 builds	missing `timerfd`, `clone3` fallback, allocator exhaustion under threads
Instruction fuzzer	random instruction sequences and operands, all registers, flags and a 64 KB scratch area against `qemu-aarch64`, native and ARMv7 builds	exclusive reservation not cleared by same-value store; `SDIV` overflow trap
Random-context leaf check	module emitted with and without the leaf, same random context in both	passed twelve automatically generated leaves; six later caused a heap corruption at start-up, one broke the buttons
Leaf differential	original bytes under qemu vs. leaf, shaped generators, 1.2k to 53k cases per leaf, mutants per leaf	a bug in a resampler continuation leaf; a struct offset that moved between versions
Device-state replay	states captured on the device at function entry, replayed with the real executable and libraries mapped, plus perturbed variants; label coverage of the lifted code	fixed-point `fcvtzs` ignored its fraction bits in `a64nat`
Rebuild identity	refactors must leave the preprocessed runtime token-identical and the binary byte-identical	(guards the public split of the code base)
On-device gates	scripted button presses and screenshots on the device: playback, speed against wall time, dropouts, frame rate, browse latency	added after a 48 kHz configuration shipped at half speed

Table 3: Verification layers and examples of defects each one found.

Thread or task	before	after
App A decode per load (CPU-s)	20.6	7.8
App A audio thread, normal play	58 %	43 %
App A key shift, underruns/20 s*	2,024	0
App B time stretcher (% core)	45	18.7
App C time stretcher (% core)	45	14.6
App C whole-track MP3 decode (ticks)	8,316	2,623

Table 4: Effect of leaves on busy threads, measured on the device on the same track and screen before and after. *Worst key. The App B row combines hand-written leaves and `a64nat`.

guest has set a non-default rounding, flush-to-zero or default-NaN mode. The lifter aborts on any instruction it does not support and, since the bug described below, on any conversion whose operands it does not all consume. We applied it to 16 functions of App B and 20 of App C, including the time stretcher's FFTs, its phase vocoder and `atan2f`, `sinf` and `cosf` from the guest's libm.

Table 4 shows the effect of leaves on the threads that mattered. Some leaves did not pay off. A leaf for mpg123's integer dequantisation ran slower than the translated code; register-level transcription of bit-reader code gives clang nothing to improve. One leaf is deliberately not exact. The sample-rate converter's 59-tap dot product is, in the guest, one sequential fused multiply-add chain per channel; the exact leaf follows that order and kept the decoder thread at 100 % of a core, so that a 96 kHz track never filled its page buffer. An optional mode, which the device launcher enables, sums with four independent NEON accumulators instead. Its results differ from the guest's in the last bits of a float, and non-finite results still take the exact path. The dot product in that mode, about 138 cycles per output sample, is limited by the A17's 64-bit NEON datapath, and none of four alternative versions beat it. Lifting App B's waveform column builder with `a64nat` produced a leaf that passed replay and took 8.3 % of the GUI thread against 3.5 % for the translated version (its merge loop excluded): the function makes 15 virtual calls, and each one synchronised the whole register file. Replay of this leaf also found the one `a64nat` bug we know of, a fixed-point conversion that ignored its fraction bits; no shipped function uses that form. On a 32-bit host, native code wins on floating point, where it replaces the exact bodies, and rarely on integer code.

6.5 Porting leaves between versions

Another program of the family has every address in a different place. A porting tool searches the new binary for each old leaf's function by machine code with immediates and PC-relative offsets masked. From App A to App B, 26 of 39 executable leaves had a unique match and 21 passed the stricter gate (identical extent, identical read-only data); 13 library leaves carried over unchanged because 8 of 15 libraries were byte-identical. From App B to App C only 6 of 21 matched, since a different compiler configuration changed register allocation and object layouts. For App C we transcribed some leaves again with the new register assignment, regenerated the MP3 decoder leaves, and lifted the time stretcher with `a64nat`. App B's libX11 leaves broke App C's library browser for a reason we did not find, so App C runs without them. Masking immediates also hides structure offsets: one ported loop read a field that had moved by 8 bytes, and only the differential test caught it.

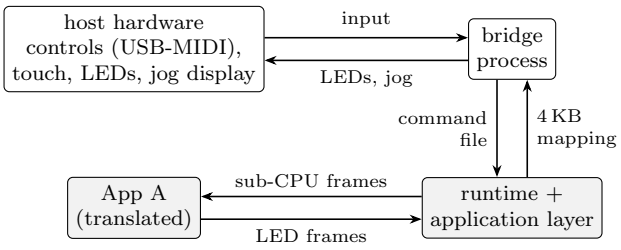

Figure 5: How the host player's hardware reaches the translated application. The application only ever sees its own sub-CPU device.

Measure (App A on the RK3288)	value	build
Waveform display rate	54.4 fps	67–76
Audio underruns, 20-minute soak	0	22
Resident set, threads	1.14 GB, 266	22
Playback speed vs. wall time	1.000	22
Playhead to audible output	4.8 ms	36
Wi-Fi beat jitter (s.d.)	1.3–1.7 ms	59
Track ready after load	about 1.2 s	67
First window, warm/cold	7.6/12.8 s	≈30

Table 5: The translated application on the host player. Each value comes from the gate or measurement of the build named; later builds passed the regression gate and 4-minute soaks.

7 Platform integration

The application layer fabricates the devices that the application expects and the host lacks. The sub-CPU's SPI device produces 64-byte transfers, 62 bytes of state and a CRC-16, from the host player's buttons, jog wheel and touch panel, and accepts LED frames in return; App B and App C read touch from a fabricated event device instead. DRM and GPIO nodes and a sysfs vsync timestamp exist where the application looks for them.

A separate bridge process on the host connects the host player's hardware to that emulation (Figure 5). It reads the host's controls, which its firmware exposes as a USB-MIDI controller, and its touch panel, and writes short text commands (hold or release a button bit, encoder delta, jog position, touch point) to a file that the runtime polls when it composes the next sub-CPU frame. The jog wheels differ: the host's platter gives about 3,165 steps per turn and the original's about 3,240, so the bridge rescales them and a turn stays a turn. The host has no keys for several functions the original has above its screen, and its panel is 80 pixels taller, so the bridge draws those keys in a bar above the application's window. The application must still believe its window sits at the origin, because it maps touch to window coordinates itself; wrappers around the X11 calls that report window positions take care of that. In the other direction, the runtime publishes the LED frame, the last playhead and status packets and the paths of the artwork files the application opened in a 4 KB shared file mapping, from which the bridge drives the host's LEDs and the display in its jog wheel. Audio leaves the program through alsa-lib and reaches the host card through a direct sink, which converts the 96 kHz output to the card's 48 kHz with a 31-tap Blackman FIR filter. The vendor's link daemons, which handle the player network and remote media, are AArch64 programs too, and we translate them with the same toolchain.

Time. The RK3288's boot firmware never sets the virtual counter offset, so the kernel runs its timer on the physical counter, which user space cannot read, and the vDSO offers no clock. Every `clock_gettime` became a system call of about 0.6 μs; on App B the audio thread alone made about 31,700 of them per second. With `RBLIFT_FAST_CLOCK` set, the runtime reads `CLOCK_MONOTONIC` from the virtual counter, which user space can read but which carries a random offset on each CPU. A helper thread calibrates each CPU's offset against bracketing system calls to about 42 ns. The runtime does not know which CPU a reading came from, so it picks the offset that lands nearest the previous value, and a global maximum keeps the clock monotonic. Every 1,024th reading is checked against the kernel, and a deviation above 100 μs switches the fast path off for good.

Real-time behaviour needed tuning that has nothing to do with translation. Because the kernel isolates CPUs 1 to 3, the application ran entirely on CPU 0 until we pinned threads by name (GUI, audio, decoders, time stretcher) to the isolated cores; the frame rate went from 9 to 24 fps. The application publishes its play position every few audio callbacks, so the 512-frame callbacks we had started with held the display to 23 Hz; 64-frame callbacks, the original's own size, raised it to 54 fps. Those numbers were measured with the engine at 48 kHz, which turned out to play every track at half speed. At the correct 96 kHz, 64-frame callbacks cost the audio thread 94 % of a core, so the launcher now asks the application for 192-frame callbacks (2 ms), which keep 54 fps. A related stall came from the audio output: with `aplay`'s default 125 ms periods the application rendered audio in bursts, its play position advanced in jumps, and the time stretcher's display fell to 29 to 36 fps; 2.5 ms periods restored 54 fps. Moving the audio DMA interrupt off CPU 0 cut the worst interval between audio callbacks on App B from 4.56 to 1.72 ms, and the performance CPU governor stopped the on-demand governor from holding the clock at 0.7 to 1.2 GHz. With the fast clock, the governor, and the audio interrupt and X server moved to CPU 2, App A's audio thread went from 36.2 % to 28.7 % of a core and the GUI from 53.9 % to 41.9 %. Finally we retimed the host panel from 60.06 Hz to 54.43 Hz, the rate at which the application paints, and derived the vsync timestamp from the panel's real vertical blank, which removed a periodic judder in the scrolling waveform.

8 Results

How we measured. All measurements ran on the device, driven from a workstation over the network by scripts that press buttons through the bridge and take

	App A	App B	App C
Libraries	24	40	36
Blocks (millions)	1.92	2.67	2.78
Emitted C, executable (GB)	2.1	2.8	2.9
ARMv7 executable (GB)	0.83	1.22	1.24
Leaves	55	57	32

Table 6: Translated firmware generations. Sizes in decimal GB; the emitted C of the libraries adds another 0.5 to 0.9 GB per generation.

screenshots. The frame rate is the number of times a screen region changes per second, read through the X protocol; a second tool records, for every new frame of the scrolling waveform, its interval and how far the picture moved, which shows pacing that a frame count hides. An underrun is an `EPIPE` from the write to the sound card, counted by the audio sink. Playback speed is the advance of the remaining-time display, read by template OCR from screenshots, against wall time; the tool refuses a result when its two halves disagree by more than 2 %, because its first version misread digits and reported a half speed that was not there. Latency is the playhead the player announces on the link minus the track position actually audible, which we find by cross-correlating a tap of the output with the decoded track, to about 1 ms. An earlier estimator based on beat onsets had a bias of about 18 ms; we found it only by running it on a case with a known answer.

Table 5 summarises App A on the host player. The display runs at the rate the application itself targets. The latency from the position the player announces on the link to the sound at the output fell from about 90 ms when audio went through a FIFO to `aplay` to 12.9 ms with a direct audio sink, and to 4.8 ms once the runtime stopped alsa-lib's file plugin from holding a full buffer. The link's beat packets crossed a Wi-Fi access point with 60 to 67 ms of jitter when broadcast; sending them to each player directly brought it to under 2 ms, and tempo synchronisation with other players works in both directions.

Other players. The same emitter and runtime run the applications of two other players of the family, which we call App B and App C (Table 6). Each needed its own sub-CPU frame format, a few model constants and new or ported leaves. App C runs its audio engine at 44.1 kHz and paints its waveform at 30 fps by design; a hook that doubles its display timer, a smoothing leaf and adjusted present timing bring it to 57 to 60 fps on the host panel, although some starts settle at 40 fps for a reason we traced to the X server side but have not fixed. A fourth program, a combined controller and player, translates and starts, then loops on CRC errors from four sub-CPU channels we do not emulate and aborts after about two minutes.

Correctness. Besides the layers in Table 3, we check every refactoring of the translator by rebuilding. Splitting the code base into a generic core and an application layer left the full App A emit identical and the ARMv7 executable byte-identical (same MD5) to the build before the split. The test programs produce the same output as under `qemu-aarch64`, natively and as ARMv7 builds under `qemu-arm`; the thread programs did so in 25 of 25 repeated runs.

Deployment. Several DJs have played sets on the translated players. We have no instrumented data from those sessions.

9 Related work

64-bit guests on narrower hosts. QEMU's TCG supported 64-bit guests on 32-bit hosts until version 10.0 [1], dynamically and with known atomicity problems in user mode. Box86 runs 32-bit x86 programs on 32-bit ARM and Box64 requires a 64-bit host [4]; FEX and the 64-bit ExaGear run on AArch64 only. ForgeARM is an early static AArch64 to ARMv7 translator for Android libraries that we know of only as a proof of concept [5]. We know of no static translator that runs a large dynamically linked 64-bit application on a 32-bit host.

Static binary translation. UQBT [8] translated binaries statically through C, and FX!32 [9] combined emulation with background translation. rev.ng [10] lifts several architectures to LLVM IR and compiles for the build host. Lasagne [6] statically translates concurrent x86-64 programs to Arm with proven fence placement, and Risotto [7] applies verified mappings in a dynamic translator; both target 64-bit hosts and benchmark programs. The console recompilation projects, N64Recomp [3] and the mk8-recomp engine [2] that we build on, translate statically linked games and replace the console operating system with native code; they do not translate a guest dynamic linker or C library.

Validation. Dasgupta et al. [11] validate an x86-64 lifter instruction by instruction and on whole programs. STOKE [12] validates rewritten code on test cases drawn from real executions before proving equivalence, which is close to our replay of captured states. QEMU's hardfloat [13] and Cota and Carloni [14] use the host FPU only where it provably matches softfloat, as our FP fast path does.

Rehosting. Firmware rehosting systems such as Firmadyne [15], HALucinator [16] and Greenhouse [20] run embedded software on other machines for security analysis, on hosts stronger than the device and without real-time requirements. Our aim is the opposite case: running an appliance's application on weaker hardware for real use.

10 Limitations and lessons

The translator has no fallback: an untranslated instruction stops the program. Guest addresses must stay below 4 GB, which holds for the non-PIE executables we target, and the product build does not check

them. The 128-bit exclusive pair is not atomic against plain stores or single-word atomics, the shipped monitor misses same-value ABA sequences, and the multi-module host runs no guest signal handlers. On the device, the sample-rate converter trades exact floating-point results for speed. The runtime builds with clang only. We have not compared against the original hardware with the same tracks, and we have not compared against QEMU 9.2’s user-mode emulator, the last version that supports 64-bit guests on 32-bit hosts. One known audio defect remains: a click when the time stretcher is switched on or off. We have not isolated it; our main suspects are three of the time stretcher’s leaves, and we do not know how the original player behaves in that case.

The defects that cost us the most time were silent and looked like flaky hardware, such as the `flock` layout. On a translated program, a symptom that appears at random is worth checking against a count of failing system calls before anything else. Our verifiers taught us that a passing run says little until the harness has caught mutants and reached the code that matters. Measurements also contradicted our expectations more than once: register locals, native FP on x86-64, a lifted integer function and the dequantisation leaf were all slower than what they replaced.

11 Legal and ethical notes

We did this work to use the software of players we own on other hardware we own. We accepted no licence terms for that software, and the firmware carries no licence agreement of the player’s manufacturer. Everything described here ran on our own devices.

We publish no vendor code, firmware, keys or other material that would let anyone run or unlock the manufacturer’s software without having it lawfully. The short excerpts in Figures 3 and 4 illustrate the method, and we show addresses in place of the manufacturer’s identifiers. The public release of `a64port` contains the translator with the vendored engine, the generic runtime and loader, the leaf SDK, the random-context leaf verifier, the instruction fuzzer, the profilers, the test programs and a tutorial. The engine is licensed GPL-2.0-or-later, and so is the release. A user must supply the AArch64 programs they are entitled to run. The device-specific application layer, the leaves, the qemu replay harnesses and `a64nat` embed knowledge of one firmware and are not part of the release.

For readers interested in the legal background: in the EU, Directive 2009/24/EC allows the lawful acquirer of a program to perform the acts necessary to use it for its intended purpose, including error correction (Art. 5(1); see also *Top System* [17]), allows a lawful user to observe, study and test how a program functions (Art. 5(3)), and permits decompilation where it is indispensable for the interoperability of an independently created program (Art. 6). The Court of Justice held in *SAS Institute* [18] that the functionality of a program and the format of its data files are not protected by copyright. In the United States, 17 U.S.C. §1201(f) contains an interoperability exemption, and *Sony v. Connectix* [19] treated intermediate copying for emulator development as fair use. This paper is not legal advice, and readers should check the rules that apply to them.

`a64port` is not affiliated with or endorsed by any manufacturer of the hardware or software mentioned in this paper.

12 Conclusion

A 64-bit AArch64 application with hundreds of threads and real-time audio runs on a 32-bit ARMv7 device by static translation, at the frame rate and latency the application expects. Beyond the changes to the translation engine, most of the work went into the 64-on-32 runtime, into code generation measured against a real workload, and into replacing a few dozen hot functions with native code that we tested against the original machine code. The public release lets others apply the same approach to their own AArch64 programs.

References

[1] QEMU Project. Removed features: 64-bit guests on 32-bit hosts (removed in 10.0). QEMU documentation, `https://www.qemu.org/docs/master/about/removed-features.html`; commit f3ca7ca2, 2025.

[2] mk8-recomp and the suyu static recompiler `arm64_to_c.h`. `https://github.com/dougchansan/mk8-recomp`.

[3] Wiseguy. N64Recomp: static recompilation of N64 games. `https://github.com/N64Recomp/N64Recomp`.

[4] ptitSeb. Box86 and Box64. `https://github.com/ptitSeb/box86`, `https://github.com/ptitSeb/box64`.

[5] ForgeARM. `https://github.com/stackvoided/ForgeARM`.

[6] R. C. O. Rocha et al. Lasagne: a static binary translator for weak memory model architectures. PLDI 2022.

[7] R. Gouicem et al. Risotto: a dynamic binary translator for weak memory model architectures. ASPLOS 2023.

[8] C. Cifuentes, M. Van Emmerik. UQBT: adaptable binary translation at low cost. IEEE Computer 33(3), 2000.

[9] R. J. Hookway, M. A. Herdeg. Digital FX!32: combining emulation and binary translation. Digital Technical Journal 9(1), 1997.

[10] A. Di Federico, M. Payer, G. Agosta. rev.ng: a unified binary analysis framework to recover CFGs and function boundaries. CC 2017.

[11] S. Dasgupta et al. Scalable validation of binary lifters. PLDI 2020.

[12] E. Schkufza, R. Sharma, A. Aiken. Stochastic superoptimization. ASPLOS 2013.

[13] E. G. Cota. fpu: introduce hardfloat. QEMU commit a94b783952, 2018.

[14] E. G. Cota, L. P. Carloni. Cross-ISA machine instrumentation using fast and scalable dynamic binary translation. VEE 2019.

[15] D. D. Chen, M. Woo, D. Brumley, M. Egele. Towards automated dynamic analysis for Linux-based embedded firmware. NDSS 2016.

[16] A. A. Clements et al. HALucinator: firmware rehosting through abstraction layer emulation. USENIX Security 2020.

[17] Court of Justice of the EU, Case C-13/20, *Top System SA v. Belgian State*, judgment of 6 October 2021.

[18] Court of Justice of the EU, Case C-406/10, *SAS Institute Inc. v. World Programming Ltd*, judgment of 2 May 2012.

[19] Sony Computer Entertainment, Inc. v. Connectix Corp., 203 F.3d 596 (9th Cir. 2000).

[20] H. J. Tay et al. Greenhouse: single-service rehosting of Linux-based firmware binaries in user-space emulation. USENIX Security 2023.