A jump-target identification method for multi-architecture static binary translation

Alessandro Di Federico
alessandro.difederico@polimi.it

Giovanni Agosta
agosta@acm.org

Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano

ABSTRACT

Static binary translation is a technique that allows an executable program for a given architecture to be translated into a different one, with a reduced overhead compared to emulators and dynamic binary translators. The main downside of the static approach lies in the absence of runtime information, which is available in other solutions. In particular, one of the key issues consists in the identification of data and code in the program, and, more specifically, in the detection of basic block start addresses (jump targets). The presence of indirect jump instructions whose target is not immediately evident, in particular due to C `switch` statements, makes the recovery of jump targets a challenging task.

In this paper, we present an effective technique for jump targets identification composed by an initial step of *global data harvesting* followed by two novel analyses: the Simple Expression Tracker and the Offset Shifted Range Analysis. Both analyses work on a Single Statement Assignment (SSA) intermediate representation and are iterated multiple times until they provide no additional information. In particular, OSRA is a data-flow analysis modeled after the typical code generated for `switch` statements. It tracks each SSA value in terms of an offset, a scaling factor, and another SSA value, comprised between a lower and an upper bound (e.g., $b = 10 + 4 \cdot x$, with $8 \leq x \leq 10$).

To validate the effectiveness of the proposed technique, we employ REVAMB, an in-house tool for binary translation leveraging QEMU and the LLVM compiler framework. Our experimental results show that we are able to run the `coreutils` test suite on ARM, MIPS and x86-64 without significant failures due to unidentified jump targets.

Keywords

Static Binary Translation, Data Flow Analysis, Jump Target Identification

CASES '16, October 01-07 2016, Pittsburgh, PA, USA

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4482-1/16/10…$15.00

DOI: http://dx.doi.org/10.1145/2968455.2968514

1. INTRODUCTION

Binary translation is a technique that, given an executable program (or a portion of it) compiled for a certain architecture (e.g., ARM), aims to translate it into a different one (e.g., x86-64). Binary translation can be performed statically or dynamically. The dynamic version is essentially a type of emulation technique where the emulation, usually a slow process, is sped-up by translating instructions into the host machine code at runtime. The static version is a much more complex endeavor, and it is aimed at building a full executable program which can be then executed autonomously. The motivation for developing a binary translator is provided by a wide range of applications. Historically, legacy code performance portability was the key motivation. Indeed, binary translation techniques have been employed to provide binary compatibility for new platforms, such as the Transmeta Crusoe [7] which achieves better performances than an emulator. Binary translators that employ an *intermediate representation* (IR) can also be used effectively to instrument or analyze existing binary programs [15, 2], as well as to retarget them for different platforms, including significant levels of target-specific optimization [6, 10]. Static binary translators are particularly interesting because they do not impose a runtime overhead as significant as other approaches, and they can be used to generate new binaries that are completely independent of the translation system. However, they pose a number of additional challenges which must be solved in order to produce a complete binary program. In particular, discovering and identifying code in a binary object is a non-trivial task. Ideally, one would start from the entry point of the program, follow all jump and call to subroutine instructions to identify the starting points of all reachable code segments. However, this is made more complex by the usage of function pointers, virtual functions and, most importantly, `switch` statements.

Contributions. In this paper, we propose a systematic approach to identify basic blocks (jump targets) in binary programs by analyzing both the global data and the code itself. The approach is general, as it does not employ heuristics or make architecture-specific assumptions, and is proven effective on a set of real world programs. We also propose a new data flow analysis (OSRA) particularly suitable to identify jump targets introduced by the sophisticated implementations of `switch` statements. Moreover we implemented the abovementioned techniques in our in-house static binary translator, based on QEMU and LLVM, and evaluated their effectiveness on a set of real-world programs on three different architectures with almost no failures due to missing jump targets.

Organization of the paper. The rest of this paper is organized as follows. In Section 2, we state the problem of identifying jump targets in executables and its challenges. In Section 3 and 4, we describe the three steps of our proposed solution, with a particular focus on the OSRA data flow analysis. In Section 5, we provide experimental evidence showing how our approach is effective in the recovery of jump targets in a real-world scenario. In Section 6, we compare our solution with related works, while in Section 7 we draw some conclusions and highlight future research directions.

2. PROBLEM STATEMENT

In this section we present the main challenges in identifying code from a static binary translator perspective, with a specific focus on jump target recovery. In particular, we illustrate a set of problematic cases with examples and investigate their origins.

2.1 Identifying code and basic blocks

One of the key issues in static analysis of binary programs consists in isolating the executable code from the program data. Most binary formats contain useful information to this end. For instance, the ELF binary format [11] divides the program in several *segments* that associate a portion of the file to a load address, a size and a set of permissions (such as readable, executable and writable). The permissions are particularly useful, since executable code must reside in a segment with execution permissions.

ELF *sections* would provide more fine grained information, but since, unlike *segments* they're not critical to execution, they are often absent. This becomes an issue when, as it is often the case, the linker merges `.rodata` and `.text` in a single segment, since they both have read-only access.

Furthermore, code and data can be mixed by the compiler, e.g., when *constant pools* are used in unified cache architectures to reduce the cost of loading constants. Once the problem of distinguishing code from data is solved, basic blocks must be identified to reconstruct the control flow. Basic blocks are delimited by instructions that alter the control flow (branches, jumps and calls), or by *jump targets* – corresponding to labels in the assembly code. It is worth noting that in static binary translation control flow reconstruction could theoretically be avoided. However, there are significant drawbacks if this choice is taken. In particular, control flow reconstruction enables more aggressive optimizations. In the absence of this information, every instruction must be considered as a basic block on its own.

Moreover, in architectures employing a variable-length instruction encoding (VLE), such as x86, a single sequence of bytes would have to be interpreted in several different ways, leading to a needless increase in translation time and output size.

Finally, management of indirect jumps is negatively affected by the lack of jump target information [12]. This is because the static binary translator typically handles indirect jumps through a dispatcher which will select at runtime the correct target from a data structure such as a hash table or a binary search tree. The size of this data structure, and possibly its access time, depend on the number of possible jump targets. Thus, while the presence of false positives among the identified jump targets is not a problem for functionality, marking all instructions as jump targets imposes a serious performance penalty on the recompiled code.

2.2 Challenges in jump target recovery

The target of a jump instruction can be either encoded directly in the jump instruction (a direct jump) or can be the content of a register or memory area at run-time (an indirect jump). Direct jumps can be either relative to the program counter (PC) or absolute, in which case the immediate represents the full destination address. In both cases, obtaining the jump target is straightforward. On the other hand, indirect jumps deserve a closer analysis, as they can derive from several different types of high level statements.

Materialized destination address. A program might need to perform a jump relative to the PC to an address whose distance from the PC is larger than the maximum representable in the instruction immediate. An option to circumvent this situation is presented in the following MIPS snippet:

```
lui     t9,0x42
addiu   t9,t9,0xd188
jr      t9
```

where the full destination address is materialized in a register, and then an indirect jump through that register is performed. Another possible solution consists in storing the full target address in a constant pool.

Return instructions. A return instruction is a form of indirect jump that diverts execution to the address stored in the link register or on the top of the stack.

Function pointers. Calling a function pointer or a C++ virtual function also requires an indirect jump.

Switch statement. Switch statements are usually implemented using jumps through a register which typically contains an address dependent on the switch value. Figure 1 reports the code emitted due to a switch statement in three different real-world cases.

From the point of view of a static binary translator, the most challenging indirect jumps are those produced by `switch` statements. In fact, their destination address is often computed at run-time and therefore it's never explicitly available in the code or data segments (see Figure 1a and 1c).

3. HARVESTING DATA AND CODE

We now introduce two methods to recover jump targets from program code and data. The proposed methods are designed to work in the context of a static binary translator supporting multiple input architectures and producing an intermediate representation agnostic with respect to the input. In practice, we employ the LLVM IR as the intermediate representation, although the concepts and methods are general, and could be employed in combination with other IRs, as long as the following assumptions hold. The intermediate representation is in SSA form, but the CPU state (and registers in particular) are mapped to global variables and, therefore, read and write operations are performed through load and store instructions. We define global variables representing a part of the CPU state a *CPU State Variable* (CSV). This is useful both for simplicity reasons and because it ensures that the semantics of a single instruction are self-contained. In fact, due to the iterative nature of our code discovery approach, we might need to split a basic block after its initial emission. It is worth noting that since hardware registers of the input architecture are mapped to global variables, ϕ-nodes are not needed in most cases. Note also that the CPU state is mapped to global variables for

```
cmp      r0, #240
addls    pc, pc, r0, lsl #2
b        21304
b        21320
b        21710
b        212fc
```

(a) $\text{pc} + 4 \cdot r0$, with $r0 < 240$

```
cmp      r3, #8
ldrls    pc, [pc, r3, lsl #2]
b        128c4
.word    0x1265c
.word    0x124ec
.word    0x12518
```

(b) $mem[\text{pc} + 4 \cdot r3]$, with $r3 < 8$

```
cmp cl,0x53
ja  471aa8
lea rax,[rip+0x3c9ca]
mov rcx, PTR [rax+rcx*4]
add rax,rcx
jmp rax
```

(c) base$+mem$[base$+4\cdot cl$], with $cl \leq 83$

Figure 1: Two ARM and an x86-64 real-world implementations of the `switch` statement. Figure 1a presents an implementation where the new address is written directly in the program counter (`pc`) and is computed as the current PC plus the switch value `r0` left shifted by two positions. The code in Figure 1b reads the jump target from an array of addresses (a *jump table*) stored in a constant pool close to the current program counter (the `.word` directives). The chosen address is determined using the switch value `r3` as an index. In Figure 1c we have an x86-64 `switch` implementation reading a value from base $+ 4 \cdot cl$ (where base $=$ pc $+$ `0x3c9ca` and cl is rcx's lowest byte), which is then added to base and used to perform an indirect jump. Note how, in all the examples, before computing the target address, the switch value is compared with a constant.

convenience: once the code is completely translated, and the runtime functions are linked in, the optimizer will be able to promote them to SSA values and let the register allocator map them to registers of the target architecture or, if required, spill them on the stack.

For maximum generality with respect to the input architecture, we also assume that call instructions have no special treatment, but are simply expanded to jump instructions. This also implies that all translated code will be emitted in a single function.

3.1 Global data harvesting

The program global data can provide useful information for jump target recovery. In fact, the `.rodata` and `.data` sections often contain function pointers, C++ virtual tables, or jump tables (see Figure 1b). Constant pools (see Section 2.2) can also be a source of pointers to basic blocks or functions, in case of jump instructions targeting addresses not reachable via an immediate offset added to the PC.

For this reason, the most straightforward approach to recover an initial set of jump targets consists in traversing byte-by-byte (or word-by-word) all the program segments looking for *code pointers*.

A *code pointer* is a byte sequence of the length of a pointer (32 or 64 bits) that, when interpreted using the appropriate endianess for the architecture, represents an address lying within an executable segment. If the ISA enforces an instruction alignment, e.g. 4 bytes for ARM, the resulting address must also be aligned to that value.

3.2 Simple Expression Tracker

Once an initial set of jump targets is available, the code at the corresponding addresses is translated. We therefore need to introduce an analysis aimed at harvesting jump targets from the translated code. The analysis we propose, called Simple Expression Tracker (SET), identifies all the store instructions and tracks in a step-by-step way how the value being stored is computed. The analysis proceeds as long as the operations composing the expression depend at most on a single non-constant operand. In fact, SET aims to collect the destination address of direct jumps and indirect jumps that materialize the destination address in multiple instructions. Consider the example of such an indirect jump in Section 2.2: SET will detect that the indirect jump (represented as a store to the CSV `t9`) targets `0x42d188`.

Algorithm 1 reports the working of the SET. When the SET processes a store instruction, it creates an empty stack, the *operations stack*, it inspects the value to be stored and it proceeds differently depending on its type.

If the stored SSA value of type i is the result of a binary operation (e.g., a subtraction) and its second operand is constant, we record on the stack the $\langle i, j\rangle$ pair, where i represents type of operation performed, and j the constant operand value. Then, we proceed considering the non-constant operand.

If the stored value is the result of a load operation from an unknown memory location (i.e., not from a CSV), we record the load operation on the stack and proceed to analyze how the address of the load operation is computed.

The analysis repeats the same operation with the newly considered operand, progressively growing the stack by pushing new operations. The process terminates when an instruction that cannot be handled is met (e.g., an addition with no constant operands) or when the operation to consider is a constant k, which means that a load from a constant address has been found or that both operands of the binary operation are constant.

In the latter case, it's possible to materialize a constant value: a variable n is initialized with k and the *operations stack* is traversed from top to bottom, updating the value of n by combining it with the operation registered on the stack. Therefore, for a load operation, if n is an address contained in a segment of the binary, its value is updated with the content of the pointed memory area, when this is statically available. For a binary operation, the new value will be the result of performing the operation i using n and j as operands.

Load/store handling. In addition to this, the SET explicitly handles loads from CSVs. A load from a CSV can be affected by multiple stores, therefore, to process them we employ a LIFO worklist. When a load from a certain CSV is met, the analysis proceed backwards, starting from the load instruction, looking for store instructions writing to that CSV and exploring recursively all the ancestor basic blocks until such a store instruction is found. For each found store instruction, a pair $\langle s, h\rangle$ is pushed on top of the worklist, where s is a reference to the store instruction, and h is an integer number representing the current height of the *operations stack*. The analysis proceeds by processing the element on the top of the worklist. When the top element is extracted from the worklist, the *operations stack* is cut to height h and the analysis proceeds from the value stored by s. This is necessary to restore the stack to its state when the work item was inserted into the worklist, discarding all the operations pushed on the stack while processing other work items.

The advantage of this approach lies in the fact that, by using a depth-first exploration, we can always reuse the lower part of the *operations stack*, without ever duplicating it, with the net effect of keeping its size in $O(n)$ of the number of instructions.

This analysis is very effective in collecting the simplest jump targets hidden in the code. More specifically, it can collect the destinations of direct jumps, indirect jumps with a constant destination materialized in a register (see Section 2.1) and also all the return addresses of call instructions. In fact, as per our previously stated assumption, the generated code doesn't have the concept of *call instruction*, but represents them as a simple write to the program counter preceded by an instruction storing the return address on the stack or in the link register. In both cases, since we track all the stores and not only those to the PC, our analysis is able to catch the return address. Figure 2 presents some examples of jump targets recovered from the code through the SET.

```
Data: A store instruction $s$
Result: A jump targets generator
Init stacks: $Ops$ (empty) and $WL$ ($\langle s, 0\rangle$);
while $WL$ is not empty do
    $c$, $h$ = pop($WL$);
    truncate $Ops$ to $h$ elements;
    $next$ = getStoredValue($c$);
    while $next$ is set do
        $i = next$;
        unset $next$;
        if $i$ is a binary operation $v \circ j$ then
            if $j$ is constant and $v$ is not then
                push $\langle i, j\rangle$ onto $Ops$;
                $next = v$;
        else if $i$ is a load instruction from $a$ then
            if $isCSV(a)$ then
                foreach $w$, previous store to $a$ do
                    push $\langle w$, getHeight($Ops$)$\rangle$ onto $WL$;
            else
                push $i$ onto $Ops$;
                $next = a$;
        else if $i$ is a constant value then
            $n$ = value($i$);
            foreach $\langle o, j\rangle$ in $Ops$, top to bottom do
                if $o$ is binary operation then
                    $n$ = apply($o$, $n$, $j$);
                else if $o$ is a load instruction then
                    $n$ = readFromSegment($n$);
            yield $n$;
```

Algorithm 1: The Simple Expression Tracker algorithm.

4. THE OSR ANALYSIS

Despite its effectiveness, SET is not able to collect jump targets due to switch statements such as those shown in Figure 1. In fact, in these cases, the jump target depends on a non-constant operand: the result of the expression evaluated by the switch statement. Therefore, we introduce a specialized data flow analysis whose goal is to try and represent each SSA value in the following form:

$$a + b \cdot x, \text{ with } \left\{ x : \begin{array}{c} c \le x \le d \\ x < c,\, x > d \end{array} \text{ and } x \text{ is } \begin{array}{c} \text{signed} \\ \text{unsigned} \end{array} \right\}$$

where: a is a constant base value; b is a constant scaling factor; and x is a reference to a *free* SSA value associated with a (possibly negated) range $[c, d]$ and a signedness (signed or unsigned).

We chose this form as the optimal tradeoff between complexity and expressive power to model how the destination address of a switch statement's indirect jump is computed. In particular, it's suitable to capture the jump targets represented in Figure 1 or part of it (e.g., $\text{pc} + 4 \cdot i$ with $i < 8$). Any increase in terms of expressive power would raise sensibly the complexity of the analysis and, as we will see, it wouldn't produce any benefit.

We define x, together with its constraints, as a *bounded value* (BV). We also define *offset shifted range* (OSR) as an instance of the above expression. We therefore call our analysis *OSR analysis* (OSRA).

A BV is always associated with an SSA value x which cannot be expressed in terms of an OSR relative to any other SSA value. In other terms, a BV represents a *free* SSA value associated to a range constraint. An example of such free SSA value might be the result of a *xor* operation, which exceeds the expressiveness of an OSR. The OSRA traverses all the program instructions and, where possible, associates them with an OSR.

In parallel, the analysis also tracks constraints that hold in a certain basic block in the form of BVs. To this end, the analysis processes comparison instructions and tracks their usage in conditional branch instructions. For example, if an instruction performs an unsigned comparison to check if an SSA value x is less than or equal to 7, OSRA will create a BV $\{x : 0 \le x \le 7, \text{unsigned}\}$ and will associate it with the comparison instruction. If the result of the comparison is then used in branch conditional instruction, the analysis will associate the BV with the basic block taken if the condition holds, and all the OSRs relative to x in this basic block will be affected.

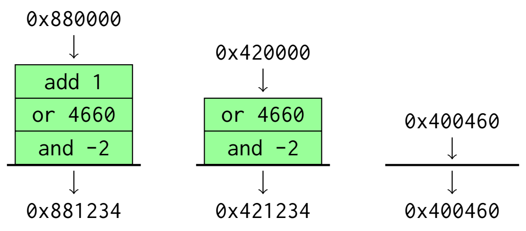
4.1 OSR tracking

Initially, no instruction is associated with an OSR. When OSRA is given an instruction i representing a binary operation, the number of non-constant operands is checked, and, as with SET, if more than one is present, it is ignored. Otherwise, the non-constant operand is considered. If it has an OSR, it is used as a base for the new OSR. If it doesn't have an OSR, a *basic OSR* is created. A *basic OSR* is an OSR with $a = 0$, $b = 1$ and x is set to a BV representing the non-constant operand. In both cases, the resulting OSR has to be updated according to semantics of the current instruction (see Table 1), and the constraints on the BV are updated for the new context. In fact, if the non-constant operand and the current instruction are in two distinct basic blocks, the constraints on a BV might be different. If i's basic block is not already associated with a BV for x, a new one is created without constraints (i.e., it is set to $\top$), otherwise, the existing one is used. Note that, unlike the SET, OSRA does not support all the possible instructions, but only the subset that can be handled considering the OSR expressive power. For example, the *rotate* and *xor* instructions cannot be handled.

It is important to understand that by cloning the OSR of the non-constant operands, the associated BV is also being propagated. This means we can have several instructions, possibly one using the result of the other, possibly on different execution paths, all expressed with respect to a single SSA value. This is particularly beneficial, since if OSRA is

```
    lui   $v0, 0x42
    ble   $a0, $t0, call ; Delay slot omitted
    lui   $v0, 0x88
    addi  $v0, 1
call:
    ori   $v0, 0x1234
    jal   $v0
```

(a) Input MIPS assembly



(b) Schematization of the SET jump target recovery

```
 store i32 0x420000, i32* @v0
 %1 = load i32, i32* @t0
 %2 = load i32, i32* @a0
 %3 = icmp slt i32 %1, %2
 br i1 %3, label %call, label %fallthrough

fallthrough:
 store i32 0x880000, i32* @v0
 %4 = load i32, i32* @v0
|%5 = add i32 %4, 1
 store i32 %5, @v0
 br label %call

call:
 %6 = load i32, i32* @v0
|%7 = or i32 %6, 0x1234
 store i32 0x400460, i32* @ra
|%8 = and i32 %7, -2
 store i32 %8, i32* @pc
 br label %dispatcher
```

(c) LLVM IR produced by the binary translator

Figure 2: Example of the SET algorithm. Figure 2a shows a MIPS assembly snippet of an indirect function call with two possible targets (`0x881234` and `0x441234`). In the example, three jump targets can be recovered: the return address being stored in the link register `ra` by the function call (`jal`) and the two possible destinations of the function call, stored in the `v0` register. In Figure 2c the LLVM IR produced by our binary translator is presented along with the two paths leading to the creation of a jump target: both start from a store to the program counter CSV, then, they split in the load `%6` and end in two distinct store of constant values. SET traverses these two paths and records in the *operations stack* all the instructions it meets, except for CSV-related load/stores (notice the vertical bars on the left of the recorded instructions), until an instruction where all the input operands are constant is met, i.e., the constant store instructions. At this point the *operation stack* is traversed from top to bottom executing the registered operations to compute the jump target. Figure 2b shows the state of the *operations stack* when the constant is met at the end of two paths, along with a zero-height stack due to the constant store in the link register CSV `@ra`.

Table 1: Effect of composing an OSR $a+b\cdot x$ with a constant k through a binary operator.

Op	Resulting OSR	Op	Resulting OSR
$+$	$(a+k)+x\cdot b$	$/$	$(a/k)+x\cdot(b/k)$
$-$	$(a-k)+x\cdot b$	$\ll$	$(a\cdot 2^k)+x\cdot(b\cdot 2^k)$
$\times$	$(a\cdot k)+x\cdot(b\cdot k)$	$\gg$	$(a/2^k)+x\cdot(b/2^k)$

able to verify that in a certain set of basic blocks an SSA value, or an OSR referring to it, is constrained in some way (e.g., has an upper bound), all the OSRs using it can benefit from this information directly.

4.2 BV tracking

A BV tracks, for a certain SSA value x, the lower and upper bound of the range within which x lies, possibly negated, and its *signedness*. By *signedness* we mean whether the SSA value represents a signed or unsigned integer, the *sign* itself is not tracked. The initial value of a BV is $\top$: it can assume any value and has an *unknown* signedness.

Each basic block is associated with a set of BVs which are known to hold for that basic block. Also certain instructions can be associated with BV, indicating their run-time result represents whether the associated BV (i.e., constraint) holds or not. This information becomes useful when the result of the instruction is used as the condition for a conditional branch instruction. In fact it is possible to state that in the basic block taken if the condition is true the constraint of the BV holds, while in the successor it does not.

To track BVs, OSRA considers three types of instructions: comparisons with constants, logical and/or instructions and conditional branches.

Comparisons and signedness. When a comparison with a constant k is met, the OSR associated with the non-constant operand is considered, or, if it doesn't have one, a *basic OSR* referred to the operand itself is created. The expression represented by the OSR is then compared with the appropriate comparison operator (e.g., signed greater than or equal) with the constant operand, obtaining a first-degree inequation.

$$a+b\cdot x \geq k \Longrightarrow x \geq \frac{k-a}{b}$$

The solution is then used as a constraint on x and a new BV is created and assigned to the comparison instruction.

Moreover, if the comparison is not simply a check for equality or inequality, it carries a signedness information, i.e. it can be signed or unsigned. This information is propagated to the BV corresponding to x associated to the basic block, which updates its signedness according to the finite-state machine in Figure 3. The signedness of a BV affects the maximum upper bound and the minimum lower bound, which are those of an `unsigned int` for an *unsigned* BV, those of a `signed int` for a *signed* BV and an intersection of the two in case of *inconsistent* signedness.

The signedness is particularly relevant for our purposes, since the lower bound of an unsigned BV is implicitly zero, and therefore with a single additional constraint (e.g., $x \leq 5$) we can limit the value of x in small range, which is desirable to the final aim of the analysis.

Logical operators. The second type of instruction handled by OSRA to track BVs are the logical *and* and *or* operators. If both instruction operands are associated to a BV referring to the same SSA value, the two constraints are merged according to an *and* or *or* policy depending on the

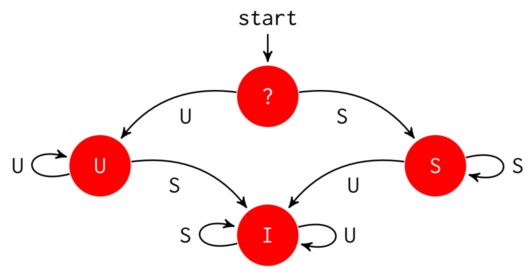


Figure 3: Finite-state machine representing the possible signedness state transitions of a BV. The `?`, `U`, `S` and `I` nodes represent respectively an *unknown* signedness, an *unsigned* value, a *signed* value and a value with an *inconsistent* signedness. The edges represent the transition performed when the value associated to BV is used with an unsigned (`U`) or signed (`S`) operation.

instruction. The merge policy considers the constraints as ranges (possibly negated) and combines them through the union operation (*or* merge policy) or the intersection operation (*and* merge policy), and generates a new constraint which is then associated with the instruction. However, the merge operation can fail, for instance if two positive disjoint ranges have to be combined with using the *or* merge policy, since the result exceeds the expressive power of the BV, which can represent at most a single positive range. In this case, the BV is set to $\perp$.

Conditional branches. The most important instruction type for tracking BVs are conditional branches, since they allow the analysis to state that a certain constraint, or its opposite, holds in a certain basic block. More specifically, when a conditional branch instruction is analyzed, if the SSA value used as a condition is associated with a BV, this BV is propagated to the first successor (the *true* branch) and its negated form is propagated to the second one (the *false* branch).

Therefore, each basic block is associated with a set of BVs obtained by propagation from its predecessors. Since a basic block might have multiple predecessors propagating different constraints, the BV considered to hold in a basic block is obtained as the result of a merge operation of the BVs coming from each predecessor using the *or* policy. Moreover, since a single predecessor might propagate a BV multiple times (a basic block might be analyzed more than once), OSRA explicitly registers which BV has been received from which predecessor. This way, the BV coming from a predecessor can be updated and it is possible to recompute without information loss the resulting BV for the basic block by *or*-merging all the BVs again. Note that if a predecessor does not provide a constraint for a certain SSA value, it is assumed to be unconstrained, and therefore the merge operation will produce a $\top$ value. Note also that in case a certain predecessor propagates a BV relative to a certain SSA value multiple times, the new constraint will be at least as strict as the previous one.

4.3 Load and store handling

To increase the effectiveness of the analysis, we also keep track of load and store instructions targeting CSVs. In particular, we have two objectives. First, we need to propagate OSRs and BVs being stored to a CSV to all the load instructions reading that value. For instance, the result of a compare instruction might be saved in a CSV, and therefore the associated constraint needs to be propagated to all the load instruction reached by that store. Second, even if the analysis cannot track what is being loaded, we want to be able to express the fact that two instructions loading the same CSV, among which a path exists without instructions writing to that CSV, are loading the same value.

To this end, when a store or load instruction using the CSV r is met, its OSR gets propagated, or, if it doesn't have an OSR, a new *basic OSR* is created referring to r. Propagation takes place by recursively exploring the subsequent instructions in the basic block and in its successors, looking for load instructions reading r, until a store to r is met. If a BV is associated to r, it is propagated too.

While propagating a load or a store, the analysis keeps track of which load instruction were affected by a the propagation in the *overtaken* set. If, while propagating a load or a store, a load instruction already associated with an OSR is met, a check on the OSR is performed: if its BV is part of the *overtaken* set, the propagation takes place and the existing OSR is overridden, otherwise it means the load instruction depends on multiple BVs. In the latter case, we do not have a merge policy and simply stop the propagation. The existing OSR is replaced with a self-referencing *basic OSR* identified with $\perp$, which will prevent any future propagation.

On the contrary, while propagating a BV, if a load instruction already associated with a BV referring to the same SSA value is found, the two constraints are merged using the *or* policy.

4.4 Integration with SET

As discussed, the primary aim of OSRA is to recover jump targets for a certain type of `switch` statements. However, while it provides useful information to this end, compared to the previously presented analysis, it presents some shortcomings: it cannot read data from memory segments present in the binary and can only handle a subset of all the possible binary operations. For this reason, we enhanced the SET to exploit the information provided by OSRA. The integration with OSRA affects two aspects: constant handling and materialization of OSRs.

For the former aspect, while describing the SET, we mentioned that it was able to handle operations with at most a single non-constant operand. Thanks to the OSRA we can expand the concept of *constant* to SSA values associated with an OSR whose BV is constrained to a single value (i.e., the lower and upper bounds match). This opens up for handling a slightly larger amount of situations.

The second, and most relevant, aspect is the OSR materialization. If, while building the *operations stack*, an instruction that cannot be handled is met, the analysis checks if an OSR is available for that instruction. If so, we compute $min = a + b \cdot c$ and $max = a + b \cdot d$. Then, all the operations on the *operations stack* are applied to them. If, in both cases, the result is a valid *code pointer*, then the OSR can be used to produce jump targets. Therefore, all the values that the OSR represents are generated, from min to max with a step size of b, and go through the operations in the *operations stack*, producing all the jump targets represented by the OSR.

In Figure 4 the code generated by an ARM compiler for a `switch` statement is exemplified and annotated with the information produced by OSRA. The example shows most of the feature of the analysis we discussed, such as propagation of stored values (`%2`$\rightarrow$`%4`) and merge of BVs coming

```
BB1:
  %1 = load i32, i32* @r1
  %2 = sub i32 %1, 4 ; [-4 + 1 * %1]
  store i32 %2, i32* @ZF
  %3 = icmp uge i32 %1, 4 ; (%1, u, 4, max)
  br i1 %3, label %BB2, label %BB3

BB2: ; (%1, u, 4, max) = <BB1, (%1, u, 4, max)>
  %4 = load i32, i32* @ZF ; [-4 + 1 * %1]
  %5 = icmp ne i32 %4, 0 ; (%1, u, 5, max)
  br i1 %5, label %exit, label %BB3

BB3: ; NOT (%1, u, 5, max) =
     ;         <BB2, (%1, u, 4, 4)>
     ;      || <BB1, NOT (%1, u, 4, max)>
  %6 = load i32, i32* @r1 ; [0 + 1 * %1]
  %7 = shl i32 %6, 2 ; [0 + 4 * %1]
  %8 = add i32 113372, %7 ; [113372 + 4 * %1]
  %9 = and i32 %8, -2
  store i32 %9, i32* @pc
```

Figure 4: Example of the LLVM IR generated by two ARM instructions: `cmp r1, #5; addls pc, pc, r1, lsl #2`. Comments indicate information produced by OSRA, in particular (x, s, c, d) represents a BV, $[a + b \cdot x]$ an OSR and $(BV) = \langle BB1, BV1\rangle \,||\, \langle BB2, BV2\rangle$ the BV associated to a basic block, obtained by *or*-merging BV1 (coming from BB1) and BV2 (coming from BB2).

from multiple predecessors (`BB3`). Note that `%8` holds the key information to obtain 5 jump targets, but, since OSRA does not handle logical and on OSRs (`%9`), the SET is necessary to let the information associated to `%8` reach the PC store.

4.5 Formalization of the DFA

In this section we sketch the formalization of the OSRA data flow analysis (DFA).

OSRA can be seen as a combination of two parallel data flow analysis: one for tracking OSRs and one for BVs. Both analysis are global (intraprocedural) DFAs with forward flow information propagation.

OSR tracking. The flow function of the former DFA associates each instruction with an OSR tuple $\langle x, a, b\rangle$, where x is the instruction in relation to which it is expressed, and a and b represents the offset and multiplier as seen in the previous sections. The DFA works on a lattice composed by the said tuples, where $\top$ represents no information associated to an instruction, $\bot$ an instruction that can only expressed in terms of itself, and all the intermediate values form a graph of SSA values where an edge from x to y represents the fact that y can be expressed in terms of x. The *confluence operation*, which is used while propagating an OSR from a CSV load/store instruction to a load instruction, can either update the value in relation to which the target OSR is expressed (see the discussion about the *overtaken* set in Section 4.3), or, if this is not possible, set the data flow information to $\bot$.

The DFA converges since the confluence operation can only move downward on the lattice. Note that a and b do not influence convergence since they are basically accessory information updated exclusively when the x is replaced. In practice this means the DFA considers only a single iteration of loops, which is an assumption that avoids the introduction of narrowing or widening operators [1], and is perfectly acceptable in our context, since compilers rarely, if ever, compute a jump target using a loop.

BV tracking. The flow function of the latter DFA associates each instruction with a BV tuple $\langle x, c, d, s, n\rangle$ representing the integer SSA value x constrained within the, possibly negated (n), range $[c, d]$, and having signedness s. This data flow information is propagated (possibly negated) through the edges of conditional branches, as previously discussed. The lattice it works on is composed by ranges on the variable x of decreasing size moving from $\top$ (any representable value) towards $\bot$ (not representable). This holds both for positive ranges and for negated ranges, which can be seen as the union of two ranges from the minimum representable value to $c - 1$ and from $d + 1$ to the maximum representable value.

The confluence operation takes place at the entry of a basic block and produces a new BV through the union of all the incoming BVs for a certain variable x. If the the result of the union of the ranges cannot be represented by a single (possibly negated) range, the BV is set to $\bot$.

The data flow converges since data flow information can only move downwards on the lattice, towards smaller ranges (or pairs of ranges, in the negated case).

Solution of the data flow problem. Both data flows are carried out in parallel since the former provides the latter with information to create useful constraints and, vice versa, the latter can create new constants (i.e. ranges where $c = d$) which open up for the creation of new OSRs.

A fixed-point solution is obtained by keeping a worklist, initially populated with all the generated instructions (in post-order), where, each time the data flow information associated to an instruction is updated, all its users are inserted back in the worklist.

5. EXPERIMENTAL RESULTS

All the presented techniques have been implemented in our static binary translator, REVAMB. REVAMB, which will be released with a Free Software license, consists in 6827 C++ SLOCs (as measured by the `sloccount` tool), and makes heavy use of QEMU, a dynamic binary translator, and LLVM, a popular compiler framework. The translator employs the QEMU's Tiny Code Generator to disassemble code from the input binary and produce an intermediate representation known as *tiny code*. This IR is then translated in LLVM IR, which is more suitable for optimization and recompilation.

Currently REVAMB is able to translate binaries compiled for a Linux-architecture combination, to a different Linux-architecture pair. To handle syscalls, we reuse the QEMU subsystem dedicated to this purpose. In fact, QEMU supports the *user mode*, which allows to run Linux binaries compiled for an architecture different from the host one by dynamically translating the code and handling syscalls in the appropriate way. The syscall handling component is then linked statically to the program generated by REVAMB.

The combination of QEMU and LLVM potentially allows us to support input binaries compiled for Alpha, ARM, CRIS, x86, MicroBlaze, MIPS, OpenRISC, PowerPC, RISC V, System Z, SuperH, SPARC and Unicore, and translate them for ARM, Hexagon, x86, MIPS, OpenRISC, PowerPC, RISC V, SystemZ, SPARC and XCore, in addition to their 64-bit counterparts (where applicable).

We adopted a development version of LLVM 3.8 and QEMU 2.5.0. For our experiments, we implemented support for 3 popular source ISA: MIPS, ARM and x86-64. This choice was guided by the attempt to test the various features an ISA can have, such as: endianess (MIPS is BE, the others are LE), register size (32 or 64 bits), CISC vs RISC designs, variable-length instruction encoding (x86-64) and delay slots (MIPS). For ease of testing, we chose as destination architecture x86-64 in all cases. All the architecture-specific code is managed by QEMU: REVAMB is completely platform agnostic. For our tests, the following toolchains have been employed: GCC 5.3.0 using uClibc for ARM and musl for MIPS, and GCC 4.9.2 with musl for x86-64.

Since REVAMB doesn't support dynamic linked binaries, all the tests applications were linked statically. Note that static binaries provide less information than dynamically linked executables, since the dynamic table and the dynamic symbols are absent. This also means that our tool handles the C standard library, which tends to be very large, include hand-written assembly and other sophisticated pieces of code which are not found in ordinary binaries. In summary, using static binaries, puts us in the most difficult setting.

The translation process. REVAMB translates the input binary in an iterative fashion. The translation starts from the entry point of the program and all the jump targets that have been found in the ELF segments, as described in Section 3.1. Once a whole basic block has been translated, direct jumps (i.e, constant stores to the PC CSV) and fall-through jump targets are automatically detected and added to the list of addresses to visit. When the code at all the known addresses has been consumed, the Simple Expression Tracker LLVM pass (see Section 3.2) is run and all the harvested jump targets are processed. SET is run repeatedly on the new code, until it doesn't produce any new jump target. At this point, the OSRA LLVM pass (see Section 4) is executed over the generated code and the collected jump targets are explored. The process is iterated until no more jump targets can be recovered: the generated LLVM IR is then considered complete and ready for optimization and compilation.

5.1 Functional testing

The first and foremost objective of REVAMB is to produce working binaries. Therefore, willing to asses the effectiveness of our approach, we took the `coreutils` project, a set of 104 popular command line utilities such as `ls`, `base64`, `md5sum` and many others, and translated its binaries. Then, we run the 567 tests in the `coreutils` test suite on the binaries translated by REVAMB with and without OSRA enabled. The translation process and the tests were run on several different machines with different characteristics. On a Linux-based machine with an AMD Opteron 8378 CPU and 32 GiB DDR3 RAM, the average translation time of an ARM program (305 kiB on average) was approximately 110 seconds.

Due to some limitations in syscall management, we expected some failures, in particular due to the absence of support of multithreading, forking and a couple of other syscalls. However, in this paper our main aim is to identify jump targets correctly. Therefore, the most relevant result is how many tests failed due to an unhandled jump target. Fixing the remaining issues is mostly a matter of engineering work in improving the integration with the QEMU's syscall translation subsystem, and lies outside the scope of this work.

Table 2 summarizes the results on the `coreutils` test suite. Enabling OSRA, the amount of passed test moves from the 51%/57%/85% of the total to 65%/82%/85%, on MIPS/ARM/x86-64 respectively. The difference is more sensible in non-VLE architectures, since `switch` statements are easily translated in the form presented in Figure 1a, while on x86-64 most of them are implemented using jump tables, which are easily caught by the global data harvesting pass described in Section 3.1.

Apart from the raw amount of passed tests, the key point to consider to evaluate the effectiveness of the analysis we developed, is the amount of programs part of the test suite failing due to an unidentified jump target. The "U" column in Table 2 shows how their number is very low even employing only SET (5 programs in MIPS and 3 in ARM), and reaches to 0 in all cases using OSRA, with the exception of MIPS. The failures in the MIPS case are due to code similar to the following:

```
lui      s3,0x40
bal      412120 ; Delay slot omitted
addiu    s3,s3,0x0a48
```

In this case, the SET wasn't able to catch the value being stored in `s3` because it is built in part before a function call and in part afterwards. A possible solution to this problem consists in detecting the function call and in making REVAMB aware of the calling convention, which would let it know that `s3` is a callee-saved register and it's therefore preserved across function calls.

Apart from this corner case, the absence of failures due to missing jump targets is a strong indicator of the effectiveness of combining the SET and OSRA analyses to recover jump targets.

5.2 Code coverage

Code coverage is another interesting feature to evaluate in its two aspects: actual code that has not been translated (*undertranslation*) and data translated as code (*overtranslation*). To understand how much code we correctly translated, we adopted as ground truth debug symbols emitted by the compiler and the assembler.

Note that, since our primary aim is to produce working binaries, we're mainly concerned with minimizing the *undertranslation* since, if a part of code has not been translated the resulting binary may fail at run-time due to an unexpected jump target.

The "Covered" column in Table 2 shows how our tool in average translates from 89% up to 95% of the input code, depending on the architecture. The next three columns divide the remaining code in three categories: code belonging to functions that have been linked in the binary but are not referenced anywhere, code composed exclusively by no-op instructions and other types of untranslated code.

The first category is due to a limitation of the classical ELF linking process, which is performed with a section granularity, unlike the Mach-O format which works at function level. This means that if a section of a certain translation unit contains three functions, but only one of them is actually used by the program, all of them will be linked into the final program. Performing manual inspection we discovered that most of these functions are part of the C standard library, and are usually interface functions, e.g. if the program uses the `printf` function also the `fprintf` function will be linked in, even if it's never used.

The second cause of *undertranslation* are sequences of instructions composed exclusively by NOPs. Compilers often emit them for code alignment purposes, but they are not supposed to be executed.

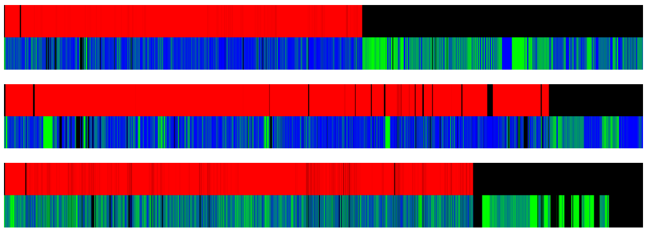


Figure 5: Visualization of the executable segment of three `coreutils` binaries: `df` for MIPS, `stty` for ARM, `dir` for x86-64. The actual code belonging to a function is reported in red in the top half of the image, the code generated by REVAMB due to a reliable jump target in blue, while the remaining translated code is in green.

Finally, in MIPS and ARM, a small amount of non-NOP instruction sequences remain untranslated. By performing manual inspection, in the MIPS case, we verified they are actually dead code mistakenly left by the compiler during an optimization pass aiming at exploiting MIPS delay slots. In the ARM case, the untranslated portions of code actually belong to constant pools that in certain cases are not marked as such in debugging symbols due to a limitation of the compiler.

In conclusion, our technique is able to correctly translate all the useful executable code on all the tested architectures.

For what concerns *overtranslation*, our results show how, in general, the amount of mistakenly translated code is not critical (less than 15% in all cases), which results in a slightly larger output binary. However, in some specific utilities, such as `printf`, the overtranslated part is significant. False positives in jump target collection might originate both from global data and the code itself.

A possible solution consists in classifying jump targets with respect to their *reliability*. To this end, we define a *reliable jump target* as a jump target obtained directly from a store to the PC (remember that we track all the *code pointers* being written to memory or to a register) and that do not represent a fall-through jump, i.e. a jump to PC immediately following the current one. *Reliable jump targets* are basically an indicator of the presence of actual control flow which is typically absent in data translated as code. Figure 5 reports in different colors the actual code and the code translated due to a reliable and a non-reliable jump target. As the figure shows, actual code has a higher density of reliable jump targets. This information might be exploited through machine learning algorithms to decide whether a portion of code is the result of overtranslation or not. Another option, under the assumption that sections containing code are always contiguous (which is the case in our test suite), is to devise a simple heuristic to identify the cutting point dividing code and data in the executable segment. However, since overtranslation is a secondary problem for our aims, we leave this aspect for future exploration.

5.3 Basic block size

A naïve approach to identify all the jump targets is to mark all the addresses in the executable segment as jump targets, but, as discussed in Section 2.1 this approach has several drawbacks.

Therefore, to assess how our solution performs compared to the naïve approach, we computed the average length of translated basic blocks, or, in other terms, the average distance in instructions among one jump target and the next one.

In Table 2 we can see the average length is well above the average length expected in the naïve approach, 1 or even less in case of VLE ISAs. Note that in the computation of the average length of a basic block, we ignored overtranslated portions of the binary, since optimizing code that will never be executed is not useful.

6. RELATED WORKS

Our work falls in the field of static binary translation, which is a subset of binary translation aiming at decoupling the translation of the binary from its execution. Binary translation has been studied for decades. Early efforts in the 1980s and 1990s focused on porting legacy code, or providing fast emulation platforms, with retargetability soon becoming a key concern [5, 3]. Applications of binary translation beyond the classic legacy code portability problem include binary instrumentation for security enforcement [15], reverse engineering, and de-obfuscation [13].

LLBT [12] is a static binary translator based on LLVM. Similar to REVAMB, this tool employs the LLVM IR to achieve retargetability in the static binary translator. The main difference between LLBT and REVAMB is that we use QEMU's Tiny Code Generator to perform the translation from binary to tiny code. Thus, REVAMB is inherently easier to maintain than LLBT, and requires much less work to add new source-target architecture pairs, as long as they are supported by QEMU and LLVM respectively. Regarding code discovery, LLBT focuses on ARM architectures and implements an ad-hoc mechanism to recover common patterns, limiting the generality of the approach. Their technique is very effective, as only 25% spurious regions are translated, but specific to the ARM/Thumb ISA mix.

UROBOROS [14] is a tool that focuses on producing disassembled code which can be reassembled without manual effort. Thus, it is very close to our own goals. The key challenge tackled by UROBOROS is to make the disassembled code relocatable. UROBOROS currently supports the disassembly of ELF binaries for the x86 and x64 architectures. However, UROBOROS is not meant to support recompilation to a different architecture, which is a goal of REVAMB.

CodeSurfer/x86 [1] is a tool based on IDAPro [9] and CodeSurfer [8], which implements *value-set analysis*, a form of data-flow analysis which tracks the contents of memory addresses, providing an over-approximation of the set of value that can be held in a memory location or register at each program point. A key difference with our tool is that we support multiple architectures instead of just x86, and that OSRA can provide more fine-grained and precise information about the tracked values.

Cifuentes and Emmerik [4] proposed a slicing analysis to identify jump targets from switch-case constructs, which is effective and portable, but does not deal with indirect calls, which causes under-translation in several cases. These effects are countered in their binary translation framework via an interpreter, which makes it necessary to use a dynamic rather than static binary translation technique.

7. CONCLUSIONS

We have introduced two methods to recover jump targets from binary code, Simple Expression Tracker and OSR Analysis, respectively targeting the values used in store instructions and the jump targets generated by switch constructs in the source code. The proposed analyses have been imple-

	Coverage						Tests (SET only)				Tests (OSRA)				Jump targets	
	Covered	Unused	NOPs	Other	Extra	IPB	S	P	F	U	S	P	F	U	Total	OSRA
MIPS	95.37%	4.61%	0.00%	0.02%	12.51%	5.17	169	228	170	5	160	266	141	3	1441834	49540
ARM	89.56%	8.91%	0.13%	1.40%	14.16%	3.98	140	220	207	3	143	350	74	0	807323	61190
x86-64	94.84%	4.70%	0.46%	0.00%	12.87%	4.22	163	343	61	0	163	343	61	0	1162082	23731

Table 2: Statistics concerning the translation of `coreutils` binaries. The "Coverage" column reports the average percentage of the original code that has been translated ("Covered") and that has been ignored because belongs to an unused function ("Unused"), it is composed exclusively of no-op instructions ("NOPs") or for other reasons ("Other"). The "Extra" column represents the amount of *overtranslated* code with respect to the actual code. "IPB" represents the average number of instructions per basic block. The "Tests" columns report the results of running the `coreutils` test suite using binaries produced by REVAMB without and with OSRA enabled. The amount of skipped (S), passed (P) and failed (F) tests is reported for each situation. A test is skipped if the test suite detects that the environment doesn't meet the conditions to run the test. The "U" column, highlighted in gray, represents the amount of `coreutils` programs failing due to a missing jump target. Finally, the total amount of jump targets collected over the whole test suite is reported, along with the number of those recovered thanks to OSRA.

mented in REVAMB, a static binary translation framework based on LLVM and QEMU. An experimental campaign on the `coreutils` binaries, compiled for ARM, MIPS and x86 targets, shows that the proposed analyses provide a coverage of the jump targets ranging from 89.56% to 95.37% depending on the target architecture, while limiting the average overtranslation to less than 15% in all cases. Furthermore, OSRA proves particularly effective on ARM binaries, increasing the number of tests successfully completed by more than 50%. In the future we plan to tackle more directly the *overtranslation* problem and, most importantly, to integrate in our framework function recognition to further improve our results.

8. REFERENCES

[1] G. Balakrishnan and T. 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] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Proc. of the 16th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 265–278, New York, NY, USA, 2011. ACM.

[3] M.-K. Chung and C.-M. Kyung. Improvement of compiled instruction set simulator by increasing flexibility and reducing compile time. In *Rapid System Prototyping, 2004. Proceedings. 15th IEEE International Workshop on*, pages 38–44, June 2004.

[4] C. Cifuentes and M. V. Emmerik. Recovery of Jump Table Case Statements from Binary Code. In *Proc. of the 7th Int. Workshop on Program Comprehension*, IWPC '99, pages 192–, Washington, DC, USA, 1999. IEEE Computer Society.

[5] C. Cifuentes and V. Malhotra. Binary translation: Static, dynamic, retargetable? In *Software Maintenance 1996, Proc., Int. Conf. on*, pages 340–349. IEEE, 1996.

[6] M. Damschen, H. Riebler, G. Vaz, and C. Plessl. Transparent Offloading of Computational Hotspots from Binary Code to Xeon Phi. In *Proc. of the 2015 Design, Automation & Test in Europe Conf. & Exhibition*, DATE '15, pages 1078–1083, San Jose, CA, USA, 2015. EDA Consortium.

[7] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing&Trade; Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-life Challenges. In *Proc. of the Int. Symp. on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 15–24, Washington, DC, USA, 2003. IEEE Computer Society.

[8] GrammaTech, Inc. CodeSurfer. http://bit.ly/1TGy7u2.

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

[10] C. Mendis, J. Bosboom, K. Wu, S. Kamil, J. Ragan-Kelley, S. Paris, Q. Zhao, and S. Amarasinghe. Helium: Lifting High-performance Stencil Kernels from Stripped x86 Binaries to Halide DSL Code. In *Proc. of the 36th ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI 2015, pages 391–402, New York, NY, USA, 2015. ACM.

[11] Santa Cruz Operation. System V Application Binary Interface, 2013. http://bit.ly/1qcy5xS.

[12] B.-Y. Shen, J.-Y. Chen, W.-C. Hsu, and W. 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.

[13] A. Wailly. Towards ultimate deobfuscation. Journée Sécurité Lille, Feb. 2015.

[14] S. Wang, P. Wang, and D. Wu. Reassembleable Disassembling. In *24th USENIX Security Symp. (USENIX Security 15)*, pages 627–642, Washington, D.C., Aug. 2015. USENIX Association.

[15] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar. A Platform for Secure Static Binary Instrumentation. In *Proc. of the 10th ACM SIGPLAN/SIGOPS Int. Conf. on Virtual Execution Environments*, VEE '14, pages 129–140, New York, NY, USA, 2014. ACM.