REV.NG: A Unified Binary Analysis Framework to Recover CFGs and Function Boundaries

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

Mathias Payer
Purdue University, U.S.A.
mathias.payer@nebelwelt.net

Giovanni Agosta
Politecnico di Milano, Italy
agosta@acm.org

Abstract

Static binary analysis is a key tool to assess the security of third-party binaries and legacy programs. Most forms of binary analysis rely on the availability of two key pieces of information: the program's control-flow graph and function boundaries. However, current tools struggle to provide accurate and precise results, in particular when dealing with hand-written assembly functions and non-trivial control-flow transfer instructions, such as tail calls. In addition, most of the existing solutions are ad-hoc, rely on hand-coded heuristics, and are tied to a specific architecture.

In this paper we highlight the challenges faced by an architecture agnostic static binary analysis framework to provide accurate information about a program's CFG and function boundaries without employing debugging information or symbols. We propose a set of analyses to address predicate instructions, `noreturn` functions, tail calls, and context-dependent CFG.

REV.NG, our binary analysis framework based on QEMU and LLVM, handles all the 17 architectures supported by QEMU and produces a compilable LLVM IR. We implement our described analyses on top of LLVM IR. In an extensive evaluation, we test our tool on binaries compiled for MIPS, ARM, and x86-64 using GCC and clang and compare them to the industry's state of the art tool, IDA Pro, and two well-known academic tools, BAP/ByteWeight and angr. In all cases, the quality of the CFG and function boundaries produced by REV.NG is comparable to or improves over the alternatives.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

Keywords static binary analysis, function boundary detection, control-flow graph recovery

1. Introduction

Binary analysis is an effective approach to recover useful information from legacy and commercial off-the-shelf (COTS) binaries for which the source code is not available. Such information helps the analyst in understanding the behavior of the program (e.g., to classify benign or malicious functionalities), to evaluate and test if specific security policies hold, or even to retrofit security features such as CFI [23, 19, 10, 22] directly into the binary.

In the context of static binary analysis, when dealing with C/C++ programs stripped of debugging information, two particularly challenging tasks are the recovery of an accurate *control-flow graph* (CFG) for the program and the detection of the boundaries of the original functions. This information is very supports reverse engineering purposes or debugging code and it is essential to efficiently rewrite or transform the underlying binary.

While a number of existing tools already provide such information, they often struggle to support multiple architectures. Existing tools either develop a set of ad-hoc heuristics to handle each new architecture (IDA Pro [8]) or, if the tool employs an intermediate representation for its analyses, have to create a new frontend to handle each new architecture. Examples of the latter case are BAP/ByteWeight [3, 2] (x86 and ARM only), MC-Semantics [18] (x86 only) and LLBT [14] (ARM only). A key consequence is that existing tools focus only on one or few architectures, which means that software built for architectures that are less popular, or emerging (e.g., RISC-V) is difficult to analyze. This problem intensifies with the spread of IoT devices which often employ low power or ultra-low power processors with less known and highly customized ISAs.

We propose a unified system for binary analysis which employs a set of principled techniques rather than architecture-specific heuristics for its analyses and that, unlike existing work, defers the burden of providing a reliable frontend for a wide range of architectures to existing tools. In our prototype we rely on QEMU to provide such a frontend. QEMU is a dynamic binary translator that lifts binary code into a custom intermediate representation (IR) for 17 different architectures, including x86, x86-64, MIPS, ARM, and AArch64. As a tool aiming at full system emulation, QEMU supports even the most sophisticated ISA extensions. For instance, it already supports the recently introduced Intel MPX ISA extensions. The large community and industry interest around QEMU virtually guarantees that new architectures and extensions are supported promptly (e.g., RISC-V [12]).

The main benefit of this approach, besides the vastly reduced effort in handling a large number of ISAs, relies on the fact that all the supported architectures are first-class citizens. Moreover, QEMU's IR abstracts all architecture-specific details. All analyses building on top of the QEMU IR, therefore, have to be designed in an ISA-agnostic way. In practice, when handling ISA-specific peculiarities, such as MIPS delay slots or ARM predicate instructions, the approaches are equally effective, respectively, on OpenRISC delay slots and x86-64 conditional move instructions.

On top of this framework, we build a set of sophisticated, ISA-agnostic analyses to recover two critical pieces of information for further analysis: the CFG and function boundaries. Unlike existing tools that strive to achieve similar aims, we increase the accuracy

of the recovered CFG by also handling hand-written, low-level assembly functions with non-trivial CFGs. Recovered CFGs and function boundaries are the foundation for static binary analysis and static binary rewriting mechanisms. Precision is crucial as imprecision leads to overhead, loss of functionality, and potentially to security vulnerabilities when the transformed binary is executed.

In our prototype implementation, REV.NG, we translate the IR provided by QEMU into LLVM IR, an environment that facilitates further analyses. In our evaluation we compare our prototype against the de-facto commercial standard, IDA Pro [8], and two open-source tools, BAP/ByteWeight [3, 2] and angr [17, 16], showing how we obtain comparable or better results preserving the generality of our approach by avoiding any ISA-specific heuristic.

Contributions. In this paper, we make the following contributions:

- We design a framework for static binary analysis that handles different architectures.
- We propose a set of fully automated principled analyses, built on top of our framework to recover an accurate CFG and function boundaries, even in presence of hand-written assembly and neither employing ISA-specific (hand-crafted) heuristics nor relying on debugging information or symbols.
- We evaluate our prototype implementation against the de-facto industry standard for static binary analysis, IDA Pro, and two related works [3, 17, 16, 2], and we show that our results are comparable or better without having to fall back to per-ISA hand-crafted heuristics. We also demonstrate that we handle the CFG of non-trivial, real world hand-written assembly functions where other tools fail.

Organization of the paper. Section 2 presents the main challenges to address in order to obtain an accurate CFG and accurate function boundaries. In Section 3 we introduce our analysis framework and the binary lifting process. Section 4 discusses our solution, while in Section 5 we report the results of our experiments and provide a case study. Finally, Section 6 reviews the state of the art and, in Section 7, we draw our conclusions.

2. Challenges for binary analysis

Recovering the *control-flow graph* and the function boundaries of a binary program for which no source code or debugging information are available presents several challenges [20]. In the following, we discuss them focusing on the issues that an ISA-independent analysis framework faces.

2.1 Challenges in CFG recovery

Recovering the CFG of a program consists in essentially two phases. The first phase identifies the basic blocks composing the application, while the second phase establishes the correct relationships among them in terms of control-flow. In practice, this means that the analysis has to enumerate all basic block starting addresses, their size, and whether a control-flow transfer from one basic block to another is feasible or not.

Basic block identification. The initial step of any analysis on a binary program consists in identifying the subset of the program image that contains executable code. While distinguishing code from data is an orthogonal problem and outside the scope of this work, typically the various image formats provide a segmentation of the program in terms of executable, readable, and writable areas, even in absence of debugging information. Indeed, this information is essential for the operating system to assign appropriate permissions to the underlying memory pages when an instance of a program is initialized.

After the identification of the executable portion of the program, basic blocks need to be identified. The starting addresses of basic blocks can be drawn from multiple sources. The first and foremost source consists in the program's entry point and, in case of a dynamic library, in the exported functions. Other basic block starting addresses can be harvested through code analysis. Pointers to other basic blocks are found in direct jump instructions, constants materialized through one or more instructions, and global data (`.rodata` in particular) where jump tables or function pointers are typically stored.

CFG recovery. After collecting basic blocks' starting address and size, the control-flow recovery can start. Direct jump instructions provide useful information and are straightforward to incorporate. Unfortunately, they are not sufficient to completely recover the CFG. The main challenge consists in handling indirect control-flow transfer instructions, i.e., indirect function calls and indirect branch instructions. In general, it is impossible to enumerate the exact set of possible *jump targets* for indirect control transfers. The worst case is represented by a jump to a user-controlled value, where all executable code becomes reachable. Alternatively, the destination address may be the result of an arbitrarily complicated computation, which might even be impossible to track.

With this premise, we classify *indirect* control-flow transfer instructions in three categories that, when handled correctly, provide an accurate CFG:

Compiler-generated, function-local CFG. All the indirect jump instructions generated by a compiler to efficiently lower the control-flow of certain statements, most notably C `switch` statements.

"Reasonable" hand-written assembly. Indirect jumps manually introduced by the developer in assembly, usually to optimize low level routines such as `memcpy`. Compared to the CFG that a compiler typically generates, the developer can produce more efficient but hard-to-analyze code. By *reasonable*, we mean, for instance, a function that does not make assumptions about the values of its parameters, but rather enforces (implicitly or explicitly) the constraints locally (see example in Section 5.2).

Indirect function calls. Indirect function calls through function pointers or C++ virtual functions.

In this work, we aim to handle accurately the first case and to develop a set of analyses to handle as many cases as possible of the second category. In fact, these two classes make up the most part of the CFG needed for our final goal: the recovery of function boundaries. On the other hand, indirect function calls might involve virtual tables or function pointer fields of dynamically allocated objects, which are harder to track statically. In short, recovering targets of function calls is an orthogonal problem, which deserves to be treated on its own and does not compromise functionality, since pointer to targets of indirect function calls are typically available elsewhere (e.g., virtual tables). Note that, to enable support for large binaries, as a design choice, we do not employ SMT-solvers, which can provide the most accurate results but limit the scalability of the approach [17].

2.2 Challenges in the recovery of function boundaries

The recovery of function boundaries in a program consists in identifying all function entry points and associating them all with their reachable basic blocks, skipping over function call instructions. This task poses a series of challenges.

First of all, the accuracy of the function boundary recovery is highly dependent on the quality of the underlying CFG. In fact, if, e.g., the CFG lacks information about the destinations of an indirect

jump, all the destination basic blocks might not be considered part of the function, leading to a loss in accuracy.

Another issue is deciding whether a certain basic block is the entry point of a function or not. The presence of an explicit function call to its address is a strong indication, but it may not always be available. Specifically, a certain function might never be called directly but only through a function pointer, a C++ virtual call, or a tail call.

Further, several common challenges emerge while trying to identify function boundaries across architectures. In the following we report some of the most relevant:

Call thunks. In ISAs where the program counter is not addressable, it is a common practice to perform a function call to the next instruction so that the program counter becomes available on the stack or in the link register. The destination of such a function call should not be mistakenly interpreted as the entry point of an actual function.

`noreturn` functions. A `noreturn` function, in C terms, is a function that never returns (e.g., `exit` or `longjmp`). These functions are sometimes called through a function call instruction and not through a simple jump. This leads to a spurious path from the call site to the next instruction, which might even be part of a distinct function.

Shared code. Two functions may share a portion of their bodies, in particular, two hand-written assembly functions might share the footer or a sequence of instructions for error handling.

Calls to the middle of a function. In certain cases, a function might have multiple entry points. This case is mostly seen in hand-written assembly and it is usually employed to provide a faster version of a function that does not verify certain preconditions that are known to hold.

Tail calls. Tail calls appear in the code as simple unconditional jump instructions, and, therefore, have to be handled in a way that prevents them from being mistakenly identified as part of the function-local CFG.

3. First steps: binary lifting and a basic CFG

This section discusses preliminary steps on how a high-level program representation and a basic control-flow graph is obtained from binary code. Leveraging the ISA-independent QEMU binary translator, binary code is lifted into QEMU IR which can then be translated into LLVM IR. Starting from this LLVM IR, a set of analyses (OSRA and SET) [6] recover a basic control-flow graph.

3.1 An ISA-independent binary analysis framework

A primary objective of our work consists in developing ISA-independent analyses. While this is challenging *in itself*, supporting different instruction sets in a unified manner increases these challenges. The ideal situation would be to work on an *intermediate representation* while abstracting all the details specific to an architecture and making the behavior of each instruction explicit, along with all its side effects. While this has been done in the past [3, 14], from an engineering point of view, it requires a large amount of work, in particular for large and complex CISC instruction sets such as x86 and its successors. Moreover, such an effort has a non-diminishing marginal cost for supporting new architectures, since, in most cases, the opportunities for code reuse in different architectures are limited. Therefore, related works typically focus on a limited set of architectures (usually one or two) and often only support a subset of instructions in complex ISAs like x86, thus ignoring, e.g., vector instructions or floating point instructions.

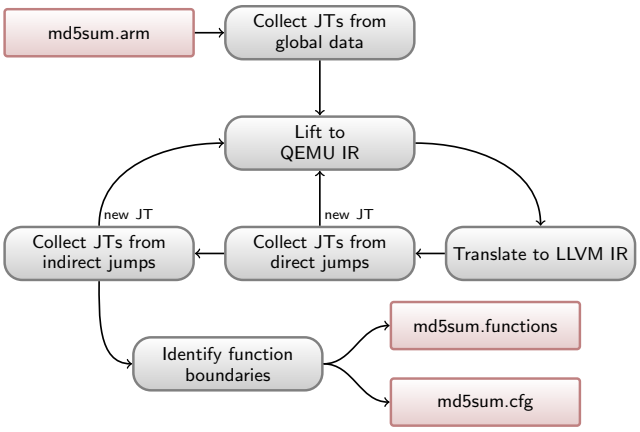

Figure 1. Overview of the REV.NG system. JT stands for *jump target*, and the *new JT* notation represents the fact that at least a new jump target has been discovered.

REV.NG decouples the problem of CFG recovery from interpreting an ISA by offloading the task of handling different architectures to an existing abstraction layer: QEMU. The core component of QEMU enabling this kind of abstraction is the *tiny code generator* (TCG) which translates instructions of a supported ISA into TCG instructions, QEMU's IR. In emulator mode, QEMU translates its IR to executable code for the host architecture. In our case, instead of generating machine code, we further translate the QEMU IR into a higher level IR, namely the LLVM IR. By employing QEMU's *tiny code generator* as a frontend, we obtain an IR from any of the architectures it supports with minimal effort.

While our analyses are agnostic with respect to the underlying IR, leveraging LLVM has several advantages. For example, the LLVM IR is in SSA-form and provides use-def chains out of the box. LLVM has a well-developed and clean API. As a compiler framework, LLVM allows the recompilation of generated code for any of the supported target architectures. Employing LLVM in a binary analysis framework allows building a static binary translator with minimal effort [6].

3.2 From binary code to LLVM IR

As shown in Figure 1, the translation process begins by parsing the binary image and loading its segments into memory. Then, the program's global data is scanned in search of pointer-sized values pointing to an address in the executable segment. Each target address (or *jump target*), is passed to QEMU, which translates the basic block at the corresponding address into TCG instructions. The sequence of TCG instructions are then expanded to equivalent LLVM IR and collected into a set of basic blocks. These basic blocks, in turn, are collected into a function, known as *root*.

Destination addresses of direct jumps observed during translation are registered as *jump targets* for further exploration. The process proceeds iteratively until there are no known untranslated addresses. Then, as we will describe in more detail in the next section, targets of indirect control flow transfers are recovered, and, if necessary, fed back to QEMU. Once all the possible *jump targets* have been recovered, the CFG is analyzed for the recovery of function boundaries, leading to the desired output.

Note that the translation from TCG instructions to LLVM IR is straightforward. In particular, each TCG register (i.e., each part of the CPU state) is mapped onto a local variable in the *root* function, which we call *CPU State Variable* (or CSV). In case of recompilation, the register allocator takes care of lowering such variables in the most efficient way.

3.3 Basic control-flow graph recovery

SET and OSRA [6] are two key analyses to obtain the *jump targets* of indirect jumps. These analyses can be employed as a first step towards the recovery of an accurate CFG, enabling further analyses, such as the detection of function boundaries.

In the following, we present them, assuming that the directly reachable portion of the CFG (i.e., the part obtained from *direct* control-flow transfers) is available. Without loss of generality we describe the analyses in the context of the LLVM environment compiler framework.

Simple Expression Tracker. The Simple Expression Tracker (or SET) is an analysis that collects literal values observed in the code. A typical situation where SET is effective are direct jumps whose destination is too far from the current PC to be encoded directly as an in-instruction immediate, which in architectures such as MIPS and ARM has a limited size. Therefore, these types of jumps are often lowered to a jump through a register where the destination address is *materialized*. For instance the following ARM code uses two instructions to materialize the address to load (`0x21024`):

```
movw    r3, #4132 ; 0x1024
movt    r3, #2
ldr     r3, [r3]
```

The SET looks for instructions performing a write to a CSV (e.g., a register) and tracks how the value is computed. To do so, the definition of the stored SSA value is inspected: if it is an operation employing at most a single non-constant operand, we register it on the stack and proceed to inspect the non-constant operand. If an operation with no non-constant operands (i.e., a constant) is met, the stack in its current status is traversed from top to bottom applying all the recorded operations one-by-one. The final result is the value that would be written in the CSV.

Note that the operations supported by SET include memory loads. This means that if, while traversing the stack to materialize a value, a memory load is found, and the value materialized up to that point is a pointer to global data, SET will read the corresponding value from the program image and proceed.

OSRA. OSRA is a data-flow analysis that maps each SSA value to an *offset shifted range* (OSR, hence the name, *OSR Analysis*). An OSR represents the fact that the considered SSA value can be expressed in terms of another SSA value x, multiplied by a factor b and with an offset a added. Moreover, it associates x in a specific basic block with a single, possibly negated, range $[c, d]$ and a *signedness*. We define x along with bound constraints and signedness information a *bounded value*. Symbolically, an OSR can be summarized as follows:

$$a + b \cdot x, \text{ with } \left\{ x : \begin{array}{l} x \in [c,d] \\ x \notin [c,d] \end{array} \text{ and } x \text{ is } \begin{array}{c} \text{signed} \\ \text{unsigned} \end{array} \right\}$$

The offset and the multiplying factor come into effect, respectively, through add/subtract operations and multiply/divide/shift operations. Information about the bounds of a *bounded value* are enforced on targets of conditional jump instructions comparing an SSA value associated with an OSR and a constant. Finally, the signedness of the *bounded value* is inferred by inspecting the type of instructions in which it is employed (e.g., signed division versus unsigned comparison).

The results provided by OSRA are then employed by SET. Suppose that SET, during its exploration, finds an instruction for which OSRA can provide useful information, i.e., the OSR's *bounded value* is within a reasonably small range. In this case, SET will *materialize* each value the OSR can assume considering the bounds of its *bounded value* and run it through the stack as explained before.

As an example, consider the following LLVM IR reporting the tracked OSR in comments, and suppose we want to enumerate all the possible values that can be stored in the PC:

```
  %1 = call i32 user_input()
  %2 = shl i32 %1, 2 ; [0 + 4 * %1]
  %3 = icmp ult i32 %1, 8 ; (%1 < 8, uint)
  br %3, label %lower, label %exit
lower:
  %4 = add i32 %2, 9 ; [9 + 4 * %1, %1 < 8, uint]
  %5 = load i32, i32 *%4
  store i32 %5, i32 *@pc
```

The `shl` instruction affects the factor b, the `add` instruction the offset a, while the comparison instruction enforces an upper bound of `8` on `%1` in the `lower` basic block. Note that, since the constraint on `%1` is unsigned (`uint`), we also know the lower bound, i.e., `0`.

At this point, since OSRA cannot handle the load instruction (except for loads from a CSV), SET is required to recover the values of the PC. SET will start from the store instruction, record the load and enumerate all the possible values of `%4` provided by OSRA. It will then perform a load from those values, interpreting them as addresses, effectively producing all values that the PC can assume.

4. REV.NG Design

The basic approach illustrated in Section 2 lifts binary code into LLVM IR and recovers (rudimentary) information about the control-flow graph. While the presented approach suffices for cross-ISA binary translation (due to the option to fall back to an "oracle" mapping table that mitigates imprecision in the analysis at run-time), the precision is too low to accurately identify function boundaries.

Recovering an *accurate* control-flow graph is a much more ambitious and challenging process. For this reason, in the first part of this section we substantially extend and increase the CFG analysis to improve the accuracy. In the second part of the section we describe the function boundary recovery process that is only possible on an accurate CFG.

4.1 Handling of reaching definitions

OSRA propagates tracked values across load/store instructions. Therefore, it is critical to know which definitions (i.e., store instructions) reach a certain load and viceversa. This information is provided by our *reaching definition analysis*.

Here, we introduce three extensions to the basic reaching definition analysis used in [6]: (i) merging reaching definitions, (ii) path-sensitive merging, and (iii) conditional reaching definitions. We also discuss how these improvements are integrated into OSRA. These extensions improve the number of jump targets recovered from indirect jumps in common scenarios encountered while analyzing binaries of different architectures, increasing the accuracy of the recovered CFG.

Merging reaching definitions. One of the limitations of OSRA consists in the loss of precision every time a load is reached by multiple definitions. In this case, the load instruction is associated with a $\top$ value, i.e., a self-referencing, unconstrained OSR. Therefore, all constraints available through the reaching definitions are lost, resulting in an over-approximation that reduces the accuracy of the overall analysis.

Defining a merging policy for the OSRs associated with the reaching definitions addresses this challenge.

Suppose a load instruction is reached by n reaching definitions r_i associated with an OSR in the form $a_i + b_i \times x_i$, with $x_i \in [c_i, d_i]$. All the OSRs of the reaching definitions are considered. The merge is performed only if all the multiplying factors

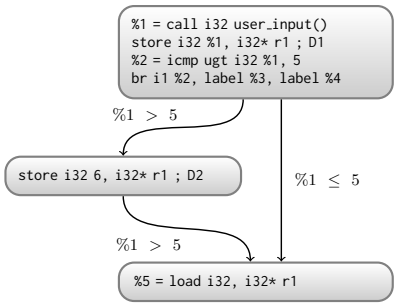

Figure 2. LLVM IR example of the need for *path-sensitive merging*. The branch instruction propagates the constraint $\%1 > 5$ in the *true*-branch and $\%1 \leq 5$ in the *false*-branch. The final basic block receives both constraints, resulting in useless information about $\%1$. However, from the perspective of the `load` $\%5$, only the constraint coming from the *false*-branch is relevant, since the store in the *true*-branch aliases the load from $r1$.

b_i are the same. If this is the case, a_i/b_i is added to the lower and upper bounds (c_i and d_i) of each *bounded value*. All the resulting *bounded values* are then merged according to the *or*-policy (i.e., computing the union of the constraints). If the resulting set of ranges can still be represented with the expressive power of a *bounded value* (i.e, as a single contiguous range, possibly negated), then it is employed to build a new *bounded value* that will be associated with an OSR referencing the load instruction itself. Finally, an OSR, having $a = 0$ and $\forall i, b = b_i$, will be assigned to the load instruction.

Path-sensitive merging. Even in the presence of the above merge policy, due to how constraints are propagated, some useful constraints known to hold on the path from the definition to the load may not survive until they reach the load. Such a situation can be explained by tracking the reaching definitions of the load $\%5$ in Figure 2. Consider the definition $D1$ that uses $\%1$. When the control-flow splits due to a conditional branch we obtain two opposite constraints about $\%1$ on the successor basic blocks: $\%1 > 5$ and $\%1 \leq 5$. When the two paths later merge again, those constraints cancel each other out. However, we are only interested in the constraint on $\%1$ along one of the two paths from $\%1$ to the load $\%5$: the one on the *false*-branch. In fact, along the path that goes through the *true*-branch we have $D2$, a definition of $r1$ aliasing $D1$.

To handle such cases, we need to make the merge policy path-sensitive. The key idea of the *path-sensitive merge policy* is to visit, in depth-first order, all the ancestors of the basic block containing the instruction l loading the variable x and stop when a basic block containing a definition d of x is met. When this happens, all the constraints on the path from d to l on the value associated to the OSR of d are considered. First, since all of them have to hold on the path from d to l, they are *and*-merged together, i.e., the intersection of the constraints is computed. Then, the resulting constraint is accumulated in a $result$ variable through an *or*-merge policy. Therefore, at the end of the process, $result$ will contain a constraint holding on all the paths from each definition of x to the load d.

Algorithm 1 details how the *path-sensitive merge policy* is implemented. First we initialize the $result$ variable with an empty constraint and create an empty stack s_i for each reaching definition i. Then, another stack ws is created to support our depth-first exploration of the ancestors of the basic block l containing our target load l. ws contains a pair of basic blocks $\langle a, b\rangle$ which are used

Data: The basic block of the target load (l) and the set of basic blocks of its reaching definitions ($d = \{d_i\}$).
Result: The merged constraint $result$.
$result = \bot$;
create an empty constraints stack s_i for each reaching definition i;
create a stack ws of $\langle basic\,block, basic\,block\rangle$ pairs;
ws.push($\langle l$, firstPredecessor(l)$\rangle$);
while ws *is not empty* **do**
  $\langle origin, cur\rangle = ws$.pop();
  foreach *reaching definition* i **do**
    cut s_i to the height of ws;
    $c =$ getConstraint($i, origin, cur$);
    s_i.push(c);
  if cur *is not the last predecessor of* $origin$ **then**
    ws.push($\langle origin$, nextPredecessor($origin, cur$)$\rangle$);
  $stop =$ false;
  foreach d_i *in* d **do**
    if $d_i = cur$ **then**
      $tmp = \top$;
      foreach *constraint* c_k *in* s_i **do**
        $tmp = tmp$ and c_k;
      $result = result$ or tmp;
      $stop =$ true;
  if *not* $stop$ **then**
    ws.push($\langle cur$, firstPredecessor(cur)$\rangle$);
return *result*;

Algorithm 1: The *path-sensitive merging* algorithm for constraints. firstPredecessor(a) returns the first predecessor of basic block a, nextPredecessor(a, b) returns the next element in the list of predecessors of a after b, while getConstraint(i, b, c) returns the constraint on the i-th reaching definition holding on the edge $c \rightarrow b$ (i.e., from basic block c to basic block b).

to identify the edge $b \rightarrow a$ that needs to be explored next. ws is initialized with the edge going from l to its first predecessor.

The algorithm keeps considering the top element of the stack ws until all the paths from the reaching definitions i to l have been explored. In each iteration, the next element to be visited is first recorded on ws and then the $cur \rightarrow origin$ edge is considered.

First of all, all the stacks s_i are reset to the same height as ws. Then, the constraints holding on $cur \rightarrow origin$ on each reaching definition i are pushed onto the respective stack s_i. If cur contains one of the reaching definitions i, all the constraints on the stack s_i are *and*-merged and the result is accumulated in $result$ (with an *or* merge policy). On the other hand, if cur does not contain a reaching definition we can proceed one level deeper in our exploration, and the edge coming from the left predecessor of cur is registered on the stack ws.

If we apply this approach to Figure 2, we first observe that $D1$ is expressed in terms of $\%1$, therefore its stack s_{D1} will collect constraints on $\%1$. $D2$ does not need a stack, since it is a constant definition. We start from the last basic block, proceed to its first predecessor, push $\%1 > 5$ on s_{D1} and find the definition $D2$. Since the definition is constant, we directly *or*-merge it in $result$, obtaining the constraint $\%5 = 6$. Note that the constraint $\%1 > 5$ is ignored, since it is not related to the value being stored in the definition $D2$. At this point, we remove an element from the stack s_{D1}, proceed to the right predecessor, push $\%1 \leq 5$ on s_{D1} and meet $D1$. By *and*-merging all the constraints on s_{D1} we obtain

```
100: cmp   r2, #0          1: lt = r2 < 0;
104: addlt r2, r2, #1      2: if (lt) { r2++; }
108: blt   exit            3: if (lt) { return; }
10c: add   r1, r1, r2      4: r1 = r1 + r2;
```

Figure 3. On the left, ARM assembly snippet with multiple consecutive instructions sharing the same predicate. On the right, equivalent C pseudo-code.

$\%1 \leq 5$, which is in turn *or*-merged into $result$, leading to the final constraint $\%1 \leq 6$, as expected.

To keep the algorithm simple, each edge is visited only once. Note that our analysis only considers load instructions accessing a CSV or the address pointed by a CSV plus a constant offset. As a consequence, performing a conservative alias analysis is straightforward. Note also that employing the *path-sensitive merging* policy is resource demanding, and, therefore, we only employ it as a fallback if the straightforward analysis is not successful.

Conditional reaching definitions. Depending on how reaching definitions are computed, the accuracy of our system varies. In particular, the naïve implementation may lead to spurious reaching definitions in ISAs heavily employing predicated instructions.

Consider the ARM assembly snippet on the left of Figure 3. The `r2` definition at `0x104` (the add instruction) should not reach the use at `0x10c`, since, if the addition is executed, the branch gets executed too (they share the same predicate). However, a binary analysis system has to consider each instruction on its own, therefore it will interpret those instructions as illustrated in pseudo-C on the right of Figure 3. In this situation, a traditional reaching definition analysis would propagate the `r2` definition in the body of the `if` block to line 3. Then, since no other definition aliases it, it would not only be further propagated to the body of the second `if`, but also (incorrectly) to the next instruction, reaching the use at line 4.

For this reason, we created the *condition numbering analysis* (CNA), which groups all conditional branch instructions that share the exact same condition. CNA checks each pair of conditions to verify if they compute the same operation on either the same operands, or on operands reached by the same set of definitions. Such a grouping mechanism is efficiently implemented through a hash-map using an appropriate hash function considering the involved operations and their operands.

The CNA's results are then employed by the *conditional reaching definitions analysis*. This analysis, along with tracking definitions, records whether a condition identified by CNA (or its negation) holds in each basic block.

When a definition is propagated to the successors of a conditional branch, the identifier of the branch's condition is retrieved, and, if present in the set of conditions known to hold in the basic block containing the definition, it is propagated only to the *true*-successor. Otherwise, if the negated condition is present, it is propagated only to the *false*-successor.

In practice, going back to the example in Figure 3, the two `lt` conditions are identified by the same integer, say 42. Therefore, when propagating the definition at line 2 in line 3, the branch condition is inspected, and since it is also identified by 42, the definition proceeds only towards the *true*-branch, preventing it from reaching line 4.

4.2 Function boundaries recovery

The function boundary identification process is split into five steps that we present in the following.

1. Identify call/return instructions. Since, by design, our analyses have to be ISA-agnostic, we assume that the underlying IR used for the analysis does not explicitly provide the concept of *function call* or *return* instructions. For this reason, we redefined these two concepts in an architecture-agnostic way.

Function call. A branch instruction preceded by an instruction performing a store of a constant integer matching the next PC, considering delay slots if necessary. This integer is the *return address*.

Return. Any indirect branch instruction whose destinations are either unknown or an address known to be a *return address*.

Note that these definitions are generic enough to handle all the known actual implementations of function call instructions in real ISAs. Specifically, the function call definition successfully captures both architectures saving the return address in a register (e.g., `lr` for ARM, `ra` for MIPS) or on the stack (e.g., x86). The first step in the function boundary recovery process consists in scanning the code for instructions matching the *function call* definition, and then, once all the possible *return addresses* have been collected, for instructions matching the *return* definition.

2. Identify initial candidate set. The second step collects an initial set of *candidate function entry points* (or CFEPs). Specifically, we have three initial types of CFEPs.

(a) **Called jump targets.** The most important and reliable source of CFEPs are function calls, since they explicitly indicate that their destinations are functions.

(b) **Unused jump targets in global data.** Global data can also be a source of CFEPs, e.g., due to function pointers stored in global data or C++ virtual tables. However, global data also contains jump tables used to implement C `switch` statements. These addresses do not represent pointers to a function and may lead to a large number of false positives. For this reason, we only consider the *unused* portion of global data. By *unused* we mean that a specific interval in global data has never been accessed by SET. In fact, as mentioned in Section 3.3, SET can read global memory areas to materialize addresses contained in a jump table, which are therefore blacklisted.

(c) **In-code constants.** The code itself can contain function pointers, for instance if a function pointer is materialized in a register and then stored to memory. All the jump targets recovered by SET are considered and filtered: we register only jump targets that never end directly in the PC and that are never used as a load address. The rationale behind these choices is that values ending up in the PC will become part of the regular CFG of the program and we can handle them in other ways, while if a load is performed at a certain address, we assume that the target is data, and not code.

We say that a CFEP has its *address taken* if it is of type (b) or (c).

3. Identify reachable basic blocks. Once a preliminary set of CFEPs is available, for each one of them we follow the CFG and associate each basic block reachable from there with the CFEP. When we reach a call instruction we do not follow it, but we proceed to its return address, and when we reach a return instruction, we stop our exploration. Moreover, when associating a basic block with a CFEP, we also keep track of how we reached it, that is either through regular control-flow or by proceeding to the return address of a call instruction.

Once all the basic blocks reachable from a CFEP have been identified, each branch instruction is inspected again to verify if it is a *skipping jump*. A *skipping jump* is a jump instruction that has at least a CFEP of the type (a) between its location and its destination. This check is performed to identify if the branch instruction is jumping over a basic block we reliably know to be the entry point of

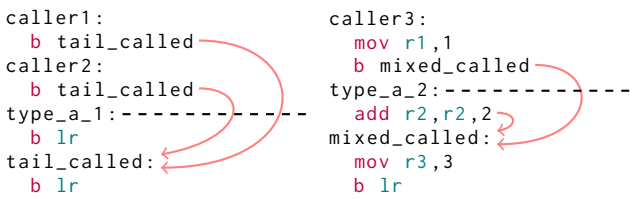

Figure 4. ARM example of *skipping jumps*. `tail_called` is a valid CFEP, since it can be reached only through *skipping jumps* (going on over the type (a) CFEP `type_a_1`). On the other hand, `mixed_called` is discarded as a CFEP and its basic basic block is considered part of `caller3` and `type_a_2`. In fact, `mixed_called` can be reached both through a *skipping jump* (coming from `caller3`), but also through the fall-through path after the `add` instruction in `type_a_2`.

a function. This type of instructions are often a hint for the presence of a tail call, therefore we create a new CFEPs out of their targets and process them as described.

4. Filter candidates. At this point we have sufficient information to perform an evaluation of which CFEPs should be kept, and which should instead be discarded. The criterion to keep or discard a CFEP is expressed with a simple rule:

The CFEP is kept if it is reachable exclusively through call instructions or skipping jumps.

In practice we want to keep all the CFEPs except those that are reachable through the local CFG of another function (i.e., return paths from a function call, fallthrough paths or jumps not going over other CFEPs). This means that we preserve CFEPs whose addresses are taken or are reachable only through tail calls, as long as they do not appear to be part of the local CFG of another function. On the other hand, we discard CFEPs which are reachable both through the local CFG of a CFEP and *skipping jumps*, since this is a strong hint that the *skipping jump* is not a tail call, but simply a result of two functions sharing some code.

Figure 4 reports an example of a CFEP reachable only through *skipping jumps* (`tail_called`, on the left) and a CFEP reachable both through a *skipping jump* and function-local control-flow (`mixed_called`, on the right). The latter example, is a typical situation where two functions share the main part of their body but have slightly different headers. In these situations, we deem it appropriate to assign the basic blocks of the main part of the body to both functions.

5. Finalize the set. The last step consists in promoting all the jumps to the survived CFEPs to the status of function calls and consequently recompute for each one of them the set of basic blocks reachable from the entry point. The final result is a set of functions, possibly sharing code.

`noreturn` functions. Consider the following ARM snippet:

```
main:
  add     r0, r0, #3
  bl      exit
hello:
  add     r0, r0, r1
  bx      lr
```

Note how the `main` function does not have a return instruction, in fact it is not necessary since the `exit` function will never return. In C terms, `exit` is known as a `noreturn` function.

While exploring the basic blocks reachable from `main`, if our analysis does not identify `exit` as a `noreturn` function, we might mistakenly assign basic blocks belonging to the `hello` function to `main`. Therefore, identifying `noreturn` functions is paramout to accurately recover function boundaries.
We detect the following types of `noreturn` functions:

Syscall wrappers. Before each syscall we inject an instruction loading the CSV associated with the register holding the syscall identifier. In this way, our reaching definition analysis will provide a list of all the reaching definitions. These definitions are monitored by SET, and, in case we notice that one of them writes a constant value corresponding to the identifier of a `noreturn` syscall (such as `exit`), we mark its basic block as a *killer basic block*.

Infinite loops. We mark all the basic blocks belonging to a loop in the CFG of a function with no exit nodes (i.e., an infinite loop) as *killer basic blocks*. Such a situation is typical in the implementation of the `abort` function as a last chance to prevent execution from proceeding, in case raising a signal does not have the desired effect.

`longjmp`. Our analysis also looks for basic blocks that overwrite the stack pointer register with a value that is neither obtained as an offset from its previous value (e.g., $\text{sp} = \text{sp} + 8$) nor loaded from a memory address relative to its value (e.g., $\text{sp} = \mathit{load}(\text{sp}-16)$). Such a behavior typically identifies the `longjmp` function and its derivatives. Such basic blocks are marked as *killer basic blocks* too.

At this point, all the *killer basic blocks* are temporarily modified to have a single successor: the *sink*. All the nodes post-dominated by the *sink* are in turn marked as *killer basic blocks*. In practice this means we reach the entry point of functions such as `abort`, `exit`, `execve`, `longjmp`, and all their wrappers and correctly identify them as `noreturn` functions.

5. Experimental results

The analysis framework and the set of analyses presented in the previous section have been implemented in a tool, REV.NG. REV.NG is a C++ project composed of 8,011 SLOCs (according to the `sloccount` tool) publicly released under a Free Software license.

REV.NG makes heavy use of LLVM (version 3.8) and the *tiny code generator* (TCG) of QEMU (version 2.5.0). While the TCG supports many different architectures[1], to evaluate our prototype, we focused on Linux binaries on three popular architectures:

MIPS. Using GCC 5.3.0 and clang 3.8 with musl [11].
ARM. Using GCC 5.3.0 and clang 3.8 with uClibc [5].
x86-64. Using GCC 4.9.2 and clang 3.8 with musl [11].

Our choice was guided by the diversity of their features such as register size, presence of delay slots, support of predicate execution, CISC/RISC designs, endianness, and variable-length instruction encoding. To test the robustness of our approach, all the binaries we employed were stripped of debugging information and linked statically.

Note that statically linked binaries provide less information than dynamically linked executables, since the dynamic table and the dynamic symbols are missing. This also means that our tool handles the C standard library, which is large, includes hand-written assembly and other manually optimized pieces of code which are not typically found in programs. Extending our work to support dynamically linked programs consists in loading the main binary and all its libraries and perform our analyses on the whole code

[1] QEMU supports the Alpha, ARM, CRIS, x86, MicroBlaze, MIPS, OpenRISC, PowerPC, RISC V, System Z, SuperH, SPARC and Unicore architectures and their 64-bit versions, where applicable.

corpus. In summary, using statically linked binaries, results in the most challenging setting.

The only "structural" information we left available to the evaluated tools was the section list, which allows distinguishing between code and data regions. This distinction enables, e.g., REV.NG to exclude spurious jump targets and function calls which would introduce noise in our evaluation. Section information is preserved even when `strip`ing an ELF binary. Symbols are only employed to collect the ground truth: our tool never uses them to recover function boundaries.

5.1 Accuracy of the recovered function boundaries

We built the 105 programs of the coreutils project for the three architectures, including `md5sum`, `ls`, `install`, `df`, `cp`. The programs, including the C standard library, have been compiled using GCC and clang in three different configurations: optimized for performance (`-O2`), aggressively optimized for performance (`-O3`) and optimized for code size (`-Os`). Since uClibc does not support clang, for ARM, the C standard library has been compiled using GCC in all the configurations. Table 2b reports the average size of the code section (`.text`) for each tested configuration.

Collecting ground truth for *control-flow graphs* is a challenging task. In GCC, CFG information cannot be obtained, since the backends implicitly generate basic blocks by printing strings of assembly. We considered LLVM, but since *each backend* must be instrumented in non-trivial ways, it would have resulted in prohibitive engineering effort. For this reason, we focused the evaluation on the accuracy of the recovered function boundaries instead. Since an accurate CFG is a requirement for recovering accurate information about function boundaries, the presented results can be considered a *lower bound* for the accuracy of the CFG itself.

The ground truth for function boundaries is easier to recover. Specifically, we employed the `STT_FUNC` ELF symbols from the binaries, which provide the starting address and size for each function. From these ranges we excluded constant pools, since they should not be translated, and `nop` instructions, since they are mostly used for function and instruction alignment purposes.

We compare our results against related work by letting the tools produce a CFG. Then, starting from each entry point, the CFG is explored, and each basic block reachable from there is recorded as part of that function. Since one of our goals was to assess the quality of the CFG, we ignored the basic blocks that were assigned to a function but that were not reachable from the entry point, since this means that the tool assigned a basic block to a function but could not understand how it takes part of the CFG of the function.

We compared REV.NG with IDA Pro 6.6 using a custom IDAPython script, the most recent version of angr (as of November 2016) employing the `CFGFast` CFG recovery option [17], and BAP 0.9.9, which implements the ByteWeight approach [2], with the `--phoenix` option and collecting the CFG data from the GraphViz output. We employed the latest available ByteWeight signatures and we extended BAP to output the size of basic blocks. Note that BAP does not support MIPS.

Table 1 reports the results of our experiments. The most important information needed to assess the quality of the results is the *Jaccard index*, which we computed for each detected function against its best match in our ground truth. The index is computed comparing the set of the basic blocks assigned to the function against the actual set of basic blocks, according to the ground truth. The Jaccard index provides a concrete measure of the accuracy of the match, penalizing missing or extra basic blocks. Table 1 reports another interesting metric reported in the *Matched* column, that is the percentage of matched functions, ignoring the quality of the match.

The results in terms of accuracy of REV.NG are very close to those of IDA Pro, and sensibly better than those of BAP and angr in all the tested configurations. The difference between IDA Pro and REV.NG comes from few functions that only IDA Pro identifies. By performing manual inspection of the functions detected by IDA Pro but not by REV.NG, we verified that in most cases it is dead code, i.e., code whose address is not *taken* and has no direct control-flow transfers pointing to it. This is due to the fact that the heuristics implemented in IDA Pro can detect function prologues. However, since in all inspected cases the difference was due to dead code, we do not consider this a limitation but a design choice.

Note that we already tried to mitigate this problem by compiling our code using the `-ffunction-sections` GCC option, along with `-Wl,--gc-sections`, which is supposed to minimize the amount of dead functions. However, hand-written assembly functions are not pruned.

In addition to dead code, we found additional sources of inaccuracy in the CFG, which affect both REV.NG and IDA Pro:

Aggressively optimized nested `switch`. In certain x86-64 functions using nested `switch` statements, REV.NG was unable to track the size of the jump tables used by the inner `switch` statement. Since the starting address of the jump table was available, we could devise a heuristic to recover it. However to provide coherent results, we decided not to do so.

Jump table addresses spilled on the stack. In certain situations, e.g., in MIPS, GCC can spill the starting address of a jump table on the stack in the function prologue because it might be used multiple times across the function. While REV.NG implements a basic mechanism to track stack values, doing so across function calls is non-trivial, since during the CFG recovery phase we have no information about function calls.

In our (non-exhaustive) exploration, we did not find major inaccuracies in the CFG that IDA Pro handled and REV.NG did not. On the contrary, we found several examples where REV.NG is more precise than IDA Pro. The next section discusses such an example as a case study.

Looking at BAP's results, we found that (i) some functions were missing several instructions in the function prologue and (ii) a series of spurious functions in the middle of actual functions manipulating the stack (e.g., for variable-length arrays). Tail calls and indirect jumps due to `switch` statements also appeared to be handled poorly.

For what concerns angr, despite often matching the most functions compared to the other tools, the accuracy of the matching is lower. The main issues are related to mishandled predicated return instructions, code after `noreturn` function calls forced to a new function in all cases, and incomplete handling of certain indirect jumps due to `switch` statements. It is worth noting that the angr project was hand-tuned for the x86 architecture, which we did not evaluate.

In conclusion, REV.NG results are comparable or sensibly better compared to the other evaluated tools, proving the effectiveness of our approach.

Memory usage and processing time. The last set of rows in Table 1 reports, for each tool, the peak memory usage (*RSS* column) averaged over all the 105 processed binaries. As the table shows, REV.NG memory usage is mostly comparable to other tools on x86-64, but on ARM and MIPS our tool results to be more resource demanding. This is due to a limitation of our current implementation. Specifically, OSRA propagates each constraint indefinitely, leading to a considerable memory overhead. Our development branch addresses this issue by limitating the propagation of constraints on a SSA value up to the farthest instruction employing it in an

			x86-64				MIPS			ARM			
			IDA	REV.NG	BAP	angr	IDA	REV.NG	angr	IDA	REV.NG	BAP	angr
Jaccard index	GCC	-O2	97.98	98.00	84.47	90.92	96.37	94.30	84.95	96.56	95.60	79.10	67.13
		-O3	98.35	96.72	85.31	91.06	95.17	93.16	82.94	96.41	95.60	79.08	66.65
		-Os	98.16	98.77	87.03	93.08	97.10	94.79	88.82	95.92	94.08	81.67	66.38
	clang	-O2	98.39	98.31	85.84	92.05	96.41	92.48	78.50	94.62	94.49	78.34	65.98
		-O3	98.34	97.77	82.73	91.90	96.31	91.65	78.22	94.64	94.53	78.16	63.43
		-Os	97.83	98.44	83.70	91.01	96.52	92.66	81.07	94.01	93.38	80.84	65.89
Matched (%)	GCC	-O2	94.56	98.37	83.51	93.76	93.09	98.13	93.32	85.59	88.77	80.31	97.27
		-O3	93.25	98.43	83.30	94.59	92.33	97.58	95.44	84.18	87.90	79.35	97.47
		-Os	95.31	97.90	83.39	93.15	92.60	97.03	93.21	88.22	87.82	78.23	97.37
	clang	-O2	94.31	98.83	67.42	94.29	84.91	83.26	78.26	84.01	86.92	77.37	97.20
		-O3	94.27	98.78	59.79	94.64	85.66	83.35	78.33	83.45	86.38	77.46	94.41
		-Os	94.91	98.65	59.20	93.85	86.35	84.50	79.06	86.91	85.65	76.87	96.60
RSS (MiB)	GCC	-O2	227.88	248.59	284.62	297.63	195.86	628.95	201.24	198.62	463.88	264.71	261.58
		-O3	228.67	289.81	286.67	297.03	197.72	4325.39	1017.87	199.01	568.37	280.59	273.94
		-Os	227.08	229.48	261.51	256.15	196.02	784.30	223.17	198.24	369.03	235.64	259.80
	clang	-O2	226.67	215.32	200.39	213.04	196.63	549.36	233.53	198.73	401.64	268.06	244.27
		-O3	227.07	236.36	177.09	204.82	196.19	584.53	207.99	198.99	439.32	282.41	508.11
		-Os	226.47	743.53	149.52	734.89	196.06	574.46	224.67	198.29	1293.56	245.84	676.03

Table 1. Comparison of the experimental results obtained by IDA Pro, REV.NG, BAP, and angr on the 105 coreutils binaries compiled using GCC and clang for x86-64, MIPS and ARM. Three different optimization levels have been employed: `-O2`, `-O3`, and `-Os`. Note that BAP does not support MIPS. The *Matched* rows represent the percentage of functions that have been matched at least partially, ignoring the quality of the match. *Jaccard index* is the average Jaccard index of each detected function against its best match in the set of original functions. The average is weighted over the size of the function and over the size of the `.text` section of each program. Finally, *RSS* is the average (over all the binaries) of the maximum resident set size (i.e., peak memory usage).

	x86-64	MIPS	ARM
IDA	5.17 s	8.06 s	5.39 s
REV.NG	44.57 s	222.11 s	119.53 s
BAP	35.33 s	n.a.	25.48 s
angr	384.15 s	273.83 s	147.82 s

(a)

		x86-64	ARM	MIPS
GCC	-O2	115.12 kiB	98.50 kiB	115.12 kiB
	-O3	161.12 kiB	105.01 kiB	161.12 kiB
	-Os	123.81 kiB	89.11 kiB	123.81 kiB
clang	-O2	138.72 kiB	102.56 kiB	138.72 kiB
	-O3	125.51 kiB	106.03 kiB	125.51 kiB
	-Os	126.59 kiB	95.12 kiB	126.59 kiB

(b)

Table 2. Table (a) reports the time spent (in seconds) to collect the control-flow graph and the function boundaries of the `ls` binary, compiled with `-O2` using GCC for x86-64, MIPS and ARM. The presented results are averaged over 10 runs. Table (b) reports the average size (in kilobytes) of the `.text` section of a coreutils program compiled using the specified configuration.

```
memset:
copy_loop:
  cmp     r2, #8              ; A
  blt     remaining           ; B
  ; ...
  sub     r2, r2, #8          ; C
  cmp     r2, #8              ; D
  subge   r2, r2, #8          ; E
  bge     copy_loop           ; F
remaining:
  add     pc, pc, r2, lsl #2  ; Z
  nop
  strb    r1, [r3], #1
  strb    r1, [r3], #1
  strb    r1, [r3], #1
  strb    r1, [r3], #1
  strb    r1, [r3], #1
  strb    r1, [r3], #1
  strb    r1, [r3], #1
  mov     pc, lr
```

Figure 5. uClibc ARM implementation of `memset`. `r2` contains the size of the buffer. `copy_loop` is a (partially omitted) unrolled loop copying 8 bytes at a time, while `remaining` takes care of copying the bytes left over by `copy_loop`. Each one of the 7 `strb` instructions copies a single byte; "`add pc, ...`" jumps to one of them depending on how many bytes are left to copy.

OSR. In our preliminary testing of such a solution on the `ls` binary, the memory consumption is reduced from 1.78 GiB to 899 MiB on MIPS and from 1.19 GiB to 476 MiB on ARM.

Since the continuous integration system where we run our tests is composed by servers featuring different hardware specifications, Table 1 does not report timing results. Instead, we collected timing results on a single machine with 32 GiB of RAM and an Intel i7-6820HQ CPU, with 4 physical cores clocked at 2.7 GHz. We ran each one of the four tools 10 times against the `ls` binary compiled for MIPS, ARM and x86-64. Table 2a reports the results. As expected, tools such as IDA Pro and BAP, which employ heuristics or machine learning techniques, are notably faster compared to REV.NG. On the other hand our prototype implementation outperforms angr. Note however that the detailed information we collect can serve as a basis for more sophisticated analyses whose purposes goes beyond recovering the CFG or the function boundaries. In fact, if we compare the time taken by the IDA Pro's Hex-Rays Decompiler to analyze the whole binary, we get more comparable results. In particular, the Hex-Rays decompiler took approximately 37 s to analyze `ls` compiled for x86-64 and 27 s for the ARM version. Note also that the previously mentioned development branch of REV.NG reduces the analysis time for `ls` from 44.57 s to 31.59 s on x86-64, from 222.1 s to 15 s on MIPS and from 119.53 s to 50.53 s on ARM.

5.2 Case study: the buggy memset

Figure 5 shows a simplified version of the ARM `memset` implementation included in uClibc [5]. It is a hand-optimized implementation that copies 8 bytes at a time (see the `copy_loop` label), and then copies the (at most 7) leftover bytes one by one (see the `remaining` label).

We consider the recovery of the CFG of this function interesting for several reasons. Specifically, it would be beneficial to prove that instruction `Z` can only reach one of the 7 `strb` instructions or the return instruction. As we will see, most of the analyses presented in Section 3.3 have to be employed.

First of all, `r2` cannot be expressed in terms of a single value. In fact, its usage in `Z` can be affected by the definition in `C`, `E` or any other definition of `r2` in the callers of `memset`. A merge policy with multiple reaching definitions must therefore be leveraged.

Second, to have an accurate set of definitions of `r2` reaching `Z`, we need to handle predicate instructions correctly. If the naïve reaching definition approach is employed, the definition in `E` is propagated on both successors of `F`, effectively preventing the definition in `C` from reaching `Z`.

Third, if the adopted merge policy is not *path-sensitive*, the `r2` use in `Z` sees two constraints on the `r2` definition in `C`: `r2` < 8, through the path `CDZ`, and `r2` ≥ 8, through the path `CDEFABZ`. This makes the definition of `r2` in `C` unbounded in `Z`, preventing the analysis from proving that `r2` is lower than 8 in all cases. However, using the proposed *path-sensitive* merge policy, the constraint `r2` ≥ 8 is ignored: going backward through the `CDEFABZ` path, the definition in `E` prevents the analysis from reaching `C` and taking into account the constraint associated to that path. In conclusion, REV.NG was able to correctly recover all the jump targets, unlike all the other tools we tested.

The last reason why this `memset` is relevant consists in a bug it contains and that we discovered through REV.NG. All the comparisons performed by this function are signed. Therefore, a malicious user in control of the size parameter of `memset` (`r2`) can control the program counter by simply passing a negative number. This bug was recently fixed in uClibc-ng [4].

6. Related work

Traditional techniques used to identify function start points employ manually crafted patterns and then use recursive disassembly to identify the set of bytes belonging to a function body. Such techniques are adopted in current tools, both commercial and research-oriented, such as IDA Pro [8], Dyninst [9, 7], as well as in other disassemblers [21], and angr [16, 17].

angr [16, 17] adopts an approach similar to ours, since it employs VEX, Valgrind's IR, to perform their analysis. However, VEX is only available for a subset of the architecture handled by QEMU. Most importantly, the largest part of their effort for accurate recovery of CFG and function boundaries relies on symbolic execution, which hinders the scalability of the approach.

Rosenblum et al. [13] employed machine learning to address function boundary identification, overcoming variation in the function start due to compiler-related effects such as optimization or scheduling. Basically, they proposed to automatically generate the set of function start patterns from a large corpus of binaries, instead of crafting it manually.

ByteWeight [2] refines this idea, leveraging machine learning classification to label each byte of a program as a function start or not. It employs weighted prefix trees of function start sequences in place of a pattern collection, followed by static analysis (recursive disassembly combined with value set analysis [1]) to detect the remaining bytes of the function. Shin et al. [15] aim at improving precision and speed of recovery over ByteWeight employing recurrent neural networks.

For all of the above mentioned machine learning-based approaches, the main goal is to reconstruct a set of probable start patterns. Our technique is fundamentally different, in that it relies on code pointers to identify function starting points. We leverage data-flow analyses that can provide more fine-grained and precise information about the tracked values with respect to value set analysis, as shown in [6].

7. Conclusions and future works

Recovering CFG and function information from binaries is an important technology that enables further analyses like static bi-

nary instrumentation, security analysis, reverse engineering, or retrofitting defense mechanisms.

We design a set of analyses that statically recover the CFG and the function boundaries of a binary with high accuracy, without relying on ISA-specific heuristics. While heuristics can be very effective (as shown by IDA Pro), they are not portable to other architectures and require vast manual effort. Developing ISA-independent analyses simplifies (or even removes) any porting efforts while retaining overall comparable results. Such approaches meet the sweet spot between black box analysis employing machine learning and hard-coding ISA-specific heuristics.

In the future, we plan to (i) automatically detect the calling convention employed by individual functions (i.e., which registers are clobbered and which ones are preserved) and (ii) infer the stack frame layout (i.e., location and usage of local variables, register spill areas, and function parameters). This will allow us to further increase the accuracy and leverage our tool for additional analyses.

Acknowledgements

We thank the anonymous reviewers for their constructive feedback on this work. The work was supported, in part, by the National Science Foundation under grants number CNS-1513783 and CNS-1657711. REV.NG is available as Free Software at `https://rev.ng/`.

References

[1] Gogul Balakrishnan and Thomas Reps. *Compiler Construction: 13th Int. Conf., CC 2004*, chapter Analyzing Memory Accesses in x86 Executables, pages 5–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[2] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 845–860, Berkeley, CA, USA, 2014. USENIX Association.

[3] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 463–469, Berlin, Heidelberg, 2011. Springer-Verlag.

[4] Lucian Cojocar. Commit fixing the memset bug in uClibc-ng, 2016. `http://bit.ly/2cx2Lp2`.

[5] Erik Andersen. uClibc, 2012. `https://www.uclibc.org/`.

[6] Alessandro Di Federico and Giovanni Agosta. A jump-target identification method for multi-architecture static binary translation. In *Proceedings of the 2016 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (to appear)*, CASES '16, Piscataway, NJ, USA, Oct 2016. IEEE Press.

[7] Laune C. Harris and Barton P. Miller. Practical analysis of stripped binary code. *SIGARCH Comput. Archit. News*, 33(5):63–68, December 2005.

[8] Hex-Rays. IDA Pro. `http://bit.ly/1gybdzm`, retrieved Feb. 2016.

[9] Xiaozhu Meng and Barton P. Miller. Binary Code is Not Easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 24–35, New York, NY, USA, 2016. ACM.

[10] Mathias Payer, Antonio Barresi, and Thomas R. Gross. Fine-Grained Control-Flow Integrity Through Binary Hardening. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings*, 2015.

[11] Rich Felker. musl. `https://www.musl-libc.org/`, 2016.

[12] RISC-V Foundation. riscv-qemu, 2016. `https://riscv.org/software-tools/riscv-qemu/`.

[13] Nathan Rosenblum, Xiaojin Zhu, Barton Miller, and Karen Hunt. Learning to analyze binary computer code. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, AAAI'08, pages 798–804. AAAI Press, 2008.

[14] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. LLBT: An LLVM-based Static Binary Translator. In *Proc. of the 2012 Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '12, pages 51–60, New York, NY, USA, 2012. ACM.

[15] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 611–626, Berkeley, CA, USA, 2015. USENIX Association.

[16] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *NDSS*, 2015.

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

[18] Trail of Bits, Inc. MC-Semantics. `http://bit.ly/2geNQEJ`, 2016.

[19] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. PathArmor: Practical ROP protection using context-sensitive CFI. In *ACM CCS*, 2015.

[20] Victor van der Veen, Enes Goktas, Moritz Contag, Andre Pawlowski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A Tough Call: Mitigating Advanced Code-Reuse Attacks At The Binary Level. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, USA, May 2016. IEEE.

[21] Giovanni Vigna. *Static Disassembly and Code Analysis*, pages 19–41. Springer US, Boston, MA, 2007.

[22] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical Control Flow Integrity & Randomization for Binary Executables. In *IEEE Security and Privacy*, 2013.

[23] Mingwei Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 337–352, Berkeley, CA, USA, 2013. USENIX Association.