Performance, Correctness, Exceptions: Pick Three

A Failproof Function Isolation Method for Improving Performance of Translated Binaries

Andrea Gussoni
Politecnico di Milano
andrea1.gussoni@polimi.it

Alessandro Di Federico
Politecnico di Milano
alessandro.difederico@polimi.it

Pietro Fezzardi
Politecnico di Milano
pietro.fezzardi@polimi.it

Giovanni Agosta
Politecnico di Milano
agosta@acm.com

Abstract—**Binary translation is the process of taking a program compiled for a given CPU architecture and translate it to run on another platform without compromising its functionality. This paper describes a technique for improving runtime performance of statically translated programs.**

First, the program to be translated is analyzed to detect function boundaries. Then, each function is cloned, isolated and disentangled from the rest of the executable code. This process is called *function isolation*, and it divides the code in two separate portions: the *isolated realm* and the *non-isolated realm*.

***Isolated functions* have a simpler control-flow, allowing much more aggressive compiler optimizations to increase performance, but possibly compromising functional correctness. To prevent this risk, this work proposes a mechanism based on stack unwinding to allow seamless transition between the two realms while preserving the semantics, whenever an *isolated function* unexpectedly jumps to an unforeseen target. In this way, the program runs in the *isolated realm* with improved performance for most of the time, falling back to the *non-isolated realm* only when necessary to preserve semantics.**

The here proposed stack unwinding mechanism is portable across multiple CPU architectures. The binary translation and the function isolation passes are based on state-of-the-art industry-proven open source components – QEMU and LLVM – making them very stable and flexible. The presented technique is very robust, working independently from the quality of the functions boundaries detection. We measure the performance improvements on the SPECint 2006 benchmarks [12], showing an average of 42% improvement, while still passing the functional correctness tests.

I. Introduction

Binary translation is a widely used technique for program analysis. It is most useful to understand the behavior of programs at runtime, on platforms that make unpractical or extremely time-consuming to extract the necessary information during the execution. These situations include analysis of code for a CPU architecture different from the analyst's workstation, or exotic embedded devices that might not be physically available or might not provide the necessary visibility and control over the execution environment. Binary translation is also useful on more traditional architectures, to instrument existing programs for which the source code is not available, enabling the analyst to extract insights about its execution that would be otherwise very though to obtain.

Workshop on Binary Analysis Research (BAR) 2019
24 February 2019, San Diego, CA, USA
ISBN 1-891562-58-4
https://dx.doi.org/10.14722/bar.2019.23xxx
www.ndss-symposium.org

Binary translation presents two major challenges: functional correctness, and performance. Ideally, both are very important, but in practice any implementation of a binary translator needs to make trade-offs between them.

On one hand, *dynamic binary translators* choose to translate code on-the-fly during execution, favoring functional correctness. This choice allows to preserve functional correctness even in presence of self-modifying code, runtime code generation, and unpacking of compressed code. For example, emulators based on dynamic binary translation, such as QEMU, are known to be able to emulate even full operating systems. However, this requires runtime support and it incurs in a significant performance overhead, since, at run-time, the code translation and the execution of its output are interleaved.

On the other hand *static binary translators* make the opposite choice. They make stronger assumptions on their ability to statically identify all the executable code, trading off functional equivalence for speed. In this way, they avoid the overhead of interleaving translation and execution, performing all the translation ahead of time and then executing the translated code at full speed. In addition, static binary translators are usually able to perform more aggressive optimizations for two key reasons: 1) they have global visibility on the code they are translating (unlike dynamic translation), and 2) they pay the cost of optimizations only once, while dynamic translators pay it at each run. This has shown to be beneficial in automated bug detection, especially in performance-critical scenarios such as fuzzing [11].

The main drawback of this approach is that whenever the static analysis is wrong, the translated code loses functional correctness. Given that binary programs are huge blobs of interleaved executable code and data, static binary translators still need to err on the safe side, to avoid missing some pieces of code, and be usable for most use cases.

In this work we focus on a technique for improving runtime performance of statically translated programs. We adopt an approach based on dividing the translated binary in two *realms*: the realm of isolated functions and the realm of non-isolated functions. The *isolated realm* is composed by a set of basic blocks of the original program that have been marked as belonging to the same function by a Function Boundaries Detection Analysis (FBDA). All the detected functions are then cloned and isolated from the rest of the code, so that each of them can be separately optimized during translation, generating high-performance code. The *non-isolated realm*, on the other hand, is a single large function that contains all the code of the original executable.

Due to its complexity, optimizations will be less effective on the *non-isolated realm*. However, thanks to its simplicity and fidelity to the original code, it is considerably safer in terms of correctness. On the other hand, the *isolated realm* can break functional correctness if the execution reaches an unexpected jump inside an isolated function. In such situations, we seamlessly fall back to the non-isolated realm, reusing existing *stack unwinding* mechanisms. In practice, they are used as a safety net in problematic cases to preserve functional correctness, paying the overhead of the unwinding only in exceptional situations, while running at full speed in the isolated functions for most of the time. Our approach is independent of the original CPU architecture of the translated program and agnostic about the ABI. Therefore, functional correctness is not affected by the quality of the algorithm used to detect the function boundaries before isolating them, which is an orthogonal problem [6] [14] [5] [10]. In summary, this paper makes the following contributions:

Design of an isolating static binary translator. We designed a static binary translator that divides the output program in two realms, one composed by high-performance isolated functions, and one composed by a single large and semantic-preserving function, from which it is possible to migrate seamlessly.

Implementation of an isolating static binary translator. We implemented the technique in the `rev.ng` binary translation tool, based on well-known and tested open source frameworks: QEMU [7] and LLVM [13]. `rev.ng` can handle all the CPU architectures supported by QEMU (22+) and translate programs towards all the LLVM targets that support stack unwinding via `libgcc` [2].

Measurement of the performance improvement. We used `rev.ng`'s algorithm [10] to detect function boundaries, and we measured the performance improvement on the SPECint 2006 benchmarks [12], showing an average improvement in performance of 42%, while passing the tests for functional correctness.

II. Background

This section briefly introduces the key concepts necessary to understand the rest of the work. Section II-A introduces the most important features the `rev.ng` binary translation tool. Then, Section II-B describes the features of the generated code that prevent optimizations. Finally, Section II-C provides an overview of a Function Boundaries Detection Analysis (FBDA), necessary to perform the function isolation, and the reasons why we cannot assume 100% correct results from it.

A. `rev.ng`*: a Quick Overview*

`rev.ng` is an open-source [4] binary analysis framework [10] that can also work as a static binary translator, i.e., given a Linux program compiled for an architecture A (e.g., ARM), `rev.ng` can produce an equivalent program for architecture B (e.g., x86-64), or even for A itself.

To achieve this result, `rev.ng` employs two key components: QEMU [7] and LLVM [13]. QEMU is employed as a library to lift each basic block from the input program into an architecture-independent representation, known as the QEMU *tiny code*. This representation was designed to perform lightweight analyses and transformations and, therefore, it is not very suitable to perform sophisticated analyses. For this reason, `rev.ng` translates the QEMU *tiny code* into LLVM IR. LLVM is a robust and mature compilation framework, providing a wide range of analyses and transformations, and a solid infrastructure to easily develop new ones.

While discussing the full design of `rev.ng` [9], [10] is outside the scope of this work, in the following we highlight the key points of interest for this paper.

The CPU State Variables (CSV). In `rev.ng`, each part of the CPU state, such as registers, are represented as global variables, known as the CPU State Variables (CSV). For instance, in x86-64, there is a 64-bit integer global variable for each general purpose register. The translated code reads and writes them with load and store operations.

The `root` function. The actual translated code. Each input basic block can produce multiple LLVM basic blocks. All these basic blocks are collected in a single LLVM function known as `root`. Therefore, the whole control-flow of the original program takes place inside this function. As an example, if in the original program there is a direct jump from address A to address B, the generated program will simply jump from one of the basic blocks corresponding to A to the first basic block corresponding to B.

The `dispatcher`. In case of an indirect jump for which it is not possible to statically enumerate all the possible destinations, the execution is diverted to the `dispatcher`. The `dispatcher` is a piece of code in the `root` function that decides, depending on the run-time value of the program counter, the actual target of the unresolved indirect jump in the translated program. It is implemented through a `switch` statement on the program counter, where each `case` represents the address of a basic block in the original program and its body is a jump to the corresponding LLVM basic block in the translated program.

As an example, see Fig. 1, where a simple x86-64 assembly program is presented (Fig. 1c) along with the (simplified) LLVM module produced by `rev.ng` (Fig. 1a).

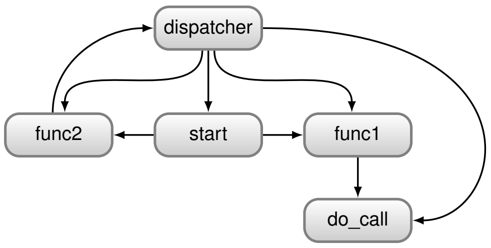
The last thing that is vital to understand for this work is the fact that in the translated program there are two distinct stacks being used. The first is the regular stack employed by the code generated by LLVM to, e.g., spill registers and record return addresses. The second is the *emulated* one, i.e., a portion of memory that is employed as a stack and that evolves exactly how it would have evolved in the original program. This distinction is very important, because it implies that the former stack does not contain vital information and that, therefore, unwinding through it and discarding its content does not corrupt the semantic of the original program.

B. *Limitations of the current approach*

The approach described above is very effective in preserving the behavior of the program, but it also has a major downside: it heavily inhibits the optimizer. Consider the x86-64 assembly

```
@pc = global i64 0
@rax = global i64 0
@rbx = global i64 0

define void @root() {
dispatcher:
  %1 = load i64, i64* @pc
  switch i64 %1 [
    i64 0x1000, label %bb.start
    i64 0x1010, label %bb.start_return_1
    i64 0x1020, label %bb.start_return_2
    i64 0x2000, label %bb.func1
    i64 0x2010, label %bb.do_call
    i64 0x2020, label %bb.do_call_return
    i64 0x3000, label %bb.func2
  ]

bb.start:
  store i64 %bb.do_call, i64 *@rbx
  ; Push 0x1010 on the stack
  br label %bb.func1

bb.start_return_1:
  ; Push 0x1020 on the stack
  br label %bb.func2

bb.start_return_2:
  %top_of_stack = ...
  store i64 %top_of_stack, i64 *@pc
  br label %dispatcher

bb.func1:
  store i64 @system, i64* @rax
  br label %bb.do_call

bb.do_call:
  %target = load i64, i64* @rax
  store i64 %target, i64 *@pc
  ; Push 0x2020 on the stack
  br label %dispatcher

bb.do_call_return:
  %top_of_stack = ...
  store i64 %top_of_stack, i64 *@pc
  br label %dispatcher

bb.func2:
  store i64 @exit, i64* @rax
  %target = load i64, i64* @rbx
  store i64 @target, i64* @pc
  br label %dispatcher

}
```

(a) The root function before isolation.



(b) The control-flow graph of the root function before isolation.

```
start:               func1:               func2:
 mov rbx, do_call     mov rax, system      mov rax, exit
 call func1          do_call:              jmp rbx
 call func2           call rax
 ret                  ret
```

(c) The assembly of the example program.

```
define void @bb.start() {
  store i64 %bb.do_call, i64 *@rbx
  ; Push 0x1010 on the stack
  call void bb.func1()
  ; Push 0x1020 on the stack
  call void bb.func2()
  ret void
}

define void @bb.func1() {
  store i64 @system, i64* @rax
  %target = load i64, i64* @rax
  store i64 %target, i64 *@pc
  ; Push 0x2020 on the stack
  call void @function_dispatcher()
  ret void
}

define void @bb.func2() {
  store i64 @exit, i64* @rax
  %target = load i64, i64* @rbx
  store i64 @target, i64* @pc
  call void @function_dispatcher()
  ret
}

define void @function_dispatcher() {
  %1 = load i64, i64* @pc
  switch i64 %1, label %missing [
    i64 0x1000, label %bb.start
    i64 0x2000, label %bb.func1
    i64 0x3000, label %bb.func2
  ]
bb.start:
  call void @bb.start()
  ret

missing:
  call void _Unwind_RaiseException()
  unreachable

; ...
}

define void @root() {
; dispatcher, bb.start_return_1, bb.start_return_2
; bb.do_call and bb.do_call_return are unchanged
bb.start:
  invoke void @bb.start() to %return unwind %catch
bb.func1:
  invoke void @bb.func1() to %return unwind %catch
bb.func2:
  invoke void @bb.func2() to %return unwind %catch
return:
  br label %dispatcher
catch:
  br label %dispatcher
}
```

(d) The translated module after function isolation, containing both the non-isolated realm (the root function) and the isolated realm (the bb.start, bb.func1 and bb.func2 functions).

Fig. 1: An example of a program and the different stages of its translation.

snippet in Fig. 1c, and `func1` in particular. A superficial analysis of the code might lead to conclude that the value read from `rax` in the `do_call` basic block is always `system`, and, therefore, an obvious optimization would be to replace `call rax` with `call system`. However, the LLVM optimizer is not allowed to this. In fact the `func2` function contains an indirect jump that might (and actually does) target the `do_call` basic block. As shown in Fig. 1b, in the code generated by `rev.ng` this is reflected by the fact that on the control-flow graph the node representing `do_call` has two predecessors: the obvious one (`func1`) and the `dispatcher`.

C. Function Boundaries Detection Analysis

The above mentioned problem could be solved by producing a version of the code above where each *function* of the original program is isolated in its own LLVM function. This function would include only the strictly necessary set of basic blocks, e.g., for the function `func1` the `func1` and `do_call` blocks only. Without the `dispatcher`, the LLVM optimizer would deal with a clean control-flow graph which would enable, among others, the above mentioned optimization.

However, while `rev.ng` provides a quite accurate Function Boundaries Detection Analysis (FBDA) [10], it is not perfect. More in general, an analysis aiming at detecting the function boundaries is only as good as the underlying analysis to recover the control-flow graph. Therefore if, for instance, the target of an indirect jump has not been statically identified, without the `dispatcher` the execution will fail at run-time. Given that statically enumerating a complete and correct set of jump targets is, in general, impossible (think about extreme situations where programs jump to user-provided values), it is also impossible to design a perfect FBDA.

Moreover, even in presence of an (hypothetical) perfect analysis for the detection of function boundaries, in certain situations the program execution might leave the body of the function in unexpected ways. Two typical examples are `longjmp` functions or C++ exceptions. As a consequence, a backup solution needs to be put in place to preserve the semantic of the program in the isolated function

III. Proposed Solution

This section presents the core contributions of this work. Specifically, Section III-A introduces the two realms in which we divide the translated program, Section III-B proceeds by describing how the functions of the isolated realm are obtained, and Section III-C concludes explaining the transition mechanism between the two realms.

A. The Two Realms

To circumvent the problem described in the previous section, we design a *fail-safe* approach to execute isolated functions, even in presence of errors in the FBDA.

The key idea behind our solution is to divide the generated program into two *realms*: the *isolated* and *non-isolated* realm.

Isolated realm. This realm is composed by the functions reconstructed on the basis of the information provided by the FBDA. Functions in this realm, have a clean, although possibly incomplete, control-flow graph. Optimizations are more effective on the smaller scopes, compared to the same code executed in the `root` function. As a consequence, it is desirable that most of the runtime is spent in this realm.

Non-isolated realm. This realm is composed by a modified `root` function. The core idea is to employ this realm as a safer, but slower, backup solution in case execution in an isolated function performs a jump to an unexpected location, i.e., a basic block not classified as part of that function.

Consider the example in Fig. 1c and assume that the FBDA detected three functions, `start` (the `start` basic block), `func1` (`func1` and `do_call`) and `func2` (`func2` only). The resulting three isolated functions are shown at the beginning of Fig. 1d.

B. The Function Isolation Process

The Function Boundaries Detection Analysis annotates each basic block in the `root` function (using LLVM metadata) with the list of functions it belongs to. We exploit this information to perform the isolation process as follows.

First of all, we scan all the basic blocks in the `root` function, and if we encounter a basic block that has been marked as a *function entry* (i.e., the entry point of a function), a new, empty, LLVM function is created.

Subsequently, we scan the `root` function again, cloning each basic block inside every function it belongs to. Notice that a single basic block can be classified as part of multiple functions by the FBDA. In such case the basic block is simply cloned multiple times, one for each function that requires it. The content of each basic block is mostly unchanged, with two main exceptions: 1) return instructions, which in the `root` function appeared as jumps to the `dispatcher`, are replaced with actual LLVM `ret` instructions; 2) function calls, which in the `root` function appeared as simple branch instructions, are replaced with actual LLVM `call` instructions to the corresponding isolated function. Again, for an example see the first three functions of Fig. 1d and compare them with corresponding basic blocks in Fig. 1a.

After this transformation, the only instruction that can compromise the semantics in the isolated realm are indirect jumps, since they can target to any location in the program, even to addresses that are not part of the current isolated function or that are not even function entries. To handle this scenario, it is necessary to build a dedicated mechanism: the `function_dispatcher`. The `function_dispatcher` (Fig. 1d) is similar to the `dispatcher` of the `root` function, with some restrictions. The `function_dispatcher` does not directly handle all the addresses of basic blocks of the program, but only those representing the entry point of a function.

The last step to complete the isolated realm is to analyze all the isolated functions, and to substitute all the unresolved indirect jumps with direct calls to the `function_dispatcher`. The `function_dispatcher` takes a single argument, the value of target address of the indirect jump, that is only known at runtime. Then, if this address is also a valid entry point for an isolated function, it directly calls it. Otherwise, whenever the target address cannot be resolved to an isolated function, it is necessary to migrate the execution of the program back to the non-isolated realm. In order to do that, the `function_dispatcher` adopts a strategy based on stack unwinding, that we illustrate in the next section.

C. From `root` to the Isolated Realm and Back

Now that we have seen how the isolated realm is built, we can describe the mechanism to migrate execution from the non-isolated realm to the isolated realm and back.

The key idea is to reuse the ELF exception handling mechanism, the same employed to implement the `backtrace` function of the GNU C Standard Library and the so-called C++ *zero-cost* exceptions. The advantage of this approach consists in the fact the we can reuse *as is* features of LLVM and the stack unwinding mechanism embedded in `libgcc` to achieve a multi-architecture migration mechanism with a very minor effort.

Non-Isolated → Isolated. In the `root` function, each basic block representing a *function entry* is transformed in a *trampoline* performing a call to the corresponding isolated function. The trampoline is composed by an `invoke` instruction, that behaves like a regular `call` instruction, except for the fact that it informs the compiler of the fact that the called function might throw an exception (in a C++ sense). Therefore, it features two possible successors: a normal fallthrough basic block to use in case the function returns in a regular way, and a `catch` block in case an exception is raised. Due to the presence of an invoke instruction, LLVM will emit all the necessary data structures to perform the stack unwinding process at run-time. These data structures end up in the `.eh_frame` section [1], and are very similar to the DWARF debug information [8] that describe the usage of the stack in a function.

Isolated → Non-Isolated. To migrate back to the non-isolated realm, an exception is thrown using the `_Unwind_RaiseException` function provided by `libgcc`. The exception causes the stack to be unwound up to the last `invoke` performed in the `root` function. Then, the execution proceeds from the `catch` successor of the `invoke` instruction, that will handle the abnormal situation by invoking the `dispatcher` of the `root` function from where the execution can resume as normal, within the non-isolated realm. Subsequently, as soon as the execution hits a basic block that has been identified as a function entry, the execution will go back to the isolated realm.

An example of the `invoke` instruction is available in Fig. 1d in the `root` function, while the `default` case of the `switch` in the `function_dispatcher` shows how an exception is thrown.

In the following, we summarize the different ways in which execution can leave a function of the isolated realm:

Return. When a function that has been marked as a return instruction by the FBDA pass, a regular `ret` is emitted, and the control passes back to the caller.

Direct function call. When an instruction that has been identified as a regular call targeting an address identified as a function, a `call` instruction to the corresponding isolated function is emitted.

Indirect function call. As above, but the target of the call is not statically known and, therefore, it is not possible to know if the target address will match an isolated function. As a consequence, a jump to the `function_dispatcher` is emitted, which might throw an exception.

Bad return address. After a `call` instruction, `rev.ng` emits a check to ensure that the target of the indirect branch (that has been identified as a return) actually matched the fallthrough address of the function call. This is basically a safety check to detect the situation in which an indirect jump has been mistakenly identified as a return instruction. If the check does not pass, an exception is thrown and the execution is passed back to the non-isolated realm.

Jump. In case of a jump (not a function call) to an address that has not been identified as part of the current function we have three options: 1) it is a direct jump to the address of a function, 2) it is a direct jump to an address that is not a function, 3) it is an indirect jump. The first situation represents a tail call, therefore we emit a plain function call to the corresponding function. The second situation might be a `longjmp` or the internals of the stack unwinding process of, e.g., a program throwing a C++ exception, therefore, we throw an exception and move back to the non-isolated realm. The third situation might be one of the two previous situations: a call to the `function_dispatcher` is emitted.

An interesting fact to note is that, except for errors in the function boundaries detection analysis, we migrate from the isolated realm to the non-isolated realm only in case of exception-related code in the original program. In practice, we handle exception using exception handling mechanisms.

D. An Example Execution Trace

In this section we briefly illustrate how the execution of the example program evolves in the translated program after the function isolation process has been performed.

The execution begins at the `start` function, which initializes `rbx` to the address of the `do_call` basic block and then calls `func1` and `func2`. `func1` sets `rax` to the address of the `system` library function and then (in the `do_call` basic block) calls it. `func2`, in turn, sets the value of `rax` to the address of the `exit` library function and jumps to `rbx`, which contained the address of the `do_call` basic block. As a consequence, the `exit` function is invoked and the program terminates.

```
@root:%dispatcher       @function_dispatcher
@root:%bb.start         @function_dispatcher:%missing
@bb.start               * Exception *
@bb.func1               @root:%bb.start
@function_dispatcher    @root:%catch
@system                 @root:%dispatcher
@bb.func1               @root:%bb.do_call
@bb.start               @root:%dispatcher
@bb.func2               @exit
```

Fig. 2: Execution trace corresponding to the translated program in Fig. 1d. Reading order from top to bottom, columnwise.

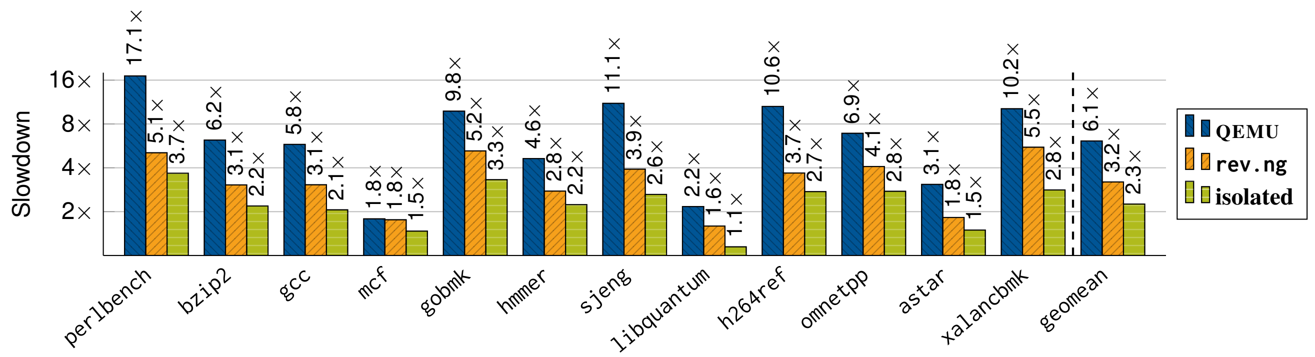


Fig. 3: Slowdown of the different translation techniques compared to native code. Logarithmic scale. Lower is better.

The execution trace in Fig. 2 corresponds to the translated program in Fig. 1d. Assuming that the `dispatcher` basic block in `root` and the `function_dispatcher` can handle calls to library (non-translated) functions, the execution evolves pretty linearly. A notable fact is how execution starts from the non-isolated realm (from the `dispatcher`) but we immediately switch to the isolated realm by jumping to the `@bb.start` function. The other relevant event is the execution of the *unexpected* jump (i.e., `jmp rbx`): an exception is thrown, the stack is unwound and execution resumes from the last `invoke` performed in the `root` function and, from there, execution is resumed in the non-isolated realm.

IV. Experimental Results

To evaluate the performance improvements, we implemented both the function isolation and the unwind-based fail-safe mechanism for functional correctness in the latest version of `rev.ng`, which is based on QEMU 2.5.0 and LLVM 7.0. For stack unwinding, it relies on the unwinding library included in `libgcc` [2], but nothing in the implementation strictly relies on it, given that the library is Application Binary Interface is mandated by the Itanium C++ ABI specification on Exception Handling [3] and it is followed by all major compilers. We also used ELF information about sections to distinguish data from code, which is an orthogonal problem, in order to avoid erroneously translating data as code.

For the evaluation, we used the SPECint 2006 benchmark suite [12]. We built all the benchmarks for Linux on x86-64 with `gcc` 4.9.3. Then we ran the benchmarks and measured performance in the following configurations:

Native. Native execution without translation.
QEMU. Emulation with dynamic binary translation.
`rev.ng`. Static binary translation with `rev.ng`.
`rev.ng` + isolation. As before, with function isolation.

All the results reported in the following come from reproducible runs (three runs of the benchmark suite) of the SPECint benchmarks for each of the experimental setups mentioned above, measured on a GNU/Linux machine with an Intel Xeon L5420 CPU running at 2.50 GHz, with 6MB of L2 cache and 24GB of RAM.

Results in Fig. 3 show that function isolation introduces a sensible speedup for code translated by `rev.ng`, which now presents an average slowdown of 2.25× with respect to native code, while the code translated without function isolation presents an average slowdown of 3.20×. In other words, function isolation provides an average speedup of 1.42× for the translated code. This enables code translated with `rev.ng` to improve the performance with respect to emulation with QEMU, which presents an average slowdown of 6.11× with respect to native code. This speedup is provided by the optimizations that LLVM can apply thanks to the simpler and clearer control-flow in isolated functions. These results are due to the quality of our FBDA, since the number of exceptions generated during the execution impacts the overall performance of the translated code. However, even in case of a FBDA of poor quality our design preserves the correctness of execution semantics. In our case, the FBDA implemented in `rev.ng` has proved to be quite accurate. In fact, out of 12 benchmarks, the only ones that caused exceptions during the execution were `perlbench` (8444 exceptions), `gcc` (329261 exceptions), `omnetpp` (2 exceptions) and `xalancbmk` (26905 exceptions). In several cases these exceptions were caused by the complementary functions `setjmp` and `longjmp`. Each exception provides also useful information to understand mistakes performed during FBDA. This information can be used to improve, when possible, the FBDA accuracy.

V. Conclusion

In this work we presented a novel approach for applying function isolation to a binary translated with `rev.ng`, while at the same time preserving the correctness of the execution semantics regardless of the quality of the FBDA used. We measured the performance of the code translated with this new technique, measuring significant improvements. In the future, we plan to further improve the performance by improving the quality of the FBDA, thus reducing the number of exception thrown, and by promoting the global variables representing the CSV to local variables, an operation that will allow for more optimizations and therefore, fewer memory accesses, to be performed.

References

[1] Exception Frames. https://refspecs.linuxfoundation.org/LSB_3.0.0/LSB-PDA/LSB-PDA/ehframechpt.html.
[2] Interface definitions for libgcc_s. https://refspecs.linuxfoundation.org/LSB_4.0.0/LSB-Core-S390/LSB-Core-S390/libgcc-sman.html.

[3] Itanium C++ ABI: Exception Handling. `https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html#base-abi`.

[4] rev.ng. `https://rev.ng/`.

[5] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 177–189, April 2017.

[6] Tiffany Bao, Johnathon Burket, Maverick Woo, Rafael Turner, and David Brumley. Byteweight: Learning to recognize functions in binary code. USENIX, 2014.

[7] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pages 41–46. USENIX, 2005.

[8] DWARF Standards Committee. The DWARF Debugging Standard. `http://dwarfstd.org/`.

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

[10] 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*, pages 131–141, 2017.

[11] Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. `rev.ng`: A multi-architecture framework for reverse engineering and vulnerability discovery. In *International Carnahan Conference on Security Technology, ICCST 2018, Montréal, Canada, October 22-25, 2018*. IEEE, 2018.

[12] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[13] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO 2004*.

[14] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *USENIX Security Symposium*, pages 611–626, 2015.