Precise Control Flow Reconstruction Using Boolean Logic

Thomas Reinbacher
Embedded Computing Systems Group
Vienna University of Technology
Vienna, Austria
reinbacher@ecs.tuwien.ac.at

Jörg Brauer
Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany
brauer@embedded.rwth-aachen.de

ABSTRACT

This paper presents a SAT-based method for control flow graph reconstruction from executable code. The key idea of the technique is to express the semantics of each basic block in a program using Boolean logic, followed by inferring pre- and postconditions for each block through interleaved forward and backward analysis. In particular, the technique relies on register-wise value-set abstractions, which are subsequently refined using alternating forward and backward analyses. Experimental evidence shows that this approach, despite being sound, recovers the control flow graph precisely for different real-world benchmarks.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Assertion checkers; model checking; formal methods*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning About Programs—*Assertions; invariants; mechanical verification*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Algorithms, theory, verification

Keywords

Abstract interpretation, static analysis, binary code, control flow recovery, refinement, SAT solving

1. INTRODUCTION

Ideally, verification and validation of embedded software are applied to the executable binary code of a program because the semantics of each instruction is fully specified on this level. This contrasts with high-level programming languages such as C, and thus, promises to provide more trustworthy results [2, 16, 29]. In practice, however, binary code analysis presents severe challenges to overcome so as to make an analysis feasible for practical applications, one of which is indirect control flow. Indirect control poses a so-called chicken-and-egg problem [5, 6, 22, 32]: In order to reconstruct the control flow graph from binary code, it is necessary to infer invariants that describe those registers which affect the target of an indirect jump/call. A control flow graph is, in turn, required to compute these invariants.

1.1 The Drive for Control Flow Recovery

In presence of indirect control, e.g., implemented using indirect jump instructions, the lack of a precise control flow graph often implies a drastic loss in terms of precision for any subsequent verification effort. We discuss the loss of precision incurred by spurious jump targets using an example. Consider the following macro written in C:

```
#define SWPC(a, b) (a^=b,b^=a,a^=b,a&=0xf,b&=0xf)
```

The macro swaps the contents of two different variables a and b without involving a third, based on three consecutive exclusive-or operations. Indeed, this is a well-known idiom in low-level programming [35]. In addition, the last two operations clear the upper nibbles of the results. The left-hand side of Fig. 1 shows a semantically equivalent assembly snippet after compilation for the (accumulator-based) Intel MCS-51 architecture. The compiler locates the first instruction of the macro at address 0x100 in program memory. Now assume the SWPC macro is used within a switch-case statement, say:

```
switch (p) {
 case 10:  SWPC(x, y); break;
 case 20:  foo(x, y); break;
 default:  bar(x, y);
}
```

For efficiency, the switch-case is compiled into a jump table, which is stored in program memory. The application looks up a comparison value and the corresponding jump target pc' from program memory. These two values constitute a single entry in the jump table. Address pc' indicates the first instruction that belongs to the respective case block. If the comparison matches, control is redirected to pc' using an indirect jump. In the above example, the jump table consists of the comparison values 10, 20, and #default with the corresponding jump targets: 0x100 for the implementation of SWPC and the entry addresses of functions foo() and bar(). Suppose an analysis infers a range $\delta = [0x100, 0x105]$ for the indirect jump targets pc' related to SWPC by abstract interpretation [14] using intervals [11, 28]. On architectures supporting instructions of variable length (typically CISC),

EMSOFT'11, October 9–14, 2011, Taipei, Taiwan.

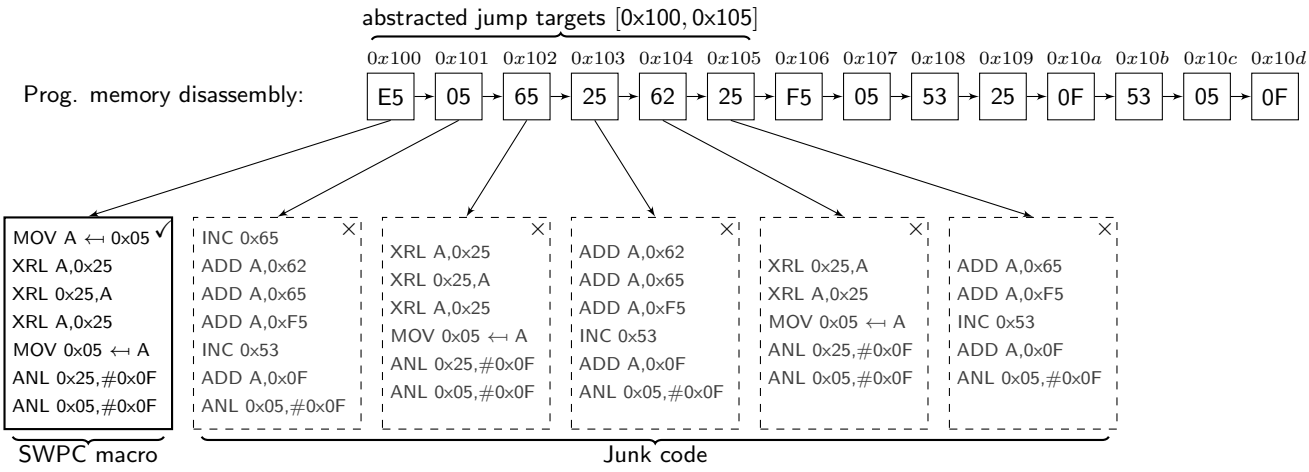

Figure 1: Approximation of indirect jump targets using intervals

edges need to be added to the CFG at the granularity of the shortest possible instruction-length. For the Intel MCS-51, this is byte-level granularity [19]. The drive for soundness forces us to add edges from the indirect jump to all concretizations of δ, i.e., 0x100, ..., 0x105. The first value 0x100 points to the SWPC macro. Not surprisingly, the next address 0x101 also leads to a valid instruction. The instruction MOV A ↤ 0x05, e.g., is represented by a two-byte opcode 0xE505 on the binary level. Likewise, the succeeding instruction XRL A, 0x25 is represented by the opcode 0x6525. Yet, the second byte of MOV A ↤ 0x05 and the first byte of XRL A, 0x25 form another valid opcode, i.e., 0x0565 that represents the instruction INC 0x65, which increments memory location 0x65. Indeed, the addresses 0x102, 0x103, 0x104, and 0x105 all indicate valid sequences of instructions (cf. Fig 1).

We call such fragments in the executable *junk code* as it is based on opcodes that are never executed, but only form part of the program semantics due to over-approximation. Subsequent analyses will therefore calculate invariants based on junk code as well. This, leads to a considerable loss of precision (also referred to as *unbearable noise propagation* [5, Sect. 1]), possibly yielding more spurious warnings. The situation is even worse if the junk code coincides with the opcodes of indirect or (un-)conditional jumps, which induces further control flow, though this is not the case in the above example. Albeit the derived interval appears to be tight at a first glance (its boundaries coincide with the jump targets), the imprecision due to the choice of intervals may lead to significant loss of analysis precision. This may render any verification efforts useless. Yet, indirect control is ubiquitous in compiler-generated as well as in handcrafted assembly code. Apart from switch-case statements, compilers generate indirect jumps/calls for function pointers or virtual methods. A more subtle point is given for return statements which alter the program counter according to a value that has been stored on the runtime stack. In some situations, compilers intentionally alter the runtime stack, e.g., when evaluating jump tables. Such code exhibits much similarity with code that exploits possibilities for buffer overruns. It is thus hardly possible to overestimate the value of precise invariants for such kind of program properties, thereby providing a means to distinguish *harmless* from *harmful* code.

1.2 Challenges in Control Flow Recovery

The problem of soundly analyzing indirect call and jump targets in real-world binary code is actually even more devious than discussed so far because of the presence of indirect stores. Such instructions store the contents of one register at a memory location indicated by another register. Observe that not only explicit usage of indirect store operations exhibits such behavior, but also PUSH instructions as well as direct calls, which automatically store the return address on the stack. Any indirect store operation thus needs to be handled properly. A further difficulty is the presence of bit-wise (logical) instructions in low-level code, and the frequent integration of overflowing behavior in the semantics of the program. For example, to compute a 16-bit addition on an 8-bit microcontroller architecture, compilers typically use an 8-bit addition followed by an 8-bit addition with carry. The remainder of this paper discusses an algorithm which performs alternating runs of forward and backward analysis [3] using SAT solving and integrates approaches to handle the challenges mentioned thus far. Basing the analysis on SAT solving allows to express the semantics of the analyzed programs relationally, thereby allowing the same representation to be used for both forward and backward analysis, which eases the efforts for implementing the technique.

1.3 Contributions

To summarize our work, the paper contributes a fully automatic algorithm for SAT-based control flow reconstruction. The technique itself only requires an encoding of the instruction-set semantics of the target hardware in propositional Boolean logic [9]. Based on these encodings, forward and backward value-set analyses can straightforwardly be implemented using existential quantification. This operation can, in turn, efficiently be implemented using SAT solving [10, 24]. By resting the analysis on a relational semantics of the instruction set, forward and backward analyses can be executed in a uniform fashion. The algorithm itself exhibits two interesting properties: (1) it is sound in the sense that it does not miss any edges in the CFG, and (2) it turns out to be precise for many examples of typical microcontroller programs. Further, the algorithm itself is fully generic, although we demonstrate it for the Intel MCS-51 architecture.

1.4 Structure of the Paper

We present the technical steps for a single basic block in Sect. 2, and then lift the approach to program-level fixed-point iteration in Sect. 3. Afterwards, Sect. 4 presents experimental evidence, before the paper concludes with a survey of related work in Sect. 5 and a discussion in Sect. 6.

2. BLOCK ABSTRACTION

In the following, we discuss the ingredients of our technique using an example. We first show how to derive tight invariants in terms of pre- and postconditions over value sets for a single basic block in the program. Afterwards, we lift the techniques described to entire programs in Sect. 3.

To simplify presentation, we first illustrate the approach for a basic block of assembly code, which corresponds to the SWPC macro introduced in Sect. 1. In general, a basic block b is a straight sequence of instructions with a single entry point and a single exit point. In the following we denote the basic block for the SWPC macro with b_{SWPC}. Our technique proceeds with representing the concrete semantics of b_{SWPC} using a propositional Boolean formula, which is composed from the instruction-level encodings of all involved instructions. This technique, which has become a standard technique in software verification due to bounded model checkers [7], is colloquially referred to as bit-blasting [8,13,23].

2.1 Bit-Blasting Blocks

As an example, consider the exclusive-or operations of the SWPC macro, e.g., XRL A, B, where A and B are some registers. This instruction computes the bit-wise exclusive-or of registers A and B and stores the result in A. To express the semantics of the instruction relationally, introduce bit-vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ to represent the values of A and B at the entry of the block. Likewise, we introduce a bit-vector $\boldsymbol{a}'$ to represent the value of A on exit (recall that B is not altered). In what follows, let $\boldsymbol{v}[i]$ denote the i^{th} bit of a bit-vector $\boldsymbol{v}$. The semantics of XRL A, B is then expressed using a Boolean formula as follows:

$$[\![\texttt{XRL A,B}]\!] := \bigwedge_{i=0}^{n-1} (\boldsymbol{a}'[i] \leftrightarrow \boldsymbol{a}[i] \oplus \boldsymbol{b}[i])$$

The force of such relational encodings is that they can be used to reason about program executions in both forward and backward direction (or even mixed). This means, given some values of $\boldsymbol{a}'$ on exit, it is straightforwardly possible to obtain potential values of $\boldsymbol{a}$ and $\boldsymbol{b}$ on entry. Similarly, given a value of, e.g., $\boldsymbol{a}$ on entry, it is also possible to infer suitable combinations of values of $\boldsymbol{b}$ and $\boldsymbol{a}'$. As another example, the increment INC C found in the junk code in Fig. 1 can be encoded over bit-vectors $\boldsymbol{c}$ and $\boldsymbol{c}'$ as:

$$[\![\texttt{INC C}]\!] := \bigwedge_{i=0}^{n-1} \left(\boldsymbol{c}'[i] \leftrightarrow \boldsymbol{c}[i] \oplus \bigwedge_{j=0}^{i-1} \boldsymbol{c}[j]\right)$$

Similar encodings can be derived for the entire instruction set of microcontrollers as discussed in [9]. In the following, we denote the encoding of an instruction i by $\mathsf{encode}(i)$. We can simply extend the propositional encoding of single instructions to entire basic blocks by first performing static single assignment conversion [15] (to avoid accidental coupling if a register is accessed more than once in the respective block). Thus, for a given block b, we construct a basic block formula ϕ_b as a conjunction over the propositional encodings of the instructions in b:

$$\phi_b = \bigwedge_{i=\mathsf{fst}(b)}^{\mathsf{lst}(b)} \mathsf{encode}(i)$$

Here, the entry and exit of a block are denoted $\mathsf{fst}(b)$ and $\mathsf{lst}(b)$, respectively. The formula for b_{SWPC} thus consists of seven conjuncts, each of which encodes a single instruction. However, passing ϕ_b to a SAT solver necessitates converting ϕ_b into conjunctive normal form (CNF) first. This is done using Tseitin conversion [26,34], which introduces fresh, existentially quantified variables $\boldsymbol{T}$ to obtain an equisatisfiable formula in CNF. We therefore denote the formula in CNF by $\phi_b(\boldsymbol{T})$. Introducing fresh variables ensures that the size of $\phi_b(\boldsymbol{T})$ in CNF is only a linear multiple of the size of ϕ_b.

2.2 Value-Set Abstraction

We apply value-set abstraction following the approach of Barrett and King [6, Sect. 3] to compute the unsigned values a register can take subject to a Boolean formula such as $\phi_b(\boldsymbol{T})$. The key idea of their algorithm is to use alternating runs of over- and under-approximation so as to converge onto a set of values for a pre-specified register. In the following, let $\langle\!\langle \boldsymbol{v} \rangle\!\rangle = \sum_{i=0}^{7} 2^i \cdot \boldsymbol{v}[i]$ denote the unsigned integer value of a bit-vector $\boldsymbol{v}$. Further, let $vars(\phi_b(\boldsymbol{T}))$ denote the set of propositional variables used in $\phi_b(\boldsymbol{T})$. To compute the value-sets of a bit-vector $\boldsymbol{v} \in vars(\phi_b)$, however, it is necessary to eliminate all variables in $vars(\phi_b(\boldsymbol{T})) \setminus \{\boldsymbol{v}\}$ from $\phi_b(\boldsymbol{T})$ using existential projection. To do so, we apply the SAT-based projection scheme of Brauer et al. [10], who showed how to perform existential quantifier elimination for propositional Boolean formulae using incremental SAT solving. Denote the function, which projects $\phi_b(\boldsymbol{T})$ onto $\boldsymbol{v}$, by $\mathsf{proj}_{\boldsymbol{v}}(\phi_b(\boldsymbol{T}))$. The result of this operation is a formula in CNF. Further, observe that $vars(\mathsf{proj}_{\boldsymbol{v}}(\phi_b(\boldsymbol{T}))) = \{\boldsymbol{v}\}$.

To illustrate, consider again the macro SWPC from the introduction. The inputs of the block are the bit-vectors $\boldsymbol{V}_{\mathsf{in}} = \{\boldsymbol{r}_{0\text{x}05}, \boldsymbol{r}_{0\text{x}25}\}$ since the output of the block only depends on the values of memory locations 0x05 and 0x25 on entry. Likewise, the outputs are $\boldsymbol{V}_{\mathsf{out}} = \{\boldsymbol{r}_{0\text{x}05}, \boldsymbol{r}_{0\text{x}25}, \boldsymbol{r}_{\mathsf{A}}\}$. Then, to determine the value set of one of the registers $\boldsymbol{v} \in \boldsymbol{V}_{\mathsf{in}} \cup \boldsymbol{V}_{\mathsf{out}}$, we compute $\mathsf{proj}_{\boldsymbol{v}}(\varphi_b(\boldsymbol{T}))$ and apply the algorithm of Barrett and King. The key ingredient of our algorithm is to derive preconditions on the bit-vectors in $\boldsymbol{V}_{\mathsf{in}}$ and post-conditions on the variables in $\boldsymbol{V}_{\mathsf{out}}$ for blocks using incremental SAT solving. After each iteration, the formula $\phi_b(\boldsymbol{T})$ is strengthened by adding a constraint that encodes the pre- and postconditions, respectively, before eliminating existential quantifiers. Using this technique, our analysis eventually converges onto a tight approximation of the actual value sets of registers.

2.3 Deriving Pre- and Postconditions

To encode pre- and postconditions in $\phi_b(\boldsymbol{T})$, we augment $\phi_b(\boldsymbol{T})$ with a propositional formulae $\psi_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}})$ or $\psi_{\mathsf{post}}(\boldsymbol{V}_{\mathsf{out}})$, respectively, which encodes the constraints imposed onto $\phi_b(\boldsymbol{T})$. Then, $\phi_b(\boldsymbol{T}) \wedge \psi_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}})$ describes the semantics of the block b subject to the precondition $\psi_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}})$. For example, if an analysis infers that memory location 0x25 on entry of b — corresponding to bit-vector $\boldsymbol{r}_{0\text{x}25}$ in the formula $\phi_b(\boldsymbol{T})$ — can take the values in $\{0\text{x}80, 0\text{x}51\}$, we augment the formula $\phi_b(\boldsymbol{T})$ with $\psi_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}})$ defined as:

$$\psi_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}}) = (\langle\!\langle \boldsymbol{r}_{0\text{x}25} \rangle\!\rangle = 0\text{x}80) \vee (\langle\!\langle \boldsymbol{r}_{0\text{x}25} \rangle\!\rangle = 0\text{x}51)$$

A CNF representation of $\psi_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}})$ can be straightforwardly derived as before by introducing fresh variables. Based on these formal notions, we can define forward and backward interpreters to derive value-set abstractions.

Forward Block-Wise Interpreter.

We define a forward interpreter $\overrightarrow{\mathcal{F}} : \mathsf{Bool}_{vars(\phi_b(\boldsymbol{T}))} \times \mathsf{Bool}_{\boldsymbol{V}_{\mathsf{in}}} \rightarrow \mathsf{Bool}_{\boldsymbol{V}_{\mathsf{out}}}$, where Bool_X describes the class of Boolean formulae over variables X. The forward interpreter determines a postcondition for a formula $\phi_b(\boldsymbol{T})$ and a precondition $\psi_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}})$ (which is initially *true*, i.e., it does not contain any constraints) as follows:

1. Let $\xi = \phi_b(\boldsymbol{T}) \wedge \psi_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}})$.
2. For each $\boldsymbol{v}_{\mathsf{out}} \in \boldsymbol{V}_{\mathsf{out}}$:
   (a) Eliminate all variables in $vars(\xi) \setminus \{\boldsymbol{v}_{\mathsf{out}}\}$ from ξ using incremental SAT solving, denoted $\mathsf{proj}_{\boldsymbol{v}_{\mathsf{out}}}(\xi)$.
   (b) Compute a value-set abstraction of $\mathsf{proj}_{\boldsymbol{v}_{\mathsf{out}}}(\xi)$, which yields a set of values in the range $0, \ldots, 255$.
3. Store results as the postcondition $\psi_{\mathsf{post}}(\boldsymbol{V}_{\mathsf{out}})$ of $\phi_b(\boldsymbol{T})$.

Backward Block-Wise Interpreter.

In a spirit similar to before, we define a backward interpreter $\overleftarrow{\mathcal{B}} : \mathsf{Bool}_{vars(\phi_b(\boldsymbol{T}))} \times \mathsf{Bool}_{\boldsymbol{V}_{\mathsf{out}}} \rightarrow \mathsf{Bool}_{\boldsymbol{V}_{\mathsf{in}}}$. The backward interpreter determines a precondition for a formula $\phi_b(\boldsymbol{T})$ and a postcondition $\psi_{\mathsf{post}}(\boldsymbol{V}_{\mathsf{out}})$ (which is initially *true*, too) as follows:

1. Let $\xi = \phi_b(\boldsymbol{T}) \wedge \psi_{\mathsf{post}}(\boldsymbol{V}_{\mathsf{out}})$.
2. For each $\boldsymbol{v}_{\mathsf{in}} \in \boldsymbol{V}_{\mathsf{in}}$:
   (a) Eliminate all variables in $vars(\xi) \setminus \{\boldsymbol{v}_{\mathsf{in}}\}$ from ξ using incremental SAT solving, denoted $\mathsf{proj}_{\boldsymbol{v}_{\mathsf{in}}}(\xi)$.
   (b) Compute a value-set abstraction of $\mathsf{proj}_{\boldsymbol{v}_{\mathsf{in}}}(\xi)$, which yields a set of values in the range $0, \ldots, 255$.
3. Store results as the precondition $\psi_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}})$ of $\phi_b(\boldsymbol{T})$.

Example.

To illustrate, assume a Boolean encoding of the precondition $\psi^{\mathsf{SWPC}}_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}})$ for b_{SWPC} which defines the registers $\langle\!\langle \boldsymbol{r}_{0\times05} \rangle\!\rangle \in \{10, 20, 30\} \wedge \langle\!\langle \boldsymbol{r}_{0\times25} \rangle\!\rangle \in \{50, 60, 70\}$ on input. Computing $\overrightarrow{\mathcal{F}}(\phi^{\mathsf{SWPC}}_b(\boldsymbol{T}), \psi^{\mathsf{SWPC}}_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}}))$ yields the postcondition:

$$\psi^{\mathsf{SWPC}}_{\mathsf{post}}(\boldsymbol{V}_{\mathsf{out}}) = \left\{ \begin{array}{lcll} \langle\!\langle \boldsymbol{r}'_{\mathsf{A}} \rangle\!\rangle & \in & \{50, 60, 70\} & \wedge \\ \langle\!\langle \boldsymbol{r}'_{0\times25} \rangle\!\rangle & \in & \{4, 10, 14\} & \wedge \\ \langle\!\langle \boldsymbol{r}'_{0\times05} \rangle\!\rangle & \in & \{2, 6, 12\} & \end{array} \right\}$$

We now apply the backward interpreter $\overleftarrow{\mathcal{B}}$ to the postcondition $\psi^{\mathsf{SWPC}}_{\mathsf{post}}(\boldsymbol{V}_{\mathsf{out}}) = \overrightarrow{\mathcal{F}}(\phi^{\mathsf{SWPC}}_b(\boldsymbol{T}), \psi^{\mathsf{SWPC}}_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}}))$ to infer a precondition for b_{SWPC} in backward direction, which yields:

$$\psi^{\mathsf{SWPC}}_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}}) = \left\{ \begin{array}{l} \langle\!\langle \boldsymbol{r}_{0\times05} \rangle\!\rangle \in \left\{ \begin{array}{l} 4, 10, 14, 20, 26, \ldots, \\ 234, 238, 244, 250, 254 \end{array} \right\} \wedge \\ \langle\!\langle \boldsymbol{r}_{0\times25} \rangle\!\rangle \in \left\{ \begin{array}{l} 50, 60, 70 \end{array} \right\} \end{array} \right\}$$

However, note that applying $\overleftarrow{\mathcal{B}}$ only leads to a coarse over-approximation of the initial precondition $\langle\!\langle \boldsymbol{r}_{0\times05} \rangle\!\rangle \in \{10, 20, 30\}$. This is a consequence of the fact that some instructions are not invertible, e.g., the instruction ANL 0x05,#0xF in b_{SWPC}.

3. PROGRAM-LEVEL ABSTRACTION

The previous section has discussed the technical steps for computing abstractions of a single block in forward or backward direction. Most notably, these are bit-blasting, existential quantifier elimination, and value-set abstraction. This section extends the techniques described so far to perform a whole-program analysis using alternating forward and backward abstract interpretation. We discuss the algorithm for whole-program analysis using the example presented in Fig. 2. The code fragment in Fig. 2 (middle) displays a typical use of indirect control flow in embedded software. An array of function pointers is stored in a table within the program memory of the microcontroller. The functions are then indexed (after some compiler optimizations) using the parameter keyCode, which is passed to function keypress().

The results of the analysis are value sets for all registers. This includes, most notably, the two 8-bit data pointer registers DPL and DPH, which are concatenated to form a 16-bit register DP on the Intel MCS-51. Together with an additive offset stored in the accumulator A, register DP indicates the jump targets (cf. instruction at address 0x106 in Fig. 2). The value-sets of these three registers are then used to extend the control flow graph of the program.

3.1 Preprocessing

In the first step, the program file is disassembled (in a sweep linear fashion) until an iJMP instruction is discovered. Then, the disassembler stops, and a control flow graph is extracted from the program fragment available thus far. A basic block representation of the program is then extracted from the control flow graph and each block is bit-blasted separately.

3.2 Worked Example

The annotated CFG in Fig. 2 shows the resulting binary code after compilation for the Intel MCS-51 microcontroller using the KEIL μVISION 3 v3.23 compiler. Basic block $b_{0\times03}$ implements the comparison keyCode $\geq$ N_HANDL using a subtract instruction SUBB A, #N_HANDL. The comparison evaluates to *true* if the subtraction does not underflow, which is indicated by the carry flag. For the *true* branch, the SUBB instruction clears the carry flag c and control flow is redirected to $b_{0\times0C}$. The constant value #C_FAIL is then stored in register $\boldsymbol{r}_{0\times07}$ as the return value.

However, if the comparison evaluates to *false*, then SUBB underflows, sets the carry flag c, and passes control flow to $b_{0\times0F}$. The basic block $b_{0\times0F}$ and its successors first calculate an offset for indexing the jump table pf. Subsequently, these blocks read the corresponding entries from the table, assign it to the data pointer DP, prepare the accumulator A and finally invoke the indirect jump iJMP @(A+DP) with the parameters read from program memory. In a concrete execution of the program the value of the carry flag is immediately known and determines the succeeding block.

To illustrate, suppose that b_{0x03} is entered with a precondition $\boldsymbol{r}_{0\times07} = 20$. This register corresponds to the case where keyCode $= 20$. Then, SUBB A, #N_HANDL will determine a value of 14 for the accumulator after the subtraction, the carry flag is cleared, and $b_{0\times0C}$ is processed next. It follows that $b_{0\times0C}$ has a concrete input state $\langle\!\langle \boldsymbol{r}'_{\mathsf{A}} \rangle\!\rangle = 14 \wedge \langle\!\langle \boldsymbol{c} \rangle\!\rangle = 0$, and the function keypress() returns without invoking any event handler.

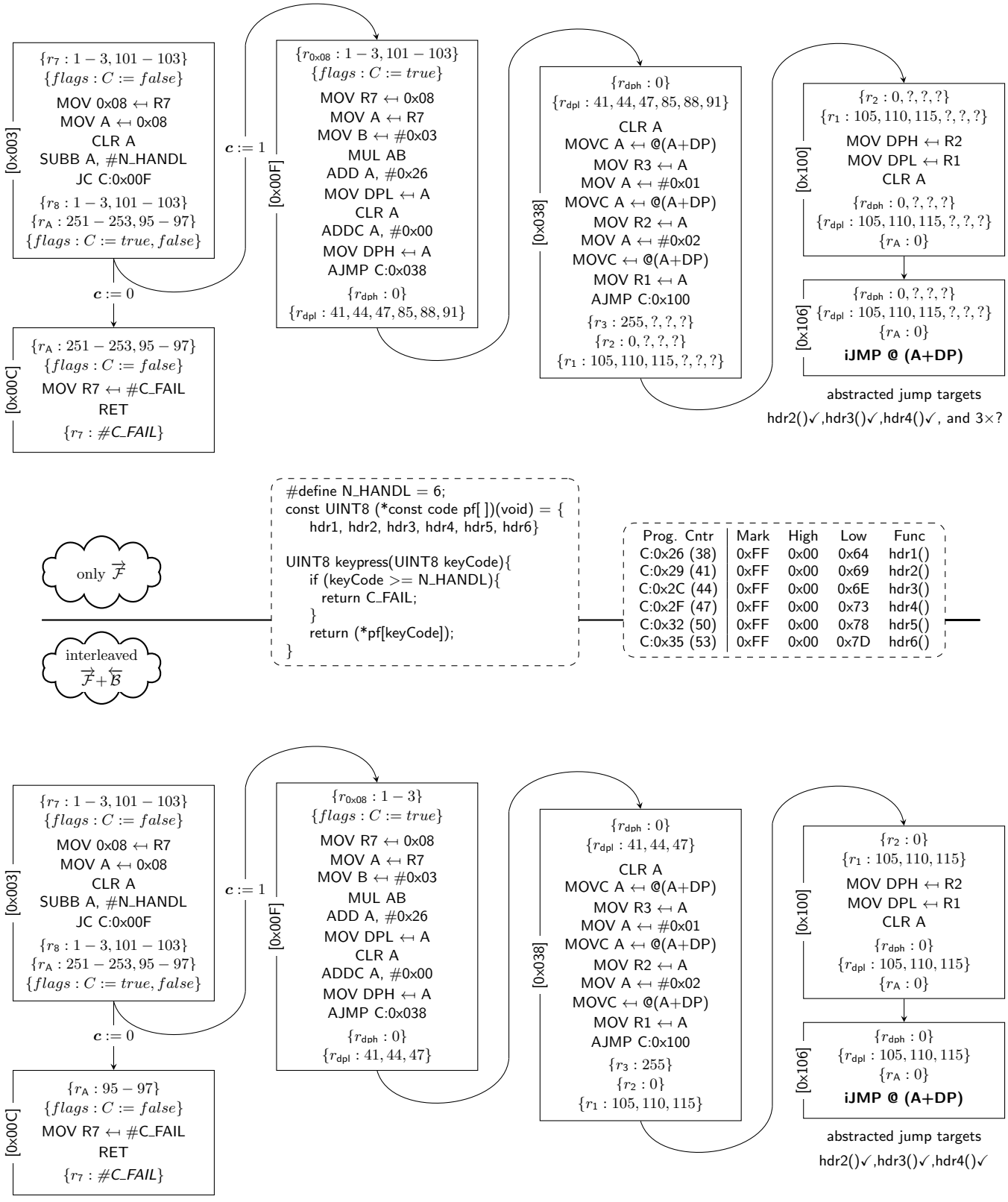

Figure 2: Forward (top) and interleaved forward-backward (bottom) interpretation

3.3 Forward Interpretation

In an abstract interpretation setting, however, we perform over-approximations of possible computations of the binary code under investigation. Concrete data types are mapped to abstract domains, i.e., non-relational value sets in our setting. Suppose the abstract interpreter enters the basic block $b_{0\times03}$ with the following precondition:

$$\psi^{b_{0\times03}}_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}}) = \left\{ \begin{array}{ll} \langle\!\langle \boldsymbol{r}_{0\times07} \rangle\!\rangle & \in \{1,2,3,101,102,103\} \quad \wedge \\ \langle\!\langle \boldsymbol{c} \rangle\!\rangle & \in \{\mathit{false}\} \end{array} \right\}$$

Applying the forward interpreter $\overrightarrow{\mathcal{F}}$ to compute the postcondition of $\phi_{b_{0\times03}}(\boldsymbol{T})$ subject to $\psi^{b_{0\times03}}_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}})$, we obtain:

$$\psi^{b_{0\times03}}_{\mathsf{post}}(\boldsymbol{V}_{\mathsf{out}}) = \left\{ \begin{array}{ll} \langle\!\langle \boldsymbol{r}'_{\mathsf{A}} \rangle\!\rangle & \in \{251-253, 95-97\} \quad \wedge \\ \langle\!\langle \boldsymbol{r}'_{0\times08} \rangle\!\rangle & \in \{1-3, 101-103\} \quad \wedge \\ \langle\!\langle \boldsymbol{c}' \rangle\!\rangle & \in \{\mathit{true}, \mathit{false}\} \end{array} \right\}$$

Observe that $\psi^{b_{0\times03}}_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}})$ contains valuations of $\boldsymbol{r}_{0\times07}$ that either fail ($\boldsymbol{c}' = \mathit{false}$) or pass ($\boldsymbol{c}' = \mathit{true}$) in the comparison. To compute a fixed point, $\psi^{b_{0\times03}}_{\mathsf{post}}(\boldsymbol{V}_{\mathsf{out}})$ is now propagated to both successors of $b_{0\times03}$, i.e., $b_{0\times0C}$ and $b_{0\times0F}$. While the propagated postcondition $\psi^{b_{0\times03}}_{\mathsf{post}}(\boldsymbol{V}_{\mathsf{out}})$ does not drastically impact analysis precision for $b_{0\times0C}$, it affects the jump targets reconstructed in the other branch. Neither of the values $\{101, 102, 103\}$ for $\langle\!\langle \boldsymbol{r}_{0\times07} \rangle\!\rangle$ can reach $b_{0\times0F}$ in a concrete execution since they would cause the comparison in $b_{0\times03}$ to hold (SUBB clears $\boldsymbol{c}$) and control would not reach $b_{0\times0F}$.

The effects of applying the forward interpreter $\overrightarrow{\mathcal{F}}$ on program-level are shown in the upper part of Fig. 2, yielding an approximation $\{\mathsf{hdr2()}, \mathsf{hdr3()}, \mathsf{hdr4()}\}$ of the jump targets. Yet, forward analysis also yields three spurious targets, denoted ? in the figure. This imprecision stems from the block $b_{0\times0F}$, which calculates the address of the corresponding entry in the jump table. The value set $\{101, 102, 103\}$ for $\langle\!\langle \boldsymbol{r}_{0\times08} \rangle\!\rangle$ on entry yields additional values $\{85, 88, 91\}$ for DPL in $b_{0\times0F}$. Reading from these addresses in $b_{0\times38}$ yields the specific byte in program memory, which depends on the remaining code of the program. This byte may thus hold an arbitrary value.

3.4 Invariant Refinement

To obtain tighter invariants, the results from forward interpretation are refined by interleaving forward and backward interpretation at conditional branching edges. Making use of the semantics of the conditional jump instruction, the outgoing edges of the respective block are labeled with the constraints from the conditional jump. To illustrate, we label the outgoing edges of $b_{0\times03}$ with the constraints on the carry flag $\boldsymbol{c}$ in Fig. 2. The edge $b_{0\times03} \rightsquigarrow b_{0\times0F}$ is thus labeled with $\boldsymbol{c} = \mathit{true}$, whereas the edge $b_{0\times03} \rightsquigarrow b_{0\times0C}$ is labeled with $\boldsymbol{c} = \mathit{false}$. From now on, given a set of edges E, we formalize edge labels by a map $\kappa : E \to \mathsf{Bool}_{\boldsymbol{V}_{\mathsf{in}} \cup \boldsymbol{V}_{\mathsf{out}}}$. Additionally, let $\mathsf{pred}(b)$ and $\mathsf{succ}(b)$ denote the set of immediate predecessors and successors of a block b, respectively.

The key idea of our algorithm is to use a form of depth-bounded backtracking, where the maximum depth is given by an unsigned integer k. To illustrate, assume that backtracking is performed starting from block $b_{0\times106}$. Then, a backtracking depth of $k = 3$ implies that the analysis backtracks along 3 predecessor blocks, i.e., it visits the blocks $b_{0\times100}$, $b_{0\times038}$, and $b_{0\times00F}$. The edges in the control flow graph succeeding conditional branching instructions are initialized with the constraints imposed by the respective instruction. Any other edge is labeled with the constraint *true*.

3.4.1 Algorithm

The analysis begins with a forward analysis of the program (1 and 2a). If an outgoing edge of the currently processed block contains a constraint, bounded backtracking is triggered (2b). To control the backtracking, we introduce an auxiliary variable i to monitor the current backtracking depth. Backtracking then starts by propagating the refined precondition of a successor block b_{succ} into b, where it is used as the postcondition so as to constrain the semantics of block b (3). This step is repeated k times. The refined value sets on input of the k^{th} predecessor are then used as the inputs for a forward analysis (4), the outcome of which is a refined value set on input of b_{succ} (5).

1. Apply $\overrightarrow{\mathcal{F}}$ to the initial block b in order to derive a postcondition $\psi_{\mathsf{post}}(\boldsymbol{V}_{\mathsf{out}}) = \overrightarrow{\mathcal{F}}(\phi_b(\boldsymbol{T}), \psi^b_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}}))$
2. For each $b_{\mathsf{succ}} \in \mathsf{succ}(b)$
   (a) If the edge does not impose any constraints, i.e., $\kappa(b \rightsquigarrow b_{\mathsf{succ}}) = \mathit{true}$, then join the precondition $\psi^{\mathsf{succ}}_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}})$ of b_{succ} with the postcondition $\psi_{\mathsf{post}}(\boldsymbol{V}_{\mathsf{out}})$ of b; repeat step 2 with the next successor
   (b) If $\kappa(b \rightsquigarrow b_{\mathsf{succ}}) \neq \mathit{true}$, continue with backtracking at step 3 and set $i = k$
3. Backtracking
   (a) Rename the constraint $\kappa(b \rightsquigarrow b_{succ})$ over variables in $\boldsymbol{V}_{\mathsf{in}}$ to range over variables in $\boldsymbol{V}_{\mathsf{out}}$; denote the resulting constraint by σ and put $\xi = \phi_b(\boldsymbol{T}) \wedge \sigma$
   (b) Apply $\overleftarrow{\mathcal{B}}$ to b, derive $\eta(\boldsymbol{V}_{\mathsf{in}}) = \overleftarrow{\mathcal{B}}(\xi, \psi^b_{\mathsf{post}}(\boldsymbol{V}_{\mathsf{out}}))$
   (c) The precondition of b is then refined by computing $\psi'^b_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}}) = \psi^b_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}}) \sqcap \eta(\boldsymbol{V}_{\mathsf{in}})$, where $\sqcap$ denotes the intersection of value sets
   (d) Decrement i. If i is positive and $|\mathsf{pred}(b)| = 1$, then repeat step 3 for $\mathsf{pred}(b)$; otherwise continue with step 4
4. Forward Refinement
   (a) Derive $\psi'^b_{\mathsf{post}}(\boldsymbol{V}_{\mathsf{out}}) = \overrightarrow{\mathcal{F}}(\phi_b(\boldsymbol{T}), \psi'^b_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}}))$
   (b) Increment i. If $i < k$ then set b to $\mathsf{succ}(b)$ and repeat step 4, otherwise continue at step 5
5. Join refined precondition
   (a) Rename $\psi'^b_{\mathsf{post}}(\boldsymbol{V}_{\mathsf{out}})$, which ranges over output variables $\boldsymbol{V}_{\mathsf{out}}$ so that it ranges over inputs $\boldsymbol{V}_{\mathsf{in}}$; denote the formula by σ'
   (b) Set $\psi^{succ}_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}}) = \sigma' \sqcup \psi^{succ}_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}})$
   (c) Continue with next successor in step 2

3.4.2 Refinement for Branching by Example

The bottom part of Fig. 2 shows the results of the algorithm with backtracking depth $k = 1$. This section discusses how the algorithm proceeds to resolve the conditional branching instruction JC C:0x00F, which passes to control to either location 0x00C or 0x00F, depending on the value of the carry flag. To do so, it first computes a postcondition $\psi^{b_{0\times03}}_{\mathsf{post}}(\boldsymbol{V}_{\mathsf{out}})$ for $b_{0\times03}$ in forward direction in step (1). This yields the same result as before, namely:

$$\psi^{b_{0x003}}_{\mathsf{post}}(\boldsymbol{V}_{\mathsf{out}}) = \left\{ \begin{array}{lll} \langle\!\langle \boldsymbol{r}'_{\mathsf{A}} \rangle\!\rangle & \in \{251-253, 95-97\} & \wedge \\ \langle\!\langle \boldsymbol{r}'_{0x08} \rangle\!\rangle & \in \{1-3, 101-103\} & \wedge \\ \langle\!\langle \boldsymbol{c}' \rangle\!\rangle & \in \{true, false\} & \end{array} \right\}$$

Then $\mathsf{succ}(b_{0x003}) = \{b_{0x00C}, b_{0x00F}\}$. Suppose the analysis proceeds with the successor b_{0x00F} in step (2) and determines the edge constraint $\boldsymbol{c} = true$, thus step (2b) applies. The algorithm then sets $i = 1$, and generates a restricted Boolean transformer $\xi = \phi_{b_{0x003}}(\boldsymbol{T}) \wedge \sigma$ in step (3a). Renaming is applied so that the constraint $\kappa(b_{0x003} \rightsquigarrow b_{0x00F})$ ranges over the outputs of b_{0x003} rather than the inputs of b_{0x00F}. Backtracking starts by applying $\overleftarrow{\mathcal{B}}$ to ξ and $\psi^{b_{0x003}}_{\mathsf{post}}(\boldsymbol{V}_{\mathsf{out}})$ in step (3b), and step (3c) yields a refined precondition:

$$\psi'^{b_{0x003}}_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}}) = \left\{ \begin{array}{lll} \langle\!\langle \boldsymbol{r}_{0x07} \rangle\!\rangle & \in \{1,2,3\} & \wedge \\ \langle\!\langle \boldsymbol{c} \rangle\!\rangle & \in \{false\} & \end{array} \right\}$$

Then, i is decremented to give $i = 0$ in step (3d). Since $\mathsf{pred}(b) = \emptyset$, the algorithm jumps to step (4) and applies the forward interpreter to the refined precondition of b_{0x003}. This yields a postcondition of b_{0x003} defined as follows:

$$\psi'^{b_{0x003}}_{\mathsf{post}}(\boldsymbol{V}_{\mathsf{out}}) = \left\{ \begin{array}{lll} \langle\!\langle \boldsymbol{r}'_{\mathsf{A}} \rangle\!\rangle & \in \{251,252,253\} & \wedge \\ \langle\!\langle \boldsymbol{r}'_{0x08} \rangle\!\rangle & \in \{1,2,3\} & \wedge \\ \langle\!\langle \boldsymbol{c}' \rangle\!\rangle & \in \{true\} & \end{array} \right\}$$

Incrementing i to 1 fails the test $i < k$ in step (4b), we thus proceed with step (5) and use the refined postcondition as precondition for the successor block by applying renaming in step (5a). Then, forward interpretation based on these new inputs gives a fresh, more precise precondition for b_{0x00F}, i.e.:

$$\psi^{b_{0x00F}}_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}}) = \left\{ \begin{array}{lll} \langle\!\langle \boldsymbol{r}_{0x08} \rangle\!\rangle & \in \{1,2,3\} & \wedge \\ \langle\!\langle \boldsymbol{c} \rangle\!\rangle & \in \{true\} & \end{array} \right\}$$

It is important to appreciate that, due to the refinement, the valuations in $\boldsymbol{r}_7$ were narrowed from $\{1,2,3,101,102,103\}$ to $\{1,2,3\}$. This refined value set coincides with those values that reach b_{0x0F} in a concrete execution of the program since the block b_{0x08} does not mutate this register. Finally, the algorithm continues with the *false* successor $b_{0x0C} \in \mathsf{succ}(b_{0x003})$ in step (2). Eventually, the algorithm refines the value set of $\boldsymbol{r}_7$ on entry of b_{0x00C} to $\{101,102,103\}$.

To identify the jump targets of the instruction at address 0x106 exactly, it is then necessary to propagate the refined precondition of block b_{0x00F} forward. This is required to compute the value of the data pointer register DP in the block b_{0x038}, where it is used twice to read a value from program memory. This is implemented using the MOVC instructions, which read the value located at address A+DP in program memory and store it in the accumulator A. Both values are then stored in registers R1 and R2, respectively. Block b_{0x100} copies the values of R1 and R2 into the data pointer register DP and clears the accumulator. Finally, block b_{0x106} uses DP to execute an indirect jump. Forward analysis computes the precondition

$$\psi^{b_{0x106}}_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}}) = \left\{ \begin{array}{lll} \langle\!\langle \boldsymbol{r}_{\mathsf{DPL}} \rangle\!\rangle & \in \{105,110,115\} & \wedge \\ \langle\!\langle \boldsymbol{r}_{\mathsf{DPH}} \rangle\!\rangle & \in \{0\} & \wedge \\ \langle\!\langle \boldsymbol{r}_{\mathsf{A}} \rangle\!\rangle & \in \{0\} & \end{array} \right\}$$

exactly, as desired. The three values 105, 110, and 115 indicate the addresses of the respective event handlers stored in the array pf of function pointers. Further, observe that the analysis thus infers that the functions hdr1, hdr5, and hdr6 are unreachable due to the precondition of the program which states that register r_7 only contains values drawn from the set $\{1,2,3,101,102,103\}$. For $r_7 \in \{1,2,3\}$, the functions hdr2, hdr3, and hdr4 are called, whereas keypress() returns for $r_7 \in \{101,102,103\}$. Finally, the program is disassembled again, this time also following the computed jump targets.

3.4.3 Optimizing for Indirect Reads

The previous section has demonstrated how the algorithm uses forward and backward analysis so as to resolve value sets after conditional branching instructions, which is crucial to discover the values of the data pointer DP in the indirect jump. A different problem is to resolve jump tables, though our algorithm handles this situation analogously. This section demonstrates this for the jump table generated for the switch-case statement from the introduction, shown in Fig. 3. A basic block b_{cmp} compares the control variable, which is stored in register R7, with the comparison value p from the jump table. The instruction MOV A $\leftarrow$ @(A+DP) fetches this value from program memory and stores it in the accumulator A. This means that the jump table is indirectly addressed by the data pointer. If the comparison matches, subsequent elements in the same row of the table are addressed by some additive offset on the accumulator. However, if the comparison fails, the data pointer is incremented twice in the next block so as to point to the next comparison value. The comparison starts again for the next case branch. Otherwise, the entry address of the corresponding case branch is loaded in the pre block and the indirect jump is executed.

Forward and backward interpretation of the block b_{cmp} thus involves reasoning about the statement MOVC A $\leftarrow$ @(A+DP), which performs an indirect read. For example, in the refinement step for the *true* successor we apply $\overleftarrow{\mathcal{B}}$ to b_{cmp} to obtain a precondition that causes the accumulator to hold the value 0 after execution of b_{cmp}. The effects of MOVC A $\leftarrow$ @(A+DP) thus need to be modeled. This could be encoded in the SAT instance by modeling the indirect read as a conditional read. Yet, this would cause the formula to explode in size. We therefore deviate from this approach and separate b_{cmp} into three blocks so as to handle the MOVC instruction outside the SAT solver, as a single atomic block.

We thus divide b_{cmp} into three blocks b_{a}, b_{b}, and b_{c} (cf. right-hand side of Fig. 3). Applying $\overrightarrow{\mathcal{F}}$ to b_{b} simplifies the process of collecting all bytes in memory which can be accessed subject to $\psi^{b_{\mathsf{b}}}_{\mathsf{pre}}(\boldsymbol{V}_{\mathsf{in}})$. Afterwards, we apply $\overleftarrow{\mathcal{B}}$ to b_{b}, which yields combinations of the DP and A that yield a fixed value for A on output of the block. We then only propagate the feasible combinations of DP and A to the successor b_{c}. We apply the same strategy for division DIV A,B and multiplication MUL A,B that are known to yield hard SAT instances.

4. EXPERIMENTS

We have integrated the analysis described in this paper into the [MC]SQUARE framework [31], which is written in JAVA. For SAT solving, we used SAT4J [25]. All experiments were performed on a Intel Core i5 CPU equipped with 4 GB of RAM. To evaluate the precision of our technique, we have applied it to two different sets of benchmarks for the Intel MCS-51. We have conducted the experiments with the expressed aim of answering the following two questions: (**i**)

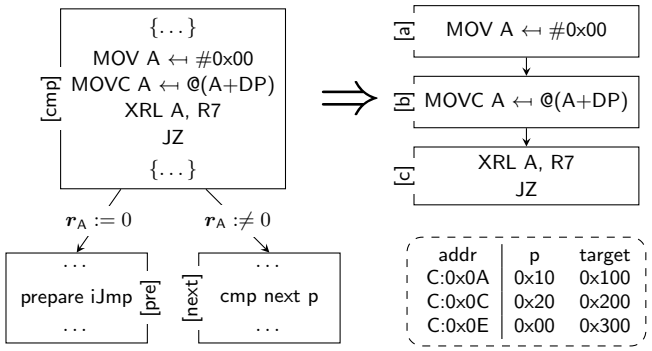

Figure 3: Data pointer refinement

Is the runtime tractable on non-trivial examples? (**ii**) How precisely are indirect jump targets recovered?

To show that our approach does not depend on the compiler used, we compiled all programs with both the KEIL μVISION 3 v3.23 and the SDCC v3.0.0 compiler. All programs use indirect control flow. The first benchmark set consists of embedded C programs which implement the sample programs in the tutorial *Array of Pointers to Functions* in the *Embedded Systems Programming* magazine [20]. The programs extensively use function pointers and pointer arithmetic.

Single Row Input The application reads data from a bidirectional input port of the microcontroller, which is connected to several push-buttons; each button is associated to a certain handler function. The starting addresses of the handler functions are stored in an array of function pointers within the program memory.

Keypad The application interfaces a 3×3 keypad. Whenever a key is pressed, the column and row number is used to lookup the corresponding handler implementation in a two-dimensional function pointer array.

Communication Link The application handles requests transmitted over a serial link (e.g., RS-232). Valid command sequences are stored as a table in program memory . A table-lookup together with pointer arithmetics determine the index of a function pointer table that holds the callback function to handle the request.

Task Scheduling The application implements a low-level task scheduler. It operates on a data structure that consists of an activation interval and a function pointer. An array holds one such entry for each task in the application (5 in the implementation). Whenever a time tick is received, the application iterates over all entries in the table and checks whether the interval in the table matches the elapsed time. It then uses the stored function pointer to indirectly branch to the task.

The second set of benchmarks consists of programs that make use of non-trivial switch-case statements, another major source of indirect control flow in practice.

Single Switch-Case An application where the control flow of the program is controlled by a non-nested switch-case statement with 18 distinct cases and one default branch. A compact range of values to test causes the compiler to optimize the switch-case statement into a jump table. The structure of this benchmark is similar to the motivating example in Fig. 1, yet it is more complicated to analyze due to the larger jump table.

Emergency Stop The application implements the *emergency stop* function block specified by the PLCopen consortium, which has defined safety-related aspects within the IEC 61131-3 development environment to support developers of Programmable Logic Controllers. The *emergency stop* function [27, pp. 40–45] monitors an emergency stop button in an industrial setting.

Tab. 1 shows the experimental results for these benchmarks. The sizes of the programs range from 52 to 180 instructions overall. The table clearly shows that pure forward analysis is insufficient for recovering jump targets precisely. Most of the data pointer values from forward analysis point to locations in program memory that are out of bounds, or that do not contain any meaningful instruction. Yet, integrating backward analysis with a small bound ($k = 2$) eliminates the redundant jump targets for all except one benchmark (Switch Case compiled using SDCC v3.0.0). This imprecision stems from the translation applied by the compiler. Here, the jump target depends on the carry flag, which is propagated through the program. Since our value set analysis is non-relational, it fails to capture this behavior. The jump targets are thus computed for both possible values of the carry flag, thereby leading to twice the number of jump targets. It is also interesting to note that in some situations combined forward and backward analysis is cheaper than pure forward analysis. This is because the value sets propagated around the program tend to be much smaller.

From our experience, the runtimes can be further reduced by substituting SAT4J using a more competitive SAT engine such as MINISAT (the speed-up is often tenfold). Thus the given timings are very conservative; indeed SAT4J was chosen to maintain the portability of [MC]SQUARE rather than for raw performance.

5. RELATED WORK

Albeit the problem of control flow reconstruction has recently received increasing attention [5, 16, 18, 21, 22], it has been studied for more than a decade already [32, 33]. The approach by De Sutter et al. [32] discuss control flow reconstruction for the Digital Alpha architecture. In this paper, they use so-called *hell nodes*, to which control is redirected if the jump target cannot be resolved. In contrast to our approach, their analysis starts with a conservative CFG which contains edges to hell nodes whenever indirect control is found. During the analysis, they incrementally discover information that allows to replace edges to hell nodes by edges to regular instructions. Thus, their analysis proceeds diametrically opposed to ours. In another early work, Theiling [33] includes knowledge about the compiler and the target architecture in his analysis for the TriCore and PowerPC architectures. He uses a bottom-up analysis, which is in some sense similar to our strategy. Cifuentes and Van Emmerik [12] described a technique for discovering jump tables in binary code (for decompilation) using a form of slicing and expression propagation. More recently, Holsti [18] proposed to use partial evaluation of switch tables to recover indirect jump targets. This approach, however, relies on

Binary Program					$\overrightarrow{\mathcal{F}}$ interpreter			$\overrightarrow{\mathcal{F}} + \overleftarrow{\mathcal{B}}$ interpreter				
Name	Compiler	loc_C	instr_B	JT	RT	FT	Time	RS	k	RT	FT	Time
Single row input	Keil	80	67	6	2401	2395	2.6	2	2	6	–	3.32
	Sdcc		52		460	454	2.4	2	2	6	–	2.0
Keypad	Keil	113	113	9	3844	3835	3.49	4	2	9	–	4.33
	Sdcc		80		1508	1499	3.08	4	2	9	–	2.57
Communication Link	Keil	111	164	8	6889	6881	4.56	2	2	8	–	4.37
	Sdcc		118		84	76	3.38	2	2	8	–	4.29
Task Scheduler	Keil	81	105	5	>1000	>995	>5m	17	2	5	–	14.03
	Sdcc		97					23	2	5	–	10.23
Switch Case	Keil	82	166	19	>5000	>4981	>5m	94	2	19	–	17.49
	Sdcc		180		3304	3285	2.31	6	2	38	19	2.6
Emergency Stop	Keil	138	150	9	768	759	2.8	2	2	9	–	2.6
	Sdcc		141		256	247	2.9	2	2	9	–	3.1

loc_C ... Lines of C code
instr_B ... Number of assembly instructions
JT ... Number of jump targets

RT ... Number of recovered targets
FT ... Number of recovered false targets
RS ... Number of refinement steps applied
k ... Backtracking depth
Time ... Analysis time in seconds

Table 1: Experimental results for pure forward analysis as well as combined forward and backward analysis

knowledge about the compiler used so that one can identify jump tables in program memory. By way of comparison, our approach computes such information. Kinder et al. [22] have presented a formal framework that incrementally builds a control flow graph from binary code using interleaving disassembly and abstract interpretation. They show that their algorithm yields the most precise CFG that can be recovered using the abstract domain employed. Since they only use local propagation of memory values, their algorithm, which was also implemented in the tool JakStab [21], cannot properly resolve indirect function calls. Later, the work of Kinder et al. was extended by Flexeder et al. [16] to the interprocedural setting. Control flow reconstruction is also performed by IdaPro [17], a tool that uses linear sweep disassembly. This tool is based on several techniques such as brute-force decoding of all addresses, pattern matching, and so on. However, the control flow graph provided by IdaPro is unsafe as shown in [4, 21]. Just recently, Bardin et al. [5] have tackled the problem using bounded value sets (k-set propagation). They have observed that typically only few constraints need to be tracked to resolve jump targets precisely. Based on this observation, they have thus developed an algorithm that takes care of what they call *precision requirements*. In this approach, refinement is used to control the k-bounds, based on backward propagation of precision requirements. A similar degree of locality in the analysis can also be seen in our framework, with the key difference that we adjust the backward propagation depth.

Note, though, that our algorithm is not bound to resolving indirect jump targets. As a side-effect, it also resolves the targets of indirect memory operations. Inferring the targets of indirect stores is especially important for microcontroller code since platforms such as the AVR series map registers, such as the status register, into the same address space as SRAM. Any indirect store operation may thus overwrite the contents of, say, the global interrupt flag [11]. On platforms such as the AVR, precision is even more important since the status register is located in the memory location adjacent to the address where the compiler places global variables, which are frequently addressed indirectly. A key difference of our method compared to that of others like Balakrishnan et al. [1, 2] is that we infer concrete integer addresses, rather than symbolic ones. This difference naturally manifests itself in the design of the analyses and the domains used (cf. the value-set abstract domain in [2]). The idea of using alternating of forward and backward analysis is not new. For instance, Balakrishnan et al. [3] use this technique to eliminate infeasible paths (called *syntactic language refinement*). Though applied to a different field, the correctness argument discussed in [3] also applies to our work. Mixed forward and backward analysis was also used by Rival [30] for counterexample generation using abstract interpretation.

6. CONCLUDING DISCUSSION

This paper argues that Boolean logic and SAT solving provide a promising means to reason about rather intricate examples of binary code. In particular, it shows that alternating executions of forward and backward analysis are useful to soundly recover jump targets in the value-set abstract domain. Expressing the concrete semantics of a program in the computational domain of propositional Boolean formulae allows to define its semantics relationally. This dovetails with our approach since the same encodings can thus be used for forward as well as for backward analysis, thereby sidestepping the difficulty of designing backward transformers [30]. Moreover, the approach benefits from the progress on state-of-the-art SAT solvers, which can easily decide satisfiability of structured problems involving thousands of variables and clauses. Indeed, the problems we confront the solver with can almost be seen as trivial by nowadays standards. We have explained and presented the techniques in the context of jump target recovery. Though this is a compelling problem, it is important to appreciate that our methods are merely independent of it. They can, similar to the handling of indirect reads in Sect. 3.4.3, also be used to model indirect stores.

7. ACKNOWLEDGMENTS

The work of Thomas Reinbacher has been supported within the FIT-IT project CevTes managed by the Austrian Research Agency FFG under grant 825891. The work of Jörg Brauer has been, in part, supported by the DFG Cluster of Excellence on *Ultra-high Speed Information and Communication*, German Research Foundation grant DFG EXC 89 and the DFG research training group 1298 *Algorithmic Synthesis of Reactive and Discrete-Continuous Systems*. The authors want to thank Andy King, Axel Simon, Heinrich Moser, and Stefan Kowalewski for interesting discussions.

8. REFERENCES

[1] G. Balakrishnan and T. W. Reps. Analyzing memory accesses in x86 executables. In *CC*, pages 5–23, 2004.

[2] G. Balakrishnan and T. W. Reps. WYSINWYX: What You See Is Not What You eXecute. *ACM Trans. Program. Lang. Syst.*, 32(6), 2010.

[3] G. Balakrishnan, S. Sankaranarayanan, F. Ivančić, O. Wei, and A. Gupta. SLR: Path-sensitive analysis through infeasible-path detection and syntactic language refinement. In *SAS*, volume 5079 of *LNCS*, pages 238–254. Springer, 2008.

[4] S. Bardin and P. Herrmann. OSMOSE: Automatic structural testing of executables. *Softw. Test., Verif. & Reliab.*, 2009.

[5] S. Bardin, P. Herrmann, and F. Védrine. Refinement-based CFG reconstruction from unstructured programs. In *VMCAI*, volume 6538 of *LNCS*, pages 54–69. Springer, 2011.

[6] E. Barrett and A. King. Range and set abstraction using SAT. *Electron. Notes Theor. Comput. Sci.*, 267:17–27, October 2010.

[7] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.

[8] J. Brauer and A. King. Automatic abstraction for intervals using boolean formulae. In *SAS*, volume 6337 of *LNCS*, pages 167–183. Springer, 2010.

[9] J. Brauer, A. King, and S. Kowalewski. Range analysis of microcontroller code using bit-level congruences. In *FMICS*, volume 6371 of *LNCS*, pages 82–98. Springer, 2010.

[10] J. Brauer, A. King, and J. Kriener. Existential quantification as incremental SAT. In *CAV*, volume 6806 of *LNCS*, pages 191–207. Springer, 2011.

[11] J. Brauer, T. Noll, and B. Schlich. Interval analysis of microcontroller code using abstract interpretation of hardware and software. In *SCOPES*. ACM, 2010.

[12] C. Cifuentes and M. Van Emmerik. Recovery of jump table case statements from binary code. In *IWPC*, pages 192–199. IEEE Computer Society, 1999.

[13] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[14] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[15] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Effciently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, pages 451–590, 1991.

[16] A. Flexeder, B. Mihaila, M. Petter, and H. Seidl. Interprocedural control flow reconstruction. In *APLAS*, volume 6461 of *LNCS*, pages 188–203. Springer, 2010.

[17] Hex-Rays SA. Idapro. http://www.hex-rays.com/idapro/.

[18] N. Holsti. Analysing switch-case tables by partial evaluation. In *WCET*, 2007.

[19] Intel Cooperation. *MCS 51 Microcontroller Family User's Manual*, 1994. Order No.: 272383-002.

[20] N. Jones. Arrays of pointers to functions. *Embedded Systems Programming Magazine*, 05:46–56, May 1999.

[21] J. Kinder and H. Veith. Jakstab: A static analysis platform of binaries. In *CAV*, volume 5123 of *LNCS*, pages 423–427. Springer, 2008.

[22] J. Kinder, H. Veith, and F. Zuleger. An abstract interpretation-based framework for control flow reconstruction from binaries. In *VMCAI*, volume 5403 of *LNCS*, pages 214–228. Springer, 2009.

[23] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.

[24] S. K. Lahiri, R. E. Bryant, and B. Cook. A Symbolic Approach to Predicate Abstraction. In *CAV*, volume 2725 of *LNCS*, pages 141–153. Springer, 2003.

[25] D. Le Berre. SAT4J: Bringing the power of SAT technology to the Java platform, 2010. http://www.sat4j.org.

[26] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.

[27] PLCopen. Safety software, technical specification, part1: Concepts and function blocks. online, 2006.

[28] J. Regehr and A. Reid. HOIST: A system for automatically deriving static analyzers for embedded systems. *ACM SIGOPS Operating Systems Review*, 38(5):133–143, 2004.

[29] T. W. Reps, J. Lim, A. V. Thakur, G. Balakrishnan, and A. Lal. There's plenty of room at the bottom: Analyzing and verifying machine code. In *CAV*, volume 6174 of *LNCS*, pages 41–56. Springer, 2010.

[30] X. Rival. Understanding the origin of alarms in Astrée. In *SAS*, volume 3672 of *LNCS*, pages 303–319. Springer, 2005.

[31] B. Schlich. Model checking of software for microcontrollers. *ACM Trans. Embedded Comput. Syst.*, 9(4), 2010.

[32] B. D. Sutter, B. D. Bus, K. D. Bosschere, P. Keyngnaert, and B. Demoen. On the static analysis of indirect control transfers in binaries. In *PDPTA*, 2000.

[33] H. Theiling. Extracting safe and precise control flow from binaries. In *RTCSA*, pages 23 –30, 2000.

[34] G. S. Tseitin. On the complexity of derivation in the propositional calculus. In A. O. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic*, volume Part II, pages 115–125, 1968.

[35] H. S. Warren. *Hacker's Delight*. Addison-Wesley, 2002.