Recovering high-level conditions from binary programs*

Adel Djoudi[1], Sébastien Bardin[1] and Éric Goubault[2]

[1] CEA, LIST, Université Paris-Saclay, France
first.name@cea.fr
[2] Lix, École Polytechnique, CNRS, Université Paris-Saclay, 91128 Palaiseau, France
first.name@polytechnique.edu

Abstract. The need to get confidence in binary programs without access to their source code has pushed efforts forward to directly analyze executable programs. However, low-level programs lack high-level structures (such as types, control-flow graph, etc.), preventing the straightforward application of source-code analysis techniques. Especially, conditional jumps rely on low-level flag predicates, whereas they often encode high-level "natural" conditions on program variables. Most static analyzers are unable to infer any interesting information from these low-level conditions, leading to serious precision loss compared with source-level analysis. In this paper, we propose *template-based recovery*, an automatic approach for retrieving high-level predicates from their low-level flag versions. Especially, the technique is sound, efficient, platform-independent and it achieves very high ratio of recovery. This method allows more precise analyses and helps to understand machine encoding of conditionals rather than relying on error-prone human interpretation or (syntactic) pattern-based reasoning.

1 Introduction

Context. Static analysis [28] offers techniques for computing safe approximations of the set of values or behaviors arising at run-time when executing a program. Since the early 2000's, many successful source-code analysis techniques and tools have been proposed to check safety and security properties of industrial software [17,23,2].

Yet, there are many important situations *where the program must be analyzed directly at the level of executable code*, for example mobile code, off-the-shelf components, malware, etc. [13,2]. Such binary-level static analysis is highly challenging. Even on *managed code* (executable coming from standard high-level language such as C and compiled in a standard way), it is very hard to match the precision of an analysis performed at source-level mainly due to the lack of high-level information, such as types, variables, control-flow information or high-level conditions. The last decade has seen significant progress in binary-level static analysis, including precise control-flow graph recovery [7,24,22,14], formal intermediate representations (IR) [6,18,32,8], type and variables identification [26,5], or dedicated abstract domains [12,32,29,9,10]. Yet, the field remains highly challenging.

* Work partially funded by ANR, grant 12-INSE-0002.

Problem and challenges. We focus in this paper on *high-level condition recovery* from low-level flag conditions. Indeed, on most modern architectures, high-level conditions from the original program are translated at binary-level into low-level predicates operating on *flags*, i.e. boolean registers recording either high-level relationships between registers (==, $\leq$) or low-level facts such as occurrences of signed / unsigned overflows. High-level constructs such as `if, while, for`, etc. are no longer available. Hence, unless tracking relational information between program instructions, guard transfer functions of simple static analyzers will fail to refine propagated abstract states, because conditional jumps depend on flag values and not directly on registers and/or memory locations that set the corresponding flags.

Several solutions have been proposed in [11,13,32,25] to address low-level condition issues, yet they are either unsound and/or architecture dependent (patterns [13]), or intermediate-representation dependent (virtual flags [32]), or not generic enough (logic-based recovery [25,32,11,27]). We are looking for a solution which is both *sound*, *generic* (i.e. achieve a high recovery ratio in practice) and *independent from a given architecture or IR-encoding*.

Contributions. Our main contributions are the followings

- We present *template-based condition recovery* (Section 4), a new approach to high-level predicate recovery enjoying all the above desired properties: automatic, sound, architecture- / IR- independent, efficient and achieving a high recovery ratio in practice. The approach extends the logic-based method [25,32,11,27] and yields to significantly better recovery ratio. Compared to pattern-based methods [13], the technique is architecture independent and can infer high-level conditions from "non-regular" patterns – for example, optimized patterns introduced by compilers, cf. Section 7. Moreover, both template-based and pattern-based recovery can be fruitfully combined.
- We also address two questions closely related to the problem of high-level condition recovery and precise static analysis: the issue of ubiquitous data transfer between registers and memory, and the detection and positioning of widening points. We describe in Section 5 the two problems and present our solutions, namely a lightweight domain dedicated to equality propagation (on arbitrary lhs of the program) and a smart widening point positioning heuristic.
- The approach has been implemented in the new BINSEC/VA static analysis module of the BINSEC platform [19]. We detail the implementation (Section 6), and we describe several experimental evaluations (Section 7) assessing the recovery ability, efficiency and practical utility of our technique. Especially, template-based recovery yields only a small overhead and achieves a very high ratio of high-level condition recovery (between 89% to 95% in average). We also sketch potential applications to value analysis and deobfuscation.

Impact. Template-based recovery can help to adapt any formal analysis from source-level analysis to binary-level analysis, as illustrated in Section 7. It can also be useful for example for computer-aided reverse engineering, where it may help the reverser to quickly understand the real semantic of unusual flag manipulations, introduced either for optimization or obfuscation purposes (Section 7).

2 Motivation

2.1 The issue of low-level conditions in binary-level program analysis

The following example illustrates the problem of (low-level) flag encoding.

Example 1. Let us consider the following x86 instructions: `cmp x 100; je addr` encoding the high level condition `if (x = 100) then ...`. According to Intel documentation [21], this sequence reads as follows: instruction `cmp x 100` evaluates if equality `x = 100` holds and stores the (boolean) result into the specific flag `ZF` (other flags are updated, but they are not relevant here), then instruction `je` branches to address `addr` if `ZF` contains 1, to the next address otherwise. This can be expressed in the more abstract formalism of DBA (cf. Section 3) as follows (right column), where `ZF` is a 1-bit variable:

```
1: ZF := (x = 100);                  // x ↦ ⊤
2: if (ZF) then goto addr;           // x, ZF ↦ ⊤, [0, 1]
...                                  ...
addr: ...                            // x, ZF ↦ ⊤, [1, 1]
```

Let us now consider a standard interval analysis starting from $x \in \top$ at address `1`. The analysis will derive $ZF \in [0, 1]$ after the first instruction, then $ZF \in [1, 1]$ if the true branch is taken. However, nothing is derived for x (i.e. $x \in \top$) while it is straightforward that $x \in [100, 100]$. ⊠

Note that while this sort of low-level encoding can be found in C code, the situation is much problematic on binary code where low-level condition encoding is the norm.

Our goal is precisely to obtain source-level like propagation on binary code thanks to the recovery of high-level conditions.

2.2 Standard solutions and drawbacks

Logic-based solution. Several authors have independently proposed a similar solution to high-level condition recovery [25,32,11], that we call here *logic-based recovery*. The basic idea is to record relations into flag variables, to propagate these relations (taking operand updates into account) and to use them for refining the current state of the analysis once the flag value becomes 1 or 0. On the above example, flag propagation infers that `ZF` $\mapsto [x == 100]$ at line `addr`. Since `ZF` $\mapsto [1, 1]$, predicate `x==100` is also inferred, refining the abstract domain with `x` $\mapsto [100, 100]$, *which is exactly the result we are looking for.*

Yet, logic-based recovery is not always sufficient. The following example illustrates another conditional jump in x86 architecture where logic-based recovery fails.

Example 2. Let us consider the following x86 code sequence `cmp x y; jg addr`, encoding `if (x > y) then goto addr`. Internally, `jg` checks a combination of three flags updated by `cmp`, namely `ZF`, `OF` (overflow) and `SF` (sign).

```
OF := (x{31,31}=y{31,31})& (x{31,31}=(x-y){31,31});
SF := (x - y < 0);
ZF := (x - y = 0);
if (¬ZF ∧ (OF = SF)) then goto addr;
```

Here, relation propagation does not help, as the recovered low-level condition (below) is far from the *natural* high-level condition `x > y`. *Logic-based recovery is not able to identify most of high-level conditions coming from x86 flag encodings.*

```
if (¬(x-y = 0) ∧ ((x{31,31}=y{31,31}) & (x{31,31}=(x-y){31,31})) = (x-y<0)) then goto addr;
```

Pattern based solution. Balakrishnan *et al.* [13] suggests to pattern match the successions of comparisons and conditional jumps for deducing the corresponding high-level comparison. Standard x86 patterns are depicted in Table 1. *While precise on common cases, this approach is very architecture-specific. Hence, supporting several architectures is time-consuming. Moreover, it is very fragile w.r.t. non standard uses of flags and conditional branches, as found in optimization or obfuscation (cf. Section 7).* Note that ensuring soundness requires some care, e.g. taking properly into account flag / operand updates between the comparison and the conditional branch.

Table 1. High-level predicates for x86 conditional jumps [13]

	cmp x y / sub x y	cmp x y	sub x y	test x y	
	flag predicate[1]	predicate[2]	predicate[2]	flag predicate[3]	predicate[2]
ja, jnbe	$\neg CF \wedge \neg ZF$	$x >_u y$	$x' \neq 0$	$\neg ZF$	$x\&y \neq 0$
jae, jnb, jnc	$\neg CF$	$x \geq_u y$	true	true	true
jb, jnae, jc	CF	$x <_u y$	$x' \neq 0$	false	false
jbe, jna	$CF \vee ZF$	$x \leq_u y$	true	ZF	$x\&y = 0$
je, jz	ZF	$x = y$	$x' = 0$	ZF	$x\&y = 0$
jne, jnz	$\neg ZF$	$x \neq y$	$x' \neq 0$	$\neg ZF$	$x\&y \neq 0$
jg, jnle	$\neg ZF \wedge (OF = SF)$	$x > y$	$x' > 0$	$\neg ZF \wedge \neg SF$	$(x\&y \neq 0) \wedge$ $(x > 0 \vee y > 0)$
jge, jnl	$(OF = SF)$	$x \geq y$	true	$\neg SF$	$(x \geq 0 \vee y \geq 0)$
jl, jnge	$(OF \neq SF)$	$x < y$	$x' < 0$	SF	$(x < 0 \wedge y < 0)$
jle, jng	$ZF \vee (OF \neq SF)$	$x \leq y$	$true$	$ZF \vee SF$	$(x\&y = 0) \vee$ $(x < 0 \wedge y < 0)$

[1]: flag-level condition checked by the instruction. [2]: expected corresponding high-level condition
[3]: the same as [1], taking into account that `test` sets OF and CF to 0. (x' is defined by $x - y$)

Virtual flags. Sepp *et al.* [32] proposed to tackle the problem while translating machine instructions into their own Intermediate Representation RREIL. Flag calculations are translated, if possible, into arithmetic instructions. Typically a comparison between operands is assigned into a *virtual flag*. If a virtual flag is used later on, relational information between operands may be recovered and conveyed to numeric domains.

Example 3.
The succession of the two x86 instructions `cmp x y; jg addr;` seen in previous example is translated in RREIL as depicted on the right, with virtual flags `CForZF`, `SFxorOF` and `SFxorOForZF` representing combinations of concrete flags. At conditional branch, the test matching the flag is applied. Yet, the approach requires to add many virtual flags (updated at each instruction) dedicated to the targeted architecture and to ensure their consistency with the concrete flags, which can be tricky.

```
sub     t0:32, y:32, x:32
cmpltu  CF:1, y:32, x:32
cmpleu  CForZF:1, y:32, x:32
cmplts  SFxorOF:1, y:32, x:32
cmples  SFxorOForZF:1, y:32, x:32
cmpeq   ZF:1, y:32, x:32
cmplts  SF:1, t0:32, 0:32
xor     OF:1, SFxorOF:1, SF:1
brc     SFxorOF:1, addr:32
```

Summary. State of the art solutions are summarized in Fig 1, together with the template-based recovery method described latter in Section 4. *Our approach extends the logic-based method with more powerful recovery ability, while still being architecture and IR-encoding independent. Moreover, virtual flags and patterns, if available and soundly implemented, can complement template-based recovery in a fruitful way.*

Approach	archi. independent	IR encoding independent	Sound	Complete enough [1]	
				standard [1]	non-standard [1]
Patterns	✗	✓	✓ / ✗ [2]	✓	✗
Virtual flags	✓	✗	✓ / ✗ [3]	✓	✗
Logic-based	✓	✓	✓	✗	✗
Template-based	✓	✓	✓	✓	✓

[1]: does the technique achieve a large ratio of condition recovery in practice? We distinguish between standard flag encodings and non-standard ones (cf Section 7).
[2]: need to ensure that no flag / operand update happens between comparison and branching
[3]: need to ensure at each program step the coherence between virtual flags and concrete flags

Fig. 1. comparison of high-level predicate recovery approaches

3 Background

Our approach is based on abstract interpretation [15,16], a theory explaining how to link a very precise (but generally uncomputable) *concrete semantics* to its *sound approximation*, referred to as *abstract semantics*. This section first defines the syntax and concrete semantics of our Intermediate Representation, then a few notations of abstract interpretation.

DBA and concrete semantics. Automatic analysis of executables requires tools to abstract from the instruction set of each individual architecture by using an intermediate representation (IR). We rely on Dynamic Bit-vector Automata (DBA) [6], a generic, side-effect free and concise formal model for low-level programs, whose **syntax** is presented in Figure 2. DBA program manipulates a finite set of global variables ranging over

fixed-size bit-vectors (registers) and an array of bit-vectors of size 8 (memory). All bit-vector sizes are statically known. Conditions are bit-vectors of size 1. *Instructions* mostly include assignments and (static/conditional/dynamic) jumps, while *expressions* are built on standard bit-vector operators (bit-wise logical operations, shift, size restriction `e{i..j}` and extension `ext(e,n)`, concatenation `::`, (un-)signed machine arithmetic – unsigned operators are denoted with $\__u$) and memory accesses `@(e)`. A DBA program is a map from (code) addresses (i.e. bit-vectors of size `addr_size`) to DBA instructions, together with an initial address. In the following, the set of variables (resp. expressions) is denoted `Var` (resp. `Expr`).

DBA are given a standard imperative **semantic**. A *concrete state* (or environment) of a program is a map $\rho \in \mathbb{BV}^{var^+}$ assigning a bit-vector value from the set $\mathbb{BV}$ to each variable and memory location (denoted var^+). Expressions evaluate over bit-vectors. The semantics of an expression e in the concrete state ρ is denoted by $eval(e)_\rho$. In case an expression has no variable nor memory access, its semantic is given by $eval(e)_\emptyset$, simply denoted $eval(e)$. Assignments and conditions are given the standard semantic. A static jump `goto addr` branches to (the instruction at) address `addr`, while a dynamic jump `goto e` branches to address $eval(e)_\rho$.

From a modeling point of view, a single machine instruction is decoded into a block of DBA instructions - including intermediate computations and temporary variables. Floating-point arithmetic, multi-thread and self-modification are currently out of scope of DBA.

Instructions	Expressions
- `lhs := rhs, goto addr`	- `e{i .. j}`, $\texttt{ext}_{u,s}$`(e,n)`, `e :: e`
- `goto addr`	- `@(e)`
- `goto e`	- `e` $\{+, -, \times, /_{u,s}, \%_{u,s}\}$ `e`
- `ite(cond)? goto addr : goto addr'`	- `e` $\{<_{u,s}, \leq_{u,s}, =, \neq, \geq_{u,s}, >_{u,s}\}$ `e`
- `stop`	- `e` $\{\wedge, \vee, \oplus, <<, >>_{u,s}\}$ `e, !e`

Fig. 2. DBA instructions

Abstract interpretation. Abstract interpretation-based analyses [15,16] rely on an abstract domain D, whose computable elements model a set of concrete states at a given program point. Such abstract domains must provide the abstract counterparts of the concrete (set) operations over $(\mathcal{P}(\mathbb{BV}^{var^+}))^{\mathbb{N}}$: a partial order $\sqsubseteq_D$ over abstract states; a monotone concretization function γ_D from D to $\mathcal{P}(\mathbb{BV}^{var^+})$; greatest and smallest elements $\top_D$ and $\bot_D$, s.t. $\gamma_D(\top_D) = \mathbb{BV}^{var^+}$ and $\gamma_D(\bot_D) = \emptyset$; sound *over-approximations* join $\sqcup_D$ and meet $\sqcap_D$ of the union and intersection of concrete states, i.e. $\gamma_D(d_1) \cup \gamma_D(d_1) \subseteq \gamma_D(d_1 \sqcup_D d_2)$ and $\gamma_D(d_1) \cap \gamma_D(d_1) \subseteq \gamma_D(d_1 \sqcap_D d_2)$; sound abstract transfer functions $[\![i]\!]^\#_D$ from D to D that *over-approximate* the concrete semantics, i.e. $[\![i]\!](\gamma_D(d)) \subseteq \gamma_D([\![i]\!]^\#_D(d))$. The key property in abstract interpretation-based software verification is *soundness*, which ensures that each step in the abstract overapproximates all corresponding possible concrete steps.

4 Template-based recovery

4.1 Principles

We start from the idea of logic-based recovery and flag propagation. The issue here is that the high-level conditional expression may be too complex to be dealt with by basic non relational abstract domains, and that brute substitution of predicates in a non-trivial flag predicate often results in a complex low-level predicate, possibly hiding a simple predicate (cf. Example 2). *Template-based recovery* complements logic-based recovery with a *normalization* step for simplifying the current flag expression into a high-level form. It relies on **two key ideas**:

- first, there is only a finite set of high-level condition patterns we are interested in – built on $>_u, >, <_u, <, \geq_u, \geq, \leq_u, \leq, =, \neq$, with only two operands – since we consider on three-address instruction sets;
- second, equivalence between a high-level condition candidate and a given low-level condition can be checked by a SMT solver (in the theory of bit-vectors and arrays) – the check should be very efficient as the formula is expected to be very small.

Our approach works as follows. We first retrieve a set of potential operands from the low-level condition under analysis. A potential operand `x` must be either a variable, a memory access, or a restriction of a variable or memory access, i.e.

$$\texttt{x} \in \{\texttt{v}, \texttt{@[e]}, \texttt{v}_{\{i,j\}}, \texttt{@[e]}_{\{i,j\}}, \texttt{c} \mid \texttt{v} \in \texttt{Var}, \texttt{e} \in \texttt{Expr}, \texttt{c} \in \mathbb{BV}, \texttt{j>i}\}$$

Given a low-level condition `cond`, once the potential operands `x` and `y` are selected, we try to assert the equivalence of `cond` with the following high-level candidates:

$\texttt{cond} \Leftrightarrow \texttt{x} >_u \texttt{y}$	$\texttt{cond} \Leftrightarrow \texttt{x} <_u \texttt{y}$	$\texttt{cond} \Leftrightarrow \texttt{x} \geq_u \texttt{y}$	$\texttt{cond} \Leftrightarrow \texttt{x} \leq_u \texttt{y}$
$\texttt{cond} \Leftrightarrow \texttt{x} > \texttt{y}$	$\texttt{cond} \Leftrightarrow \texttt{x} < \texttt{y}$	$\texttt{cond} \Leftrightarrow \texttt{x} \geq \texttt{y}$	$\texttt{cond} \Leftrightarrow \texttt{x} \leq \texttt{y}$
$\texttt{cond} \Leftrightarrow \texttt{x} = \texttt{y}$	$\texttt{cond} \Leftrightarrow \texttt{x} \neq \texttt{y}$	s.t. $\texttt{x}, \texttt{y} \in_{syntax} \texttt{cond}$	

If an equivalence is found with candidate `cond'`, then `cond'` is used instead of `cond` during the abstract fixpoint computation. Otherwise, recovery fails and the abstract computation goes on with `cond`, following the logic-based approach.

4.2 Formalization

We consider an abstract interpretation framework with an abstract domain $F^\#$ associating to each flag an expression (elements $\texttt{f}: Flag \rightarrow Expr$), alongside a numerical non-relational abstract domain $A^\#$ lifted to program variables and memory locations $D^\#$ (elements $\texttt{d}: var^+ \rightarrow A^\#$), with its evaluation operator $(\!|.|\!) : Expr \rightarrow D^\# \rightarrow A^\#$ and condition propagation $assume : D^\# \rightarrow Expr \rightarrow D^\#$.

Our full abstract transfer function is the relation $. \rightarrow^\# . : (D^\# \times F^\# \times \mathbb{A}ddress) \rightarrow (D^\# \times F^\# \times \mathbb{A}ddress)$, from abstract states to new abstract states, described in Figure 3, where $\texttt{f} \in F^\#$, $\texttt{d} \in D^\#$ and $\texttt{l} \in \mathbb{A}ddress$. The syntax $\texttt{s}[\cdot \mapsto \cdot]$ denotes the state obtained by updating part of state `s` with a new abstract value. The flag abstract state (second component of an abstract state) is updated only at assignments, used at conditional

jumps and merely propagated through other instructions. If any operand of the flag expression $\mathtt{f}(\text{flag})$ is potentially affected by an assignment, either because one of its subterm is directly modified or because of potential memory aliasing, then $\mathtt{f}(\text{flag})$ is reset to $\top$. $\preceq$ denotes syntactic subterm, $\simeq$ denotes potential memory aliasing. Function normalize : $Expr \rightarrow F^{\#} \rightarrow Expr$ tries to recover a high-level condition from an expression $\mathtt{e}$. If high-level condition recovery fails, $\mathtt{e}$ is left unchanged. When control flow recombines after a conditional block or loop, abstract states are joined.

$$\frac{[\![flag := e; l']\!]}{\left(\mathtt{d}, \mathtt{f}, l\right) \rightarrow^{\#} \left(\mathtt{d}[flag \mapsto (\!|e|\!)(\mathtt{d})],\ \mathtt{f}[flag \mapsto e], l'\right)}$$

$$\frac{[\![v := e; l']\!] \quad v \preceq \mathtt{f}(flag_1) \quad ... \quad v \preceq \mathtt{f}(flag_n)}{\left(\mathtt{d}, \mathtt{f}, l\right) \rightarrow^{\#} \left(\mathtt{d}[v \mapsto (\!|e|\!)(\mathtt{d})],\ \mathtt{f}[flag_1, ..., flag_n \mapsto \top, ..., \top], l'\right)}$$

$$\frac{[\![@(e_1, c) := e_2; l']\!] \quad @(e_1, c) \simeq \mathtt{f}(flag_1) \quad ... \quad @(e_1, c) \simeq \mathtt{f}(flag_n)}{\left(\mathtt{d}, \mathtt{f}, l\right) \rightarrow^{\#} \left(\mathtt{d}[((\!|e_1|\!)(\mathtt{d}), c) \mapsto (\!|e_2|\!)(\mathtt{d})],\ \mathtt{f}[flag_1, ..., flag_n \mapsto \top, ..., \top], l'\right)}$$

$$\frac{[\![ite(e)?\, l_1 : l_2]\!] \quad e' = \text{normalize}(e)(\mathtt{f}) \quad \gamma((\!|e'|\!)(\mathtt{d})) \neq \{0\}}{\left(\mathtt{d}, \mathtt{f}, l\right) \rightarrow^{\#} \left(assume(\mathtt{d})(e'),\ \mathtt{f},\ l_1\right)}$$

$$\frac{[\![ite(e)?\, l_1 : l_2]\!] \quad e' = \text{normalize}(e)(\mathtt{f}) \quad 0 \in \gamma((\!|e'|\!)(\mathtt{d}))}{\left(\mathtt{d}, \mathtt{f}, l\right) \rightarrow^{\#} \left(assume(\mathtt{d})(!e'),\ \mathtt{f},\ l_2\right)}$$

$$\text{normalize}(e)(\mathtt{f}) \triangleq \begin{cases} t_1 \odot t_2 & \text{if } \phi(t_1, t_2) \Leftrightarrow t_1 \odot t_2 \\ & \text{s.t. } \phi(t_1, t_2) = e[\mathtt{f}(flag_1)/flag_1, ...; \mathtt{f}(flag_2)/flag_2], \\ & t_1, t_2 \preceq e[\mathtt{f}(flag_1)/flag_1, ...; \mathtt{f}(flag_2)/flag_2] \\ e & \text{otherwise} \end{cases}$$

$$x \preceq e \triangleq \begin{cases} true & \text{if } e \in \{x,\ x\{i,j\}\} \\ x \preceq e' & \text{if } e \in \{@(e', c),\ e'\{i,j\},\ !e',\ ext_{u,s}(e', n)\} \\ x \preceq e_1 \vee x \preceq e_2 & \text{if } e \in \{e_1 \odot e_2 \mid \odot \text{ is a binary operator}\} \end{cases}$$

$$x \simeq e \triangleq \begin{cases} (\!|e_1 + [0, c_1]|\!)(\mathtt{d}) \sqcap^{\#}_{\mathtt{d}} \\ (\!|e_2 + [0, c_2]|\!)(\mathtt{d}) \neq \bot_{\mathtt{d}} & \text{if } (x = @(e_1, c_1) \wedge y = @(e_2, c_2)) \\ x \simeq e' & \text{if } e \in \{@(e', c),\ e'\{i,j\},\ !e',\ ext_{u,s}(e', n)\} \\ x \simeq e_1 \vee x \simeq e_2 & \text{if } e \in \{e_1 \odot e_2 \mid \odot \text{ is a binary operator}\} \\ false & \text{otherwise} \end{cases}$$

$$\mathtt{f}_1 \sqcup^{\#}_{\mathtt{f}} \mathtt{f}_2 = \mathtt{f} \qquad \text{s.t. foreach } flag \text{ in } \mathtt{f}, \begin{cases} \mathtt{f}(flag) = e & \text{if } \mathtt{f}_1(flag) = \mathtt{f}_2(flag) = e \\ \mathtt{f}(flag) = \top & \text{if } \mathtt{f}_1(flag) \neq \mathtt{f}_2(flag) \end{cases}$$

Fig. 3. Abstract propagation of flags abstract domain

Theorem 1 (soundness). *The template based solution is sound i.e. if $\phi(t_1, t_2)$ is a flag predicate involving at least two terms t_1 and t_2 at address a, s.t. the template based solution asserts that $\phi(t_1, t_2) \Leftrightarrow t_1 \odot t_2$, then for each execution trace of the program the assertion holds at address a.*

4.3 Optimizations

Repeated calls to the SMT solver may raise efficiency issues. We propose two optimizations in order to mitigate this problem.

Optimization 1: Normalization cache. Each time a flag conditional is met at address `a`, the low-level condition is saved in the cache at address `a` together with the retrieved high-level condition. If the same condition at the same address `a` is met later in the analysis, then the saved high-level condition can be safely reused.

Optimization 2: Templates filtering. The order in which the templates are checked directly affects the efficiency of high-level predicate recovery. The problem is all the more important that the number of checked templates is higher. If the number of potential templates is reduced to one or two, the issue will be largely mitigated.

The idea behind *template filtering* is that many templates can be cheaply discarded by comparing the evaluation of the low-level condition to the template evaluation on a set of well-chosen values.

We denote by `cond[x/t]` the condition `cond` where each occurrence of syntactic term `t` is replaced by another syntactic term `x`. If $\texttt{op}_1$ and $\texttt{op}_2$ are non-constant operands syntactically appearing in condition `cond`, we can generate conditions $\texttt{cond}_1$, $\texttt{cond}_2$, $\texttt{cond}_3$ and $\texttt{cond}_4$:

$$\texttt{cond}_1 \triangleq \texttt{cond[0/op}_1\texttt{][0/op}_2\texttt{]}$$
$$\texttt{cond}_2 \triangleq \texttt{cond[0/op}_1\texttt{][1/op}_2\texttt{]}$$
$$\texttt{cond}_3 \triangleq \texttt{cond[1/op}_1\texttt{][0/op}_2\texttt{]}$$
$$\texttt{cond}_4 \triangleq \texttt{cond[0/op}_1\texttt{][max}_u\texttt{/op}_2\texttt{]}$$

The resulting four conditions will be evaluated in order to discard irrelevant templates. The intuition behind this set of values is that we need to distinguish between symmetric and antisymmetric operators ($(0,0)$), between the direction for antisymmetric operators ($(0,1)$), between signed and unsigned comparisons ($(0, max_u)$), and finally we have to distinguish between a strict comparison and a disequality ($(1,0)$ together with $(0,1)$).

Discarding templates according to conditions evaluation is given by the following consecutive tests. $\texttt{cond}_3$ is a special case requested when $eval(\texttt{cond}_2) = true$. Here: if $eval(\texttt{cond}_2) = eval(\texttt{cond}_3)$ then keep only template $\neq$, else remove $\neq$.

If $eval(\texttt{cond}_1) = false$ then templates ($\texttt{op}_1$ $\{=, \leq_u, \geq_u, \leq, \geq\}$ $\texttt{op}_2$) are discarded
else templates ($\texttt{op}_1$ $\{\neq, <_u, >_u, <, >\}$ $\texttt{op}_2$) are discarded
If $eval(\texttt{cond}_2) = false$ then templates ($\texttt{op}_1$ $\{<, <_u, \leq, \leq_u, \neq\}$ $\texttt{op}_2$) are discarded
else templates ($\texttt{op}_1$ $\{>, >_u, \geq, \geq_u, =\}$ $\texttt{op}_2$) are discarded
If $eval(\texttt{cond}_4) = false$ then templates ($\texttt{op}_1$ $\{>, <_u, \geq, \leq_u, \neq\}$ $\texttt{op}_2$) are discarded
else templates ($\texttt{op}_1$ $\{<, >_u, \leq, \geq_u, =\}$ $\texttt{op}_2$) are discarded

Whatever the low-level condition is, with only four tests we can eliminate all template candidates but one, which is then passed to the solver.

Example 4. Let us consider an arbitrary low-level condition `cond`, with two operands `x` and `y`. We compute cond_1 to cond_4 as defined before. Let us imagine that: $\text{cond}_1 = 0$, $\text{cond}_2 = 1$, $\text{cond}_3 = 0$, $\text{cond}_4 = 1$. Then the only remaining possible template is $\texttt{x} <_u \texttt{y}$.

$eval(\texttt{cond}_1) = false$ then templates (x $\{=, \leq_u, \geq_u, \leq, \geq\}$ y) are discarded
$eval(\texttt{cond}_2) = true$ then templates (x $\{>, >_u\}$ y) are discarded
$eval(\texttt{cond}_3) = false$ then template (x $\{\neq\}$ y) is discarded
$eval(\texttt{cond}_4) = true$ then template (x $\{<\}$ y) is discarded

$\boxtimes$

Similarly, if $\texttt{op}_1$ is already a constant value c then $\texttt{cond}_1$, $\texttt{cond}_2$, $\texttt{cond}_3$ and $\texttt{cond}_4$ are defined as follows:

$$\texttt{cond}_1 \triangleq \texttt{cond[c/op}_2\texttt{]}$$

$$\texttt{cond}_2 \triangleq \begin{cases} \texttt{cond[max}_s\texttt{/op}_2\texttt{]} & \text{if c} <_u \text{max}_s \\ \texttt{cond[max}_u\texttt{/op}_2\texttt{]} & \text{if c} >_u \text{max}_s \end{cases}$$

$$\texttt{cond}_3 \triangleq \begin{cases} \texttt{cond[0/op}_2\texttt{]} & \text{if } 0 < \text{c} <_u \text{max}_s \\ \texttt{cond[max}_s\texttt{/op}_2\texttt{]} & \text{if c} >_u \text{max}_s \end{cases}$$

$$\texttt{cond}_4 \triangleq \begin{cases} \texttt{cond[max}_u\texttt{/op}_2\texttt{]} & \text{if c} <_u \text{max}_s \\ \neg\texttt{cond[max}_s\texttt{/op}_2\texttt{]} & \text{if c} >_u \text{max}_s \end{cases}$$

5 Other issues related to the precise handling of conditions

We describe two situations closely related to low-level conditions that may yield to significant precision loss, even in the presence of high-level condition recovery, together with possible mitigation.

5.1 Ubiquitous data moves between memory and registers

Problem. Architecture-specific constraints may blur the encoding of high-level constructs. Typically, like the majority of x86 instructions, the `cmp` instruction allows at most one memory operand. So, in order to compare contents of two memory locations, we need first to load at least one of them into a register, then perform the comparison. Hence ubiquitous data move from memory (stack) to registers. Example 5 illustrates that a low-level analysis unable to track relational information through data manipulation will infer domain reduction on the compared registers (*which do not matter*), but not on the memory contents themselves (*which matter*).

Example 5. This example shows that even when a natural high level condition is available, a standard analysis may still miss obvious information.

```
1: eax := @[100];                    // @[100] ↦ [0, 130]
2: if (eax < 4) then goto addr;      // eax, @[100] ↦ [0,130],[0,130]
...                                  ...
addr: ...                            // eax, @[100] ↦ [0,3],[0,130]
```

Starting from `@[100]` $\in [0, 130]$, a static analysis with non-relation abstract domain will infer that both `eax` and `@[100]` range over the interval $[0, 130]$ after the first instruction. Yet, at address `addr`, the computed abstract state will only express that `@[100]` $\in [0, 130]$, while actually `@[100]` $\in [0, 3]$. ⊠

Solution. We propose to enrich the propagated abstract state with a *lightweight relational abstract domain* keeping track of *equalities between arbitrary* lhs (expressions of the form x or $@[e]$) syntactically present in the program. We propose to use an abstract domain of the form $\mathcal{P}^{\#} \triangleq \{\mathcal{C}(x) \mid x \in Lhs\}$ that builds a set of equivalence classes $\mathcal{C}(x) \triangleq \{y \in Lhs \mid x = y\}$. The two key points are (1) the trade-off between efficiency and precision (actually, we lose information in an aggressive manner, keeping only obvious equalities), and (2) the ability to refine the non-relational domains of all lhs of an equivalence class when one is refined by a comparison (here, we attach a domain to each class, refined each time a variable of the class is refined, and queried when a variable of the class is evaluated). On Example 5, the technique infers that `@[100] == eax` holds at the beginning of line `2`, allowing to refine `@[100]` to $[0, 3]$ at line `addr`.

Our implementation relies on a combination of union-find structure and maps, allowing efficient join and widening in $\mathcal{O}(n.ln(n))$ time, with n the number of lhs.

5.2 Widening point positioning

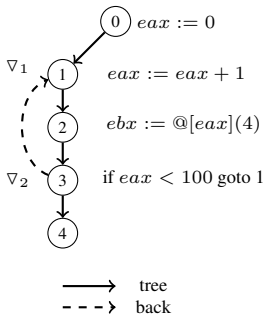
Problem. *Widening* [15] is the standard approach to ensure termination of loop treatment. Basically, widening is a kind of join operation satisfying the *non-ascending chain* property. Termination is ensured if each loop contains a widening point. In high-level programs, such widening points are easily deduced from the loop structure. However, binary programs lack such information. While we need to ensure that every cycle in the program control flow contains at least one widening point, there may have several positionings, all ensuring termination but with significant difference in precision, as illustrated in Example 6. Finding an optimal set of widening points is NP-complete [1].

Example 6. This example illustrates the effect of the widening point position on the precision of the final invariant, with widening points $\triangledown_1$ and $\triangledown_2$.

Program	abstract states (with $\triangledown_1$)	abstract states (with $\triangledown_2$)
`0: eax:= 0`	`0: eax` $\mapsto [0, 0]$	`0: eax` $\mapsto [0, 0]$
`1: eax:= eax + 1`	`1: eax` $\mapsto [min, max]$	`1: eax` $\mapsto [0, 99]$
`2: ebx:= @[eax](4)`	`2: eax` $\mapsto [min, max]$	`2: eax` $\mapsto [0, 100]$
`3: if(eax<100) goto 1`	`3: eax` $\mapsto [min, max]$	`3: eax` $\mapsto [0, max]$
`4:`	`4: eax` $\mapsto [100, max]$	`4: eax` $\mapsto [100, max]$

⊠

Solution. We rely on a depth-first numbering of the CFG nodes for the identification of widening points. Actually, the set of back edges of the depth-first search tree is an admissible set of widening points, as it ensures that each loop in the CFG features at least one widening point. Yet, considering precision, it remains an important decision to make: given a *back* edge `a` $\rightarrow$ `b`, which node should be taken at widening point? *Our solution is that a widening point should be positioned at the beginning of a conditional jump (∇_2 here), so that the guarded transfer function can refine the widened abstract state.*



6 Implementation

We have implemented the approaches described so far in BINSEC [19,20]. The platform is based on DBA [6,19] and is composed of the following main modules: loading and decoding (ELF and PE, x86 architecture), disassembly and heavy simplification of the resulting IR [19], DBA simulation, dynamic symbolic execution [20], and an ongoing static analysis module BINSEC/VA. BINSEC is implemented in OCaml. We describe here the static analysis module BINSEC/VA. It will be available in open-source at `http://binsec.gforge.inria.fr/tools.html`.

BINSEC/VA provides a generic fixpoint computation for abstract domains given as lattices, allowing one to quickly prototype binary-level analyzers. The main features are the following. **Value domain:** Since overflow and wrapping (between signed and unsigned representations) must be taken into account at binary-level, we keep track of both signed and unsigned values of each lhs, abstracted as a pair of intervals. Then, each component is reduced according to the other one. This *dual interval representation* is very simple (w.r.t. wrapped intervals [29]), and yet it still prevents the most common cases of precision loss. Typically, considering the following dual representation $d^{\#} \triangleq ([254, 255]_u, [-2, -1]_s)$, then $d^{\#}+1$ evaluates to $([0, 255]_u, [-1, 0]_s)$: here the unsigned part loses all precision, while the signed part remains precise. **Memory domain:** Value domains are lifted to byte-precise memory domains in a standard way, following for example [23] (in a simpler manner). **Other domains:** The analysis also provides flag propagation with template-based recovery and an equality domain as described in Sections 4 and 5.1.

Trading efficiency for precision. Since binary-level static analysis is very challenging, we give the user several levers for tuning trade-offs between precision and scalability, namely: k-callstring context sensitivity, loop unrolling, and different flavors of widening, such as delayed widening or widening with threshold.

CFG recovery. Precise Control-flow Graph (CFG) recovery is a major issue of binary-level static analysis [7] in presence of dynamic jumps, i.e. jump statements whose target is computed at runtime (like `jmp eax`). In our case, the imprecision due to the use of intervals may lead to significant loss of precision on dynamic jumps. We combine ideas

from [3] and [14] in order to get precise dynamic target evaluation: on a jump instruction, we compute a symbolic representation of the k-predecessors of the instruction, then pass this information enriched with our own forward abstract computation to a SMT solver and query all possible jump targets. Experiments on benchmarks from [14] confirm that the approach is precise and practical.

Precision recovery and degraded mode. In case the address of a `load` or `store` evaluates to $\top$, the user can ask the engine to *refine* the address value with a mechanism similar to that of our CFG recovery. In case refinement fails, the analysis can still enter a *degraded mode* (inspired from [22]), keeping when possible the former (non-$\top$) address value for propagation. Soundness will be lost, yet the analysis can go forward and (hopefully) discover interesting information.

7 Experiments

We want to assess the practical merits of the approach described so far. We are interested in the three following Research Questions: **RQ1:** What is the ability of template-based recovery to effectively recover high-level conditions, especially w.r.t. standard approaches? **RQ2:** What is the overhead of template-based recovery? **RQ3:** Does template-based recovery yield practical benefits to program analysis?

Practical concerns. We consider 66 programs taken from the JULIET/Samate benchmark from NIST [30] and 400 functions taken from 10 standard programs, such as `firefox` or `coreutils`. All programs are 32bit x86 executables for Linux (ELF format). Analysis have been performed with a limited bound on the calldepth (functions are stubbed once the bound is reached). Experiments have been performed on a Intel Core i5 CPU equipped with 8GB of RAM, and we rely on the Z3 SMT solver with a time-out of 1 second (no time-out was encountered). For the sake of comparison, we have implemented pattern-based recovery and logic-based recovery in BINSEC/VA.

7.1 Recovery ability (RQ1) and efficiency (RQ2)

We compare the three condition recovery approaches on all our benchmark functions. Results are presented in Table 2 (template-based approach) and Table 4 (summary).

The template-based approach performs very well (cf. Table 2), successfully recovering 89% of all conditions. A manual check on the 218 cases of failure indicates that most of them are actually not high-level conditions (columns DF, PF, x&y=0, CF_add in Table 2 – only column opt truly corresponds to unrecovered high-level conditions). The approach recovers in average 95% of high-level conditions (min: 92%, max: 100%).

Moreover, template-based recovery performs significantly better than logic-based and pattern-based approaches, which recover respectively 31% and 68% of the total number of conditions (Table 4). A more detailed analysis shows that template-based recovery is strictly better than logic-based recovery (*that was expected*), but also that template-based recovery and pattern-based recovery are not comparable, in the sense that they both recover some conditions not recovered by the other method. *This latter result was unexpected*, as patterns of Table 1 should represent all legitimate uses of flags.

Typically, the pattern-based method was able to recover tests to x&y==0, while template-based recovery typically beats patterns on optimized comparisons introduced by compilers, such as `or eax 0; je ...` for checking `eax = 0` (cf. Table 3).

Table 2. Template-based high-level condition recovery

progs	# fun	#loc†	#conds‡	#succ⋆	#fail						#smt	#cache	time	time$_{all}$
					DF	PF	x&y=0	CF$_{add}$	opt	all			(s)	(s)
firefox	40	21488	150 **(137)**	**134** | 89% **(98%)**	2	0	11	0	3	16	234	902	1.40	55.91
cat	40	6490	132 **(125)**	**116** | 88% **(92%)**	3	0	4	0	9	16	154	508	1.08	259.24
chmod	40	8954	183 **(172)**	**159** | 87% **(92%)**	3	0	8	0	13	24	203	750	1.44	313.17
cp	40	67199	174 **(162)**	**152** | 87% **(94%)**	0	0	12	0	10	22	533	4287	4.79	346.84
cut	40	7358	148 **(138)**	**132** | 89% **(96%)**	3	0	7	0	6	16	176	527	1.16	211.73
dir	40	9732	137 **(126)**	**118** | 86% **(94%)**	5	0	6	0	8	19	159	967	1.26	201.67
echo	40	8016	190 **(182)**	**168** | 88% **(92%)**	3	0	5	0	14	22	203	816	1.43	274.60
kill	40	6911	142 **(133)**	**125** | 88% **(94%)**	5	0	4	0	8	17	163	520	1.17	209.79
ln	40	88837	203 **(185)**	**177** | 87% **(96%)**	3	0	16	0	7	26	558	6565	4.88	531.58
mkdir	40	6347	125 **(117)**	**109** | 87% **(93%)**	3	0	5	0	8	16	147	505	1.01	235.80
Verisec	66	11552	394 **(370)**	**370** | 87% **(100%)**	0	8	0	16	0	24	463	735	3.31	34.48
total	466	242884	1978 **(1847)**	**1760** | 89% **(95%)**	30	8	78	16	86	218	2993	17082	22.93	2674.81

†: number of analyzed instructions only.

‡: total number of conditions (resp. high-level conditions). DF, PF, CF$_{add}$ and $x\&y = 0$ are not considered high-level.

⋆: total number of successfully recovered conditions, ratio w.r.t. total number of conditions (resp. high-level conditions)

#smt: number of calls to the smt solver – #cache: number of calls to the cache

time: time for recovery alone (s) – time$_{all}$: time for the whole analysis (s)

DF: check on direction flag – PF: check on parity flag – CF$_{add}$: low-level tricks on CF – x&y=0: encoded via `test`

opt: high-level conditions with operand update

Table 3. High-level conditions recovered by templates, not by patterns

Example	Discussion
`or eax, 0` `je ...`	The conditional jump is equivalent to `if (eax = 0) then goto ...`.
`cmp eax, 0` `jns ...`	Because the second operand of `cmp` is zero, checking the sign flag SF is sufficient to check if eax is greater that or equals zero i.e. (eax >= 0). Note that the pattern based method may miss this case if it expects a `jge` or a `jnl` instruction instead of `jns`. Note also that the folklore method may succeed to retrieve the high-level comparison if SF is encoded in DBA as `(eax < 0)` instead of `eax{31, 31}`.
`sar ebp, 1` `je ...`	Although this case seems to be complicated at first glance, the corresponding high-level predicate is merely equivalent to `if (ebp = 0) then goto ...`, where `ebp` holds its shifted value.
`dec ecx` `jg ...`	In addition to tracking assignments to flags as symbolic expressions, the template based solution also tracks assignments to operands mentioned in expressions of tracked flags. Hence it is able to infer the following high-level conditional jump `if (ecx` $\geq$ `0) then goto ...`

We can fruitfully combine patterns and templates in the following way: template recovery is used only when no pattern is found. This combination is faster, since patterns are significantly cheaper than templates, and it discovers more conditions than templates or patterns alone (especially, tests to x&y==0 are recovered). Results in Table 4 demonstrate a slight improvement in recovery and a 2x speedup.

Performance. Results in Tables 2 and Table 4 demonstrate that the approach has a low overhead, 1% in average (column time vs time$_{all}$). Hence, while template-based recovery is indeed much more expensive than the two other methods (Table 4), the extra-cost is clearly affordable. Finally, optimizations allow to win a factor 3x on average, up to 20x.

Conclusion. The template-based recovery approach is able to recover a large part of high-level conditions (**RQ1**), achieving significantly better results than related approaches. Especially, it can recover optimized comparisons introduced at compile-time,

Table 4. Summary: high-level condition recovery

method	#loc	#conds	#succ†	#fail	time	time_{all}
templates	242884	1978	**1760 (89%)**	218	**22.93**	2674.81
logic	247894	2260	**694 (31%)**	1566	0.003	2561.64
patterns	229255	1987	**1357 (68%)**	630	0.014	2373.33
templates + patterns	242884	1978	**1838 (92%)**	140	**9.17**	2659.95
templates **w/o cache**	242884	1978	**1760 (89%)**	218	29.76	2697.67
templates **w/o filtering**	242884	1978	**1760 (89%)**	218	51.13	2726.45
templates **w/o cache, w/o filtering)**	242884	1978	**1760 (89%)**	218	**66.52**	2752.73

†: ratio computed w.r.t. the total number of conditions

while they are beyond the scope of the pattern approach, and it can also sometimes synthesizes high-level conditions from low-level conditions with a priori no high-level counter-part in the source code. Concerning efficiency (**RQ2**), the method is very cheap, in the sense that its overhead w.r.t. the whole analysis cost is negligible. Moreover, templates can be fruitfully combined with patterns.

7.2 Practical impact (RQ3)

We consider two potential applications of this work: precision of static analysis and deobfuscation.

Application to value analysis. We are interested here in evaluating the gain of precision brought by better high-level condition recovery to a standard forward value propagation. We compare several versions of BINSEC/VA, based on templates, patterns and logic. Results are presented in Table 5. Here, template-based recovery leads to the computation of abstract memory states which are strictly more precise than those computed with logic-based recovery (on 41% of analyzed locations, up to 64%) and than those computed with pattern-based recovery (on 18% of analyzed locations, up to 38%). Moreover, template-based recovery does allow us to reduce the number of analysis failures in a tangible way (-18% in average, up to -80% on Verisec and -40% on firefox).

Application to deobfuscation. Obfuscation techniques aim at tricking reversers (either humans or tools) for preventing them to understand how a piece of code works. While it is legitimately used for IP protection, it is also massively used for malware protection, hence the need for automatic binary-level analysis of obfuscated programs. The code snippet `cmp eax ebx ; cmc ; jae` illustrates an obfuscation technique (reported in [33]) aiming at luring the reverser on the real semantic of a conditional jump. The standard `cmp eax ebx ; jae` pattern is usually read as branching on $eax \geq_u ebx$. But `jae` actually reads the carry flag `cf` (see Table 1), which is inverted by the `cmc` instruction. Hence, here, the true semantic of `jae` will be to branch on condition $eax <_u ebx$. *Template-based recovery succeeds to recover the true semantic of the jump, while pattern-based recovery and logic-based recovery fail.*

Table 5. Precision comparison between different condition recovery methods

progs	#loc†	# fail templ.	# fail logic	# fail patterns	#loc$_\sqsubset$ logic vs templ.	#loc$_\sqsubset$ pattern vs templ.
firefox	15099	242	433	400	8852 (59%)	5725 (38%)
cat	4192	136	143	145	1171 (28%)	604 (14%)
chmod	5768	188	201	202	1252 (22%)	652 (11%)
cp	5605	152	161	152	3237 (58%)	545 (10%)
cut	4870	148	232	156	1686 (35%)	605 (12%)
dir	5022	144	147	148	1442 (29%)	700 (14%)
echo	5570	176	185	186	2616 (47%)	1009 (18%)
kill	4626	150	157	158	1245 (27%)	625 (14%)
ln	8091	243	248	293	5166 (64%)	815 (10%)
mkdir	4062	134	141	142	1139 (28%)	589 (15%)
Verisec	8334	28	137	153	1474 (18%)	1075 (13%)
total	71239	1741	2185	2135	29280 (41%)	12944 (18%)

†: number of instructions analyzed by all three analyzers (instr. missed by at least one analyzer are discarded).
#fail: number of failures in the analysis due to a load/store index or jump expression evaluating to $\top$
#loc$_\sqsubset$: number of locations for which the abstract state computed by template-recovery is strictly more precise than the one computed with logic (resp. pattern) recovery

8 Related works

Logic-based [25,32,11,27], pattern-based [13] and virtual flag [32] solutions have already been lengthy discussed in Section 2. Basically our approach extends the logic-based method, and it is orthogonal to pattern in terms of recovery ability (yet, templates recover significantly more conditions than patterns). Templates can also be fruitfully combined with virtual flags and patterns, if available. Especially, very specific conditions such as `x&y==0` can be recovered this way. Finally, note that many syntactic disassembly techniques use patterns in an unsound way, for example not taking operand or flag updates into account.

Other more general proposed solution consists in tracing the bit-level calculation of flags. In this case, SAT solving is used to reason about values of variables. However, using SAT solving to perform static analysis faces scalability problems as soon as non-trivial loops are analyzed, even when combining SAT solving with abstraction to numeric ranges [10]. Interestingly, binary-level underapproximated techniques such as DSE [4] do not face this issue, and can rely on SAT solving and low-level flag encoding w/o any serious penalty.

9 Conclusion

We have presented template-based recovery, a sound and generic technique for recovering high-level conditions from binary codes. The method performs significantly better than state-of-the-art approaches, and it can also be combined with some of them. Template-based recovery can help to adapt any formal analysis from source-level analysis to binary-level analysis, and it can also be useful for reverse engineering.

References

1. Bourdoncle F. Efficient Chaotic Iteration Strategies With Widenings In: International Conference on Formal Methods in Programming and Their Applications, pp. 128-141, (1993)
2. Ball T., Cook B., Levin V. and Rajamani S. K. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft In: IFM 2004, Canterbury, UK, pp. 1-20, Springer 2004
3. Bardin S., Delahaye M., David R., Kosmatov N., Papadakis M. Traon Y. L. and Marion J. Sound and Quasi-Complete Detection of Infeasible Test Requirements In: ICST 2015, Graz, Austria, pp. 1-10, IEEE 2015
4. Bardin, S., Herrmann, P.: Structural Testing of Executables. In: ICST 2008. IEEE, Los Alamitos (2013)
5. Balakrishnan G., Reps T. W. DIVINE: DIscovering Variables IN Executables In: VMCAI 2007, Nice, France, pp. 1-28, Springer 2007
6. Bardin S. , Herrmann P., Leroux J., Ly O., Tabary R., Vincent A.: The BINCOA Framwork for Binary Code Analysis. In: CAV 2011. Springer, Heidelberg (2011)
7. Bardin S., Herrman P., Védrine F.: Refinement-based CFG Reconstruction from Unstructured Programs. In: VMCAI 2011. Springer, Heidelberg (2011)
8. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.: BAP: A Binary Analysis Platform. In: CAV 2011. Springer, Heidelberg (2011)
9. Brauer J. and King A. Transfer Function Synthesis without Quantifier Elimination. In: ESOP 2011. Springer (2011)
10. Brauer J. and King A.: Automatic Abstraction for Intervals using Boolean Formulae. In: Static Analysis Symposium, SAS, (2010)
11. Blazy S., Laporte V., Pichardie D.: Verified abstract interpretation techniques for Disassembling Low-level Self-modifying Code. In: ITP. springer (2014)
12. Balakrishnan, G., Reps, T. Analyzing Memory Accesses in x86 Executables. In: CC 2004. Springer, Heidelberg (2012)
13. Balakrishnan G., Reps T. W.: WYSINWYX: What you see is not what you eXecute. In: ACM Trans. Program. Lang. Syst. (2010)
14. Reinbacher T. and Brauer J. Precise control flow reconstruction using boolean logic. In: EMSOFT 2011. ACM (2011)
15. P. Cousot, R. Cousot Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints In: ACM Symposium on Principles of Programming Languages, POPL, ACM, pp. 238–252, (1977)
16. P. Cousot, R. Cousot Abstract interpretation frameworks The Journal of Logic and Computing, pp. 511–547, (1992)
17. Cousot P., Cousot R., Feret J., Mauborgne L., Miné A., Monniaux L. and Rival X. The ASTREÉ Analyzer In: ESOP 2005, Edinburgh, UK, pp. 21-30, Springer 2005
18. Dullien, T., Porst, S.: REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In: CanSecWest 2009. Available: http://www.zynamics.com/downloads/csw09.pdf
19. Djoudi A. and Bardin S., BINSEC: Binary Code Analysis with Low-Level Regions, In: TACAS 2015. Springer (2015)
20. David R., Bardin S., Ta T. D., Feist J., Mounier L., Potet M. and Marion J. BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-level Analysis In: SANER (2016)
21. Intel® 64 and IA-32 Architectures Software Developper's Manual. Order Number: 32546
22. Kinder, J., Kravchenko, D.: Alternating Control Flow Reconstruction. In: VMCAI 2012. Springer, Heidelberg (2012)

23. Kirchner F., Kosmatov N., Prevosto V., Signoles J. and Yakobowski B. Frama-C: A software analysis perspective In: Formal Asp. Comput., Volume 27, 2015
24. Kinder J., Zuleger F. and Veith H. An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries In: VMCAI 2009, Savannah, GA, USA, pp. 214-228, Springer 2009
25. Logozzo F., Fähndrich M.: On the Relative Completeness of Bytecode Analysis Versus Source Code Analysis. In: CC, pp. 197-212, Hungary(2008)
26. Mycroft A. Type-Based Decompilation (or Program Reconstruction via Type Reconstruction) In: ESOP 1999, Amsterdam, The Netherlands, pp. 208-223, Springer 1999
27. Miné A. Symbolic Methods to Enhance the Precision of Numerical Abstract Domains In: VMCAI 2006, Charleston, SC, USA, pp. 348-363, Springer 2006
28. Nielson F., Nielson H. R., and Hankin, C. Principles of Program Analysis 1999, ISBN:3540654100, Springer-Verlag New York, Inc. 1999
29. Navas J. A., Schachte P., Søndergaard H., and Stuckey P. J. Signedness-Agnostic Program Analysis: Precise Integer Bounds for Low-Level Code In: APLAS, pp. 115–130, (2012)
30. [website]: https://samate.nist.gov/SARD/testsuite.php
31. Reps T. W., Lim J., Thakur A. V., Balakrishnan G. and Lal A. There's Plenty of Room at the Bottom: Analyzing and Verifying Machine Code In: CAV 2010, Edinburgh, UK, pp. 41-56, Springer 2010
32. Sepp, A., Mihaila, B., Simon A.: Precise Static Analysis of Binaries by Extracting Relational Information. In: WCRE 2011. IEEE, Los Alamitos (2011)
33. Yadegari B., Johannesmeyer B., Whitely B., Debray S. A Generic Approach to Automatic Deobfuscation of Executable Code In: SP 2015. IEEE (2015)

A More details on recovered/unrecovered conditions

Low-level conditions leading to the failure of template-based recovery.

Pattern	Discussion	Example
DF	DF flag is especially used in string operations that move data from one register to a memory location pointer to by the EDI register. The EDI register is incremented or decremented automatically according to the setting of the DF flag	`rep stos [edi],eax`

PF	The parity flag is set if the number of bits set in the data is even and cleared if it is odd. No high-level condition corresponds to this case.	`mov edx, 3`
		`and edx, eax`
		`jp 805b4f6`

CF	The addition of 0xfefefeff and another operand a sets the carry flag if the High-order byte is not zero. In addition it decrements each byte of a by one. This idiom is used in string manipulation to check for null character.	`add ecx, 0xfefefeff` `jae 80a3620`
	Bits shifted beyond the destination operand boundary are first shifted into the CF flag, then discarded. At the end of the shift operation, the CF flag contains the last bit shifted out of the destination operand.	`shr ecx, 1` `jae 8061566`
x & y	Instructions like and, or, xor, not, and the shifts and rotates make it possible to test, set, clear, invert, and align bit fields within strings of bits. The test instruction can check to see if one or more bits in a register or memory location are non-zero.	`test al, 0x8` `jne 808e7a8`

opt	Compiler optimization may insert a mov instruction between a cmp instruction and a conditional jump. The mov instruction does not affect flags but may overwrite one operand from those used to set the flags for the following conditional jump.	`cmp [esp+0x64],eax`
		`mov eax, [esp+0x24]`
		`jg 80a3678`

High-level conditions recovered by templates, not by patterns. The following compiler idioms are not recovered by patterns, but by template-based recovery.

Example	Discussion
`or eax, 0` `je ...`	The conditional jump is equivalent to `if (eax = 0) then goto ...`.
`cmp eax, 0` `jns ...`	Because the second operand of `cmp` is zero, checking the sign flag SF is sufficient to check if eax is greater that or equals zero i.e. (eax >= 0). Note that the pattern based method may miss this case if it expects a `jge` or a `jnl` instruction instead of `jns`. Note also that the folklore method may succeed to retrieve the high-level comparison if SF is encoded in DBA as `(eax < 0)` instead of $\texttt{eax}_{\{31,\,31\}}$.
`sar ebp, 1` `je ...`	Although this case seems to be complicated at first glance, the corresponding high-level predicate is merely equivalent to `if (ebp = 0) then goto ...`, where `ebp` holds its shifted value.

`dec ecx`
`jg ...`
	Note that no pattern matches to this case. Equivalent DBA instructions are:
	`0: t := ecx - 1;`
	$\texttt{1: OF := (ecx}_{\{31,31\}}\texttt{=1}_{\{31,31\}}\texttt{)\& (ecx}_{\{31,31\}}\texttt{=t}_{\{31,31\}}\texttt{);}$
	`2: SF := (t < 0);`
	`3: ZF := (t = 0);`
	`4: ecx := t;`
	$\texttt{5: if (}\neg\texttt{ZF} \wedge \texttt{(OF = SF)) then goto ...;}$
	In addition to tracking assignments to flags as symbolic expressions, the template based solution also tracks assignments to operands mentioned in expressions of tracked flags. Hence it is able to infer the following high-level conditional jump $\texttt{if (ecx} \geq \texttt{0) then goto ...}$