Abstract Interpretation of Microcontroller Code: Intervals Meet Congruences

Jörg Brauer[a,*], Andy King[b], Stefan Kowalewski[a]

[a] *Embedded Software Laboratory, RWTH Aachen University, Germany*
[b] *Portcullis Computer Security Limited, Pinner, UK*

Abstract

Bitwise instructions, loops and indirect data access present challenges to the verification of microcontroller programs. In particular, since registers are often memory mapped, it is necessary to show that an indirect store operation does not accidently mutate a register. To prove this and related properties, this article advocates using the domain of bit-wise linear congruences in conjunction with intervals to derive accurate range information. The paper argues that these two domains complement one another when reasoning about microcontroller code. The paper also explains how SAT solving, which applied with dichotomic search, can be used to recover branching conditions from binary code which, in turn, further improves interval analysis.

Keywords: embedded systems, binary code, abstract interpretation, linear congruences, intervals

1. Introduction

Recent research in the fields of programming languages and computer security have led to the development of a large number of techniques and tools for analysing programs for runtime errors and security vulnerabilities [22, 24, 27, 32, 40, 41, 63]. These tools use model checking [3, 20] or abstract interpretation [23] to approximate the set of reachable program states. Most tools focus on analysing source code presented in a high-level programming language such as C or Java. Such tools naturally operate on a high-level of abstraction, e.g. by assuming integer variables to have unbounded precision, or by representing the heap memory symbolically. Furthermore, on desktop computers, which are the target of classical source code analysers, operating systems typically control the hardware and the interaction between hardware and software. This is not so for (microcontroller) binary code, which presents different challenges [5–7, 18, 19, 59, 60, 65, 67, 73, 74] to verification than those posed by programs written in high-level languages. Microcontroller binary code typically executes a nonterminating loop in which communication with the environment is performed. Data is then stored and processed, often using bitwise operations, before values are written to the output ports. Control logic, which is often formulated in terms of Boolean relations on status flags, and bit-wise operations necessitate reasoning about the program semantics at the granularity of bits. This presents a problem to verification efforts based on abstract interpretation since most work is tailored to representations of program semantics using geometric abstractions such as affine [42] or polyhedral spaces [26, 72]. The conceptual difference between high-level geometric concepts and low-level bitwise relations presents a semantic gap that needs to be bridged. Furthermore, the hardware is configured and controlled directly by the given program. Verification tools thus need to integrate a hardware model. Specifically on hardware such as the ATMEL microcontroller series [1], any verification argument must also pay special attention to the targets of indirect writes. An indirect write is a store operation in which the contents of one register are stored at a target address that is held in another register. The target address of the store operation is thus determined at runtime. On the ATmega family of microcontrollers, however, registers are merely reserved memory locations in the same address space as the SRAM. It is thus possible to mutate a register, such as the stack pointer or the program status word, if the target coincides with the address of the register.

*Corresponding author
Email addresses: brauer@embedded.rwth-aachen.de (Jörg Brauer), a.m.king@kent.ac.uk (Andy King), kowalewski@embedded.rwth-aachen.de (Stefan Kowalewski)
URL: http://www.embedded.rwth-aachen.de (Jörg Brauer)

```
0x50: LDI R17 0
0x51: LDI R30 66
0x52: LDI R31 0
0x53: MOV R26 R8
0x54: MOV R27 R9
0x55: RJUMP 2
0x56: LPMPI R0 Z
0x57: STPI X R0
0x58: CPI R30 69
0x59: CPC R31 R17
0x5A: BRNE -5
0x5B: RET
```

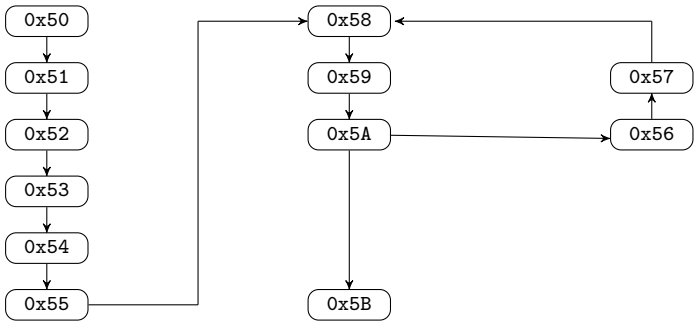

Figure 1: `memcopy` loop for the ATMEL ATmega16

1.1. Applications of Range Analysis in Binary Code Verification

One approach to microcontroller verification is thus to assume that indirect stores never access registers [58]. This approach is based on the assumption that code which accesses registers indirectly is so flawed that verification is not worthwhile anyway. Though appealing in its simplicity, this assumption is dubious for handcrafted assembly code. It is also not unknown for compilation itself to introduce errors, particularly at higher levels of optimisation [29, 78]. The problem of reasoning about the targets of indirect writes is further compounded by the fact that indirect stores often arise in loops that are responsible, for instance, for data initialisation. Then, the same indirect store operation may write to a number of different targets. A related problem is therefore showing that all such targets fall within some range that does not overlap with the registers themselves [18]. The value of this information extends beyond tracking the values of registers. Indeed, virtually any static analysis that is applied to microcontroller binary code either directly depends on or indirectly benefits from the results of range analysis. These analyses include, but are not limited to, classical gen/kill bit-vector analyses [61] and partial order reductions [34, 76] — analyses that can be considered to be client analyses of range analysis. When these client analyses are deployed on microcontroller code, which typically realise some form of concurrency, it is necessary to reason about the execution of interrupt handlers (see [66, Sect. 5] and [68]). On ATmega microcontrollers, the interrupt handlers themselves are controlled and thus depend on the value of the global interrupt flag. Furthermore, this flag forms part of the program status word, which can be accessed indirectly, hence the causal, though indirect, relationship between range analysis and the client analyses.

1.2. Illustrative Example

This paper addresses the problem of statically analysing the targets of indirect stores, whilst simultaneously modelling data at the bit-level. Since the set of targets cannot be exactly determined statically, we employ abstract interpretation [23] to compute a range of addresses that contains all possible targets. If the enclosing range is suitably tight, it is possible to verify that the registers are not overwritten. Figure 1 illustrates some ATmega16 [1] assembly code and the corresponding control flow graph (CFG). The instructions at locations `0x50` – `0x52` assign the 8-bit registers `R17`, `R30` and `R31` to the decimal constants `0`, `66` and `0`, respectively. The following two instructions initialise the registers `R26`, and `R27` with symbolic values stored in other registers. The relative jump `RJUMP 2` passes control to location `0x58`. The `LPMPI R0 Z` instruction first loads `R0` with the contents of the byte at the address in program memory determined by the 16-bit register `Z`; then `Z` is incremented. The instruction `STPI X R0` stores the contents of `R0` into the byte at address `X` and then increments `X`. On the ATmega16 microcontroller, each indirect memory access is indicated through one of the 16-bit pointer registers `X`, `Y` or `Z`. The register `X` is a short-hand for concatenating the 8-bit registers `R26` and `R27`, the 8-bit registers `R28` and `R29` constitute the 16-bit register `Y`, and likewise `Z` is an alias for `R30` and `R31`. It is important to note that the ATmega has a Harvard architecture, and hence, program memory is separate from SRAM. Location `98` in program memory, for example, is different from location `98` in SRAM. Thus, program memory is accessed with special instructions such as `LPMPI R0 Z` (load byte from program memory at address `Z` in `R0` and post-increment `Z`), whereas SRAM is accessed using instructions such as `STPI X R0` (store `R0` in SRAM at address `X` and post-increment `X`). Detecting self-modifying code, which we do not consider, is thus trivial. The instructions `CPI R30 69` and `CPC R31 R17` compare `Z` against `69`. This is implemented by subtracting the second

argument from the first one and setting the status flags accordingly. For the subsequent `BRNE` instruction, only the zero flag in the status register, which is set iff `Z` equals `69`, is relevant. The net effect of this code is to copy the contents of three locations in program memory starting at address `66` into SRAM, where the start of the target SRAM region is defined by the values of the registers `R8` and `R9` on input.

A classical interval analysis such as the one implemented in our binary code analyser [MC]SQUARE [18, 65, 66] can infer that `Z` $\in [66, 69]$ at program location `0x5A`. The interval analyser derives the bound on `Z` based on the combination of `CPI`/`CPC` instructions followed by `BRNE`. However, it fails to discover that `X` falls in a range that contains four integer values where the lower bound is defined by the initial values `R8` and `R9`. Any sound program analysis based solely on the interval domain thus has to assume that the indirect store operation at `0x57` could access any addressable memory location, including the registers. Moreover, if the `CPI`/`CPC` instructions were to restrict `X` instead of `Z` then, conversely, the values `Z` would be unbounded. This is a well-known limitation of interval analysis and stems from that fact that it is non-relational, that is, cannot express relationships between registers. To avoid such a loss of precision, we combine the results of a relational analysis for bit-level equalities with interval analysis, with the goal of proving that `X` is incremented only in combination with `Z`. This is sufficient to show partial correctness of the above program fragment with respect to indirect writes.

1.3. Linear congruences

In microcontroller code for the ATmega16 platform, a memory region is statically reserved rather than dynamically allocated. Thus, the address of the start of a region that is used as an array is fully determined. Hence, when verifying such code, it is not necessary to use a symbolic name to refer to a memory region; an address will suffice. The force of this is that there is no need to adopt a memory model in which regions with different symbolic names are assumed to be non-interfering. Symbolic memory models are often employed when the position of a region is unknown, as with dynamically allocated memory in C, but this nevertheless compromises soundness [6]. Furthermore, when basing an analysis on symbolic memory models, it is only possible to infer a relationship between a concrete offset into a symbolic region and its value [35]. By way of contrast, when analysing statically reserved regions, it is even possible to infer a numeric relationship between each address of a region, and the contents of that address.

To represent such relationships, we turn to linear congruences [2, 44, 45, 55] (see [30] for an overview of these works). In this classical abstract domain [36, 37], the relationships between variables, or equivalently registers, are described as systems of linear equations which are implicitly conjoined where each equation takes the form

$$\sum_{i=0}^{n-1} c_i \cdot x_i \mod m = d$$

where $c_i \in \mathbb{Z}$ are integer coefficients, x_i are program variables, $m \in \mathbb{N}$ is a modulus and $d \in \mathbb{Z}$ is an integer constant. Henceforth, denote the above equation by $\sum_{i=0}^{n-1} c_i \cdot x_i \equiv_m d$. Such a system may have none, one, or many solutions, where a solution is an assignment to the values of the variables $x_0, \ldots, x_{n-1}$ that satisfies each of the equations. For example, the system

$$u + 2v \equiv_{256} 3 \quad \wedge \quad v + w \equiv_{256} 1$$

has solutions:

$$\{\langle 1 + 256k_1 + 2k_3, 1 + 256k_2 - k_3, k_3\rangle \in [0, 255] \mid k_1, k_2, k_3 \in \mathbb{Z}\}$$

with the interpretation that the first, second and third elements of the tuple prescribe values for u, v and w respectively. Such relationships arise between variables — or memory locations in the case of microcontroller code — because of the modular nature of computer arithmetic. It is therefore natural to consider moduli corresponding to the size of a machine word [55]. Such systems can only represent linear relations subject to wrap-around arithmetic, but not ranges, and therefore we adopt a more expressive class of congruences based on decomposing variables into their constituent bits [16, 44, 45]. For instance, suppose u is represented by an unsigned byte whose bits are $\langle u_0, \ldots, u_7\rangle$ where $u_i \in \{0, 1\}$. Suppose too that v and w are likewise represented by $\langle v_0, \ldots, v_7\rangle$ and $\langle w_0, \ldots, w_7\rangle$, respectively. Then the above system can be expressed as

$$\sum_{i=0}^{7} 2^i(u_i + 2v_i) \equiv_{256} 3 \quad \wedge \quad \sum_{i=0}^{7} 2^i(v_i + w_i) \equiv_{256} 1$$

without any loss of information. It has been shown how such systems can be applied to verify bit-twiddling algorithms [44, 45]. The main contribution of this paper is to show how bit-wise congruences can be married with intervals to reason about the partial correctness of binary code, in particular the targets of indirect writes.

1.4. Contributions

Since this paper extends our earlier work [16], we clarify our contributions as follows:

- We explain how relational binary code semantics for microcontroller code can be applied to synthesise transfer functions for the ATmega16 instruction set;
- We show how the same semantics can be applied to recover branching conditions from binary code using repeated calls to a SAT solver [15], which refines the pattern-based approach described in [18];
- We explain how to combine intervals and congruence equalities by applying incremental SAT solving;
- We show how a contiguous range, such as $[0, 6]$, can be refined to a set of non-contiguous values, such as $\{0, 2, 4, 6\}$, by applying congruences to ranges;
- To summarise, we show how it is possible to infer accurate ranges by combining intervals and congruences, and thereby verify the partial correctness of microcontroller binary code with respect to indirect writes.

2. Outline of the Approach

Overall the paper advocates a pragmatic approach to the verification of executable code, and in particular binaries that are compliant with the AMTEL microcontroller series. To argue that an indirect write does not mutate a register, interval analysis is deployed to bound the range of addresses that can possibly be written at a particular point in the code. Interval analysis is, in turn, supported by an analysis for linear congruence relations that is deployed to primarily improve the fidelity of the range information. Then, to argue the partial correctness of all the indirect writes, it is sufficient to show that the target addresses do not overlap with an address of any of the memory mapped registers (Sect. 4.5 explains why there is no circularity in this argument).

2.1. Automatic Abstraction

Since the AMTEL series supports over one hundred instructions, we advocate automatically computing transfer functions from Boolean formulae that describe the relational semantics of each instruction. This is performed off-line by focusing on single instructions and using templates for register identifiers. Automation is nevertheless still attractive because of the complexity of the microcontroller instructions, particularly with regard to the status flags. Transfer functions are thus derived for each of the instructions for the domain of linear congruences, though using bit-wise relations so as to capture the bit-level operations are commonplace in microcontroller code.

There is no need to compute transfer functions off-line for interval analysis because, for ATMEL code at least, ranges only need describe the values of 8-bit data-objects. Therefore we deploy a form of on-line synthesis in which a Boolean function is used to describe the ranges of the registers before the instruction is executed. This formula is combined with a propositional description of the instruction itself and dichotomic search is applied to calculate the output ranges, requiring 8 invocations of a SAT solver for each bound.

2.2. Invariant Refinement

Linear congruences and intervals are complementary abstract domains: the former expresses relational information between different registers whereas the latter captures values that can be stored in individual registers (without reference to the values stored in others). Nevertheless, there is an interplay between congruence and interval information so that one can be used to refine the other, and thereby improve the quality of the analysis as a whole. Refinement is also important in the handling of microcontroller code that includes conditional branches; here the issue is how to reinterpret branching code as a form of conditional interval constraint so as to tighten ranges. Again, these refinement techniques are ultimately aimed at improving interval analysis and which, itself, it designed to meet the ultimate objective of verifying the safety of indirect writes.

2.3. Structure of the Paper

Sect. 2 briefly reviews the preliminaries required for these domains so as to keep the paper self-contained. The ethos of our method is to derive symbolic transformers for congruences and intervals, then derive program invariants, and finally refine the computed invariants in both domains by passing information between them. Sect. 3 introduces the basic ingredients of this scheme in terms of a series of examples (returning to the verification problem discussed within Sect. 1.1). A relational semantics for ATmega16 binary code, which is used to pre-synthesise symbolic transformers for the instruction set of the microcontroller, is discussed in Sect. 4, followed by an algorithm that recovers branching conditions in terms of linear constraints using incremental SAT solving in Sect. 5. Sect. 6 explains how to extract branching information from a binary that can be used to strengthen range information. A reduction technique for intervals and congruences, which strengthens the abstract invariants in both domains, is discussed in Sect. 7. The paper concludes with a survey of related work in Sect. 8 and a discussion in Sect. 9.

3. Abstract Domains

The key idea in abstract interpretation is to simulate the execution of each concrete operation $f : C \to C$ in a program using an abstract analogue $g : D \to D$, where C and D denote domains of concrete values and abstract descriptions, respectively. This section briefly reviews the abstract domains of intervals and bit-level linear congruences on which our analysis is built. In what follows, let $m = 2^w$ where $w \in \mathbb{N}$ denote the word-length of the microcontroller. Further, let $\mathbb{Z}_m = \{i \in \mathbb{N} \mid 0 \leq i \leq m-1\}$, let $\mathcal{V} = \{v_0, \ldots, v_{n-1}\}$ be a set of program variables, and let $\mathcal{P}$ denote the set of program locations (or instructions, equivalently).

3.1. Intervals

In our work, the interval abstract domain, is defined $\mathsf{Int} = \{\bot\} \cup \{[\ell, u] \mid 0 \leq \ell \leq u \leq 255\}$ and used to over-approximate subsets of $\mathbb{Z}_{256}$. A partial order over Int is defined $\bot \sqsubseteq_{\mathsf{Int}} i$ for all $i \in \mathsf{Int}$ and $[\ell_1, u_1] \sqsubseteq_{\mathsf{Int}} [\ell_2, u_2]$ iff $\ell_2 \leq \ell_1$ and $u_1 \leq u_2$. To define a Galois connection $(\wp(\mathbb{Z}_m), \mathsf{Int}, \alpha_{\mathsf{Int}}, \gamma_{\mathsf{Int}})$ between sets of concrete values and and intervals, let $\mathsf{fst}([\ell, u]) = \ell$ and $\mathsf{snd}([\ell, u]) = u$.

Definition 1. The abstraction $\alpha_{\mathsf{Int}} : \wp(\mathbb{Z}_m) \to \mathsf{Int}$ and concretisation $\gamma_{\mathsf{Int}} : \mathsf{Int} \to \wp(\mathbb{Z}_m)$ maps are defined as follows:

$$\alpha_{\mathsf{Int}}(v) = \begin{cases} \bot & \text{if } v = \emptyset \\ [\mathsf{min}(v), \mathsf{max}(v)] & \text{otherwise} \end{cases} \qquad \gamma_{\mathsf{Int}}(i) = \begin{cases} \emptyset & \text{if } i = \bot \\ \{z \in \mathbb{Z}_m \mid \mathsf{fst}(i) \leq z \leq \mathsf{snd}(i)\} & \text{otherwise} \end{cases}$$

3.2. Congruences

In addition to intervals, our analysis is based on abstracting input-output relations, expressed as propositional formulae, as systems of congruence equations. To explain this abstraction, let $\mathsf{sol}(f) \subseteq \mathbb{B}^n$ denote the set of solutions (models) of a Boolean function over n propositional variables where $\mathbb{B} = \{0, 1\}$. Likewise let $\mathsf{sol}(c) \subseteq \mathbb{Z}_m^n$ denote the set of solutions of a congruence system c over n variables.

Example 1. If $f = x_1 \wedge (x_2 \oplus x_3)$ then $\mathsf{sol}(f) = \{\langle 1,0,1\rangle, \langle 1,1,0\rangle\}$. Moreover, if $c = (x_1 + x_2 \equiv_{256} 3) \wedge (x_3 \equiv_{256} 1)$ then $\mathsf{sol}(c) = \{\langle 0,3,1\rangle, \langle 1,2,1\rangle, \langle 2,1,1\rangle, \langle 3,0,1\rangle, \langle 4,255,1\rangle, \langle 5,254,1\rangle, \langle 6,253,1\rangle, \ldots, \langle 255,4,1\rangle\}$.

Sets of Boolean vectors, hence Boolean functions, can be abstracted with systems of congruences as follows:

Definition 2. The abstraction $\alpha_{\mathsf{Cong}} : 2^{\mathbb{B}^n} \to 2^{\mathbb{Z}_m^n}$ and concretisation $\gamma_{\mathsf{Cong}} : 2^{\mathbb{Z}_m^n} \to 2^{\mathbb{B}^n}$ maps are defined:

$$\alpha_{\mathsf{Cong}}(S) = \left\{ \boldsymbol{x} \in \mathbb{Z}_m^n \;\middle|\; \begin{array}{lll} \{\boldsymbol{y}_0, \ldots, \boldsymbol{y}_{n-1}\} \subseteq S & \wedge & \{\lambda_0, \ldots, \lambda_{n-1}\} \subseteq \mathbb{Z} \quad \wedge \\ \sum_{j=0}^{j<n} \lambda_j \equiv_m 1 & \wedge & \boldsymbol{x} \equiv_m \sum_{j=0}^{j<n} \lambda_j \boldsymbol{y}_j \end{array} \right\} \qquad \gamma_{\mathsf{Cong}}(S) = S \cap \mathbb{B}^n$$

Example 2. Returning to the function f from Ex. 1, we have:

$$\alpha_{\mathsf{Cong}}(\mathsf{sol}(f)) \quad = \quad \{\langle 1,0,1\rangle, \langle 1,1,0\rangle, \langle 1,2,255\rangle, \ldots, \langle 1,254,3\rangle, \langle 1,255,2\rangle\}$$

To keep the paper self-contained, we finally discuss the closure algorithm for bit-wise linear congruences originally described in [45], which is shown in Alg. 1. The algorithm is formulated in terms of some auxiliary functions: $\mathsf{row}(M], i)$ extracts the i^{th} row from a matrix M, where the first row is taken to be 1; $\mathsf{triangular}(M)$ puts M into upper triangular form using Gaussian elimination; $\mathsf{numRows}(M)$ returns the number of rows in M. The algorithm starts with the unsatisfiable congruence system $[A \mid \boldsymbol{b}] = [0, \ldots, 0, 1]$ in line 1; this corresponds to the $\bot$ element in the congruence lattice. $[A \mid \boldsymbol{b}]$ is then incrementally extended by joining the congruence system with solutions that are not yet covered by $[A \mid \boldsymbol{b}]$; the join operation can also be computed using triangularisation [44, Prop. 3]. This construction is applied in lines 5–12. The algorithm terminates once there is no model of f left which is not described by $[A \mid \boldsymbol{b}]$. Alg. 1 effectively computes α_{Cong} as shown in [45, Prop. 1].

Algorithm 1 Compute the congruence closure α_{Cong} of a Boolean function f with respect to a fixed modulus $m \in \mathbb{N}$. Here, f is equivalently represented by its set S of models.

1: $[A \mid \boldsymbol{b}] \leftarrow [0, 0, \ldots, 0, 1]$
2: $i \leftarrow 0$
3: $r \leftarrow 1$
4: **while** $i < r$ **do**
5: $\quad \langle a_1, \ldots, a_k, b\rangle \leftarrow \mathsf{row}([A \mid \boldsymbol{b}], r - i)$
6: $\quad S' \leftarrow \{\boldsymbol{x} \in S \mid \langle a_1, \ldots, a_k\rangle \cdot \boldsymbol{x} \not\equiv_m b\}$
7: $\quad$ **if** $\exists \boldsymbol{x} \in S'$ **then**
8: $\quad\quad [A' \mid \boldsymbol{b}'] \leftarrow [A \mid \boldsymbol{b}] \sqcup [\mathsf{Id} \mid \boldsymbol{x}]$
9: $\quad\quad [A \mid \boldsymbol{b}] \leftarrow \mathsf{triangular}([A' \mid \boldsymbol{b}'])$
10: $\quad\quad r \leftarrow \mathsf{numRows}([A \mid \boldsymbol{b}])$
11: $\quad$ **else**
12: $\quad\quad i \leftarrow i + 1$
13: $\quad$ **end if**
14: **end while**
15: **return** $[A \mid \boldsymbol{b}]$

4. Invariant Generation by Example

This section builds towards our general method for analysing microcontroller code using several examples. First transfer functions are derived for each instruction of the ATmega16 prior to analysis: one transfer function for inferring invariants that are systems of congruence (cp. Sect. 4.1), and another transfer function for inferring interval constraints (cp. Sect. 4.2). For invariant generation, branching conditions which are extracted from the binary code using SAT solving (cp. Sect. 4.3) augment the system, which increases precision. Finally, the congruent invariants are strengthened with intervals, which yields more precise representations of congruences as well as intervals (cp. Sect. 4.4).

4.1. Synthesising Transfer Functions for Congruences

As a first example, consider the instruction `EOR R0 R1`, which computes the bit-wise exclusive-or of registers `R0` and `R1` and stores the result in `R0`. Although this instruction and accompanying transfer function are both relatively straightforward, the ATMEL ATmega16 possesses over one hundred different instructions in all [1], all of which need to be separately abstracted. This motivates applying the abstraction scheme of [45] to automatically derive a transfer function from a bit-level specification of the instruction prior to analysis itself.

We diverge slightly from the abstraction technique of [45] by considering generic instructions that are parameterised by registers r and s. To express the semantics of `EOR` r s, we introduce bit-vectors $\boldsymbol{r}$ and $\boldsymbol{s}$ to represent the value of r and s prior to the instruction and use $\boldsymbol{r}'$ and $\boldsymbol{s}'$ to represent their values after it. We index a bit-vector $\boldsymbol{v}$ by $\boldsymbol{v}[i]$ with the understanding that $\boldsymbol{v}[0]$ denotes the least significant element of $\boldsymbol{v}$. With $\oplus$ denoting the Boolean exclusive-or, `EOR` r s

is encoded propositionally as:

$$[\![\texttt{EOR}\ \ r\ \ s]\!] \triangleq \begin{cases} \bigwedge_{i=0}^{7}(\boldsymbol{r}'[i] \leftrightarrow (\boldsymbol{r}[i] \oplus \boldsymbol{s}[i])) & \wedge & \bigwedge_{i=0}^{7}(\boldsymbol{s}'[i] \leftrightarrow \boldsymbol{s}[i]) & \wedge \\ S' \leftrightarrow (N \oplus V) & \wedge & V' \leftrightarrow 0 & \wedge \\ N' \leftrightarrow \boldsymbol{r}'[7] & \wedge & Z' \leftrightarrow \bigwedge_{i=0}^{7} \neg \boldsymbol{r}'[i] & \wedge \\ C' \leftrightarrow C & \wedge & T' \leftrightarrow T & \wedge \\ H' \leftrightarrow H & \wedge & I' \leftrightarrow I & \end{cases}$$

where N and N' denote the values of the negative flag of the status register before and after execution of the instruction. Likewise C, S, V, Z, T, H and I respectively denote the initial values of the carry, sign, the two's complement overflow, the zero, transfer, half-carry and the global interrupt flag. The primed versions have the obvious interpretation. With the eigth status flags, the formula has $2 \cdot 8 + 8 = 24$ input variables and 24 outputs. Note that `EOR` does not update the carry flag, the transfer flag, the half-carry flag and the interrupt flag, which is reflected by the respective bi-implications in the propositional formula. We then proceed by putting the above formula $[\![\texttt{EOR r s}]\!]$ into conjunctive normal form (CNF), which introduces fresh, existentially quantified variables [56, 75]. Introducing fresh variables during CNF conversion ensures that an equisatisfiable formula is constructed with only a linear increase in size.

Once the formula is in CNF, the so-called congruent closure [45] can be applied to derive a system of congruence equations that express the relationship between the input and output variables (and those alone). The algorithm proceeds by repeatedly calling a SAT solver, thus suppose the solver has produced the following model:

$$\mathbf{s}_1 = \begin{Bmatrix} \boldsymbol{r}[0] \mapsto 0, & \boldsymbol{r}[1] \mapsto 1, & \boldsymbol{r}[2] \mapsto 0, & \boldsymbol{r}[3] \mapsto 1, & \boldsymbol{r}[4] \mapsto 0, & \boldsymbol{r}[5] \mapsto 1, & \boldsymbol{r}[6] \mapsto 0, & \boldsymbol{r}[7] \mapsto 1, \\ \boldsymbol{s}[0] \mapsto 1, & \boldsymbol{s}[1] \mapsto 1, & \boldsymbol{s}[2] \mapsto 1, & \boldsymbol{s}[3] \mapsto 1, & \boldsymbol{s}[4] \mapsto 1, & \boldsymbol{s}[5] \mapsto 1, & \boldsymbol{s}[6] \mapsto 1, & \boldsymbol{s}[7] \mapsto 1, \\ \boldsymbol{r}'[0] \mapsto 1, & \boldsymbol{r}'[1] \mapsto 0, & \boldsymbol{r}'[2] \mapsto 1, & \boldsymbol{r}'[3] \mapsto 0, & \boldsymbol{r}'[4] \mapsto 1, & \boldsymbol{r}'[5] \mapsto 0, & \boldsymbol{r}'[6] \mapsto 1, & \boldsymbol{r}'[7] \mapsto 0, \\ \boldsymbol{s}'[0] \mapsto 1, & \boldsymbol{s}'[1] \mapsto 1, & \boldsymbol{s}'[2] \mapsto 1, & \boldsymbol{s}'[3] \mapsto 1, & \boldsymbol{s}'[4] \mapsto 1, & \boldsymbol{s}'[5] \mapsto 1, & \boldsymbol{s}'[6] \mapsto 1, & \boldsymbol{s}'[7] \mapsto 1, \\ C \mapsto 1, & H \mapsto 1, & I \mapsto 1, & N \mapsto 1, & S \mapsto 0, & T \mapsto 1, & V \mapsto 1, & Z \mapsto 1, \\ C' \mapsto 1, & H' \mapsto 1, & I' \mapsto 1, & N' \mapsto 1, & S' \mapsto 0, & T' \mapsto 1, & V' \mapsto 0, & Z' \mapsto 0 \end{Bmatrix}$$

The algorithm hinges on observing that $\mathbf{s}_1$ can also be represented as the 0/1 solution to a system of congruences:

$$\mathbf{m}_1 = \begin{Bmatrix} \boldsymbol{r}[0] \equiv_{256} 0, & \boldsymbol{r}[1] \equiv_{256} 1, & \boldsymbol{r}[2] \equiv_{256} 0, & \boldsymbol{r}[3] \equiv_{256} 1, & \boldsymbol{r}[4] \equiv_{256} 0, & \boldsymbol{r}[5] \equiv_{256} 1, & \boldsymbol{r}[6] \equiv_{256} 0, & \boldsymbol{r}[7] \equiv_{256} 1, \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ C' \equiv_{256} 1, & H' \equiv_{256} 1, & I' \equiv_{256} 1, & N' \equiv_{256} 1, & S' \equiv_{256} 0, & T' \equiv_{256} 1, & V' \equiv_{256} 0, & Z' \equiv_{256} 0 \end{Bmatrix}$$

where a modulo of 256 is chosen because it coincides with the size of a single item of data (byte) on the ATMEL ATmega16 [1]. In the next step, we add a disequality constraint induced by the last congruence equation of $\mathbf{m}_1$ and construct the augmented formula $[\![\texttt{EOR r s}]\!] \wedge ([\![\boldsymbol{r}[0] \not\equiv_{256} 0]\!] \vee \ldots \vee [\![Z' \not\equiv_{256} 0]\!])$ [44] where $[\![\boldsymbol{r}[0] \not\equiv_{256} 0]\!]$ denotes the propositional encoding of the disequality constraint $\boldsymbol{r}[0] \not\equiv_{256} 0$ etc. These additional constraints (which can actually be simplified [45]) ensure that the SAT solver finds a model that is not described by $\mathbf{m}_1$ if such a model exists. Passing this formula to a SAT solver yields a new model $\mathbf{s}_2$ which is distinct from the previous solution. As with $\mathbf{s}_1$, $\mathbf{s}_2$ is then represented as a congruence system $\mathbf{m}_2'$. Next we compute a single congruence system $\mathbf{m}_2$ that describes both $\mathbf{m}_1$ and $\mathbf{m}_2'$, namely $\mathbf{m}_2 = \mathbf{m}_1 \sqcup_{\mathsf{Cong}} \mathbf{m}_2'$ where $\sqcup_{\mathsf{Cong}}$ is the least upper bound operation for systems of congruences [45, 55]. Following this strategy we construct a chain of congruence systems $\mathbf{m}_1, \mathbf{m}_2, \ldots$ where each system has strictly more solutions that the previous one. The algorithm terminates when the additional constraints make the formula unsatisfiable, at which point the very last system in the chain is as follows:

$$\alpha_{\mathsf{Cong}}([\![\texttt{EOR}\ \ r\ \ s]\!]) = \begin{cases} \bigwedge_{i=0}^{7}(128\boldsymbol{r}'[i] \equiv_{256} 128\boldsymbol{r}[i] + 128\boldsymbol{s}[i]) & \wedge & \bigwedge_{i=0}^{7}(\boldsymbol{s}'[i] \equiv_{256} \boldsymbol{s}[i]) & \wedge & C' \equiv_{256} C & \wedge \\ 128S' \equiv_{256} 128N + 128V & \wedge & V' \equiv_{256} 0 & \wedge & N' \equiv_{256} \boldsymbol{r}[7] & \wedge \\ H' \equiv_{256} H & \wedge & T' \equiv_{256} T & \wedge & I' \equiv_{256} I & \end{cases}$$

where α_{Cong} denotes the best abstraction of a formula with a system of congruences with a fixed modulo of 256. This derivation requires 18 iterations in all and takes 0.21s in our prototype that is integrated into the [MC]SQUARE [18, 65, 66] tool running on a 2.6GHz MacBook Pro. Note that a congruence equation such as $128\boldsymbol{r}'[i] \equiv_{256} 128\boldsymbol{r}[i] + 128\boldsymbol{s}[i]$ is equivalent to $\boldsymbol{r}'[i] \equiv_2 \boldsymbol{r}[i] + \boldsymbol{s}[i]$ which, in turn, expresses the desired exclusive-or relationship. This relationship is necessary (and sufficient) for proving that the sequence `EOR R0, R1; EOR R1, R0; EOR R0, R1` swaps the contents of `R0` and `R1`. Note too that the zero flag relationship $Z' \leftrightarrow \bigwedge_{i=0}^{7} \neg \boldsymbol{r}'[i]$ is not preserved since this cannot be expressed with congruences.

4.2. Synthesising Transfer Functions for Intervals

Providing transfer functions for intervals is almost as much of a burden as that of providing congruence relations for each of the microcontroller instructions. Techniques for synthesising transfer functions for whole blocks of instructions have recently been proposed [14, 15] in which transfer functions are realised as systems of guarded updates. Each instruction in the block is modelled as one of at most three Boolean formulae, according to whether it overflows, underflows, or does neither. The guards are derived which, if satisfied, ensure that each instruction within the block can only operate in one of its three modes. An update operation is coupled with each guard which details how input intervals are transformed to output intervals by the act of applying the code block. Our aspirations in this paper are more modest, namely to derive transfer functions for single instructions. This is simpler because it does not require reasoning about the overflow and underflow interactions between the instructions that constitute a block. In fact, a method can be deployed that is similar in spirit to the best transformer construction of Reps et al. [62], though based on dichotomic search [11, 15, 21].

To illustrate, consider the instruction `COM r` that computes the one's complement value of r and stores the result in r, where r is a generic register. Given bounds on the values of r, namely $\boldsymbol{r}_\ell$ and $\boldsymbol{r}_u$, the objective of the transfer function is to compute tight bounds on the possible values of r', namely $\boldsymbol{r}'_\ell$ and $\boldsymbol{r}'_u$. The propositional formula $[\![\texttt{COM } r]\!]$ required to synthesise the congruence relations, can be reused for this very purpose:

$$[\![\texttt{COM } r]\!] = \left\{ \begin{array}{llllll} \bigwedge_{i=0}^{7}(\boldsymbol{r}'[i] \oplus \boldsymbol{r}[i]) & \wedge & S' \leftrightarrow N \oplus V & \wedge & V' \leftrightarrow 0 & \wedge \\ N' \leftrightarrow \boldsymbol{r}'[7] & \wedge & C' \leftrightarrow 1 & \wedge & Z' \leftrightarrow \bigwedge_{i=0}^{7}(\neg \boldsymbol{r}'[i]) & \wedge \\ H' \leftrightarrow H & \wedge & T' \leftrightarrow T & \wedge & I' \leftrightarrow I & \end{array} \right.$$

Suppose $\boldsymbol{r}_\ell = 17$ and $\boldsymbol{r}_u = 39$. Then $\boldsymbol{r}'_\ell$ can be found by calling a SAT solver 8 times on the propositional formula $\psi = [\![\texttt{COM } r]\!] \wedge [\![17 \leq \boldsymbol{r}]\!] \wedge [\![\boldsymbol{r} \leq 39]\!]$ in the following fashion. First, $\psi \wedge \neg \boldsymbol{r}'[7]$ is tested for satisfiability so as to minimise the value of $\boldsymbol{r}'$ (recall that we interpret the values of a register as unsigned here). Since it is unsatisfiable, we conclude that $\boldsymbol{r}'[7]$ is always set, hence $\boldsymbol{r}'_\ell[7] = 1$. Second, $\psi \wedge \boldsymbol{r}'[7] \wedge \neg \boldsymbol{r}'[6]$ is tested for satisfiability. Since this instance is unsatisfiable, we conclude that $\boldsymbol{r}'[6]$ is always set, hence $\boldsymbol{r}'_\ell[6] = 1$. Third, $\psi \wedge \boldsymbol{r}'[7] \wedge \boldsymbol{r}'[6] \wedge \neg \boldsymbol{r}'[5]$ is checked for satisfiability. Since this is satisfiable, we conclude $\boldsymbol{r}'_\ell[5] = 0$. Fourth, $\psi \wedge \boldsymbol{r}'[7] \wedge \boldsymbol{r}'[6] \wedge \neg \boldsymbol{r}'[5] \wedge \neg \boldsymbol{r}'[4]$ is tested for satisfiability. Since this is unsatisfiable, we conclude $\boldsymbol{r}'_\ell[4] = 1$. Continuing in this manner we infer:

$$\boldsymbol{r}'_\ell \quad = \quad 1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 216$$

With an analogous chain of reasoning (where we instantiate each bit with 1 rather than 0), We infer $\boldsymbol{r}'_u = 238$. Unlike [14, 15], this approach does not finesse the need to call a solver when the transfer function is evaluated, but it is tractable for microcontroller code by virtue of the fact that registers are just 8 bits wide.

4.3. Synthesising Branching Predicates for Intervals

Thus far, we have seen how microcontroller instructions can be encoded propositionally to derive congruence systems and compute transfer functions for intervals. Yet these encodings have greater value than this. In particular they can be used to recover branching predicates from assembly code. To see this, consider the conditional jump `BRNE` at location `0x5A` of code given in Fig. 1. This branch depends on the instructions at locations `0x58` and `0x59`. `CPI R30 69` subtracts 69 from `R30` [1]; the register is not affected but C is updated to reflect the difference. In particular, C is set if `R30` is less or greater than 69. Then the instruction `CPC R31 R17` sums `R17` with C, and subtracts the result from `R31`, updating the status flags accordingly. This will set the zero flag Z if `R1` equals the sum of `R17` and C. `BRNE -5` then moves control to location `0x5B` if Z is set, and to `0x56` otherwise.

For precise interval analysis, it is necessary to synthesise predicates that express the values registers can assume following the branch. To this end, we describe a method that can infer that `R30`, for instance, takes a value that exceeds 68 if the branch is taken. As a first step, we apply a form of constant propagation (which is actually a degenerate form of interval analysis) to infer that `R17` stores the value 0 at location `0x59`. (The validity of this step is explained in Sect. 4.5.) Next, a sequence of instructions (a slice) is found that leads up to the conditional jump and influences its outcome. Such a sequence can be found by considering the control flags: `BRNE` depends on Z hence we include `CPC R31 R17` in the sequence since it effects Z. Yet `CPC R31 R17` is itself dependent on C, hence we include `CPI R30 69` since the latter instruction effects C. The behaviour of this instruction is fully determined by `R30` thus the construction terminates with the sequence `CPI R30 69; CPC R31 R17`.

This sequence is then modelling as a whole by using:

$$[\![\texttt{CPC}\ \ r\ \ s]\!] = \begin{cases} \bigwedge_{i=0}^{7}((\boldsymbol{r}'[i] \leftrightarrow \boldsymbol{r}[i]) \wedge (\boldsymbol{s}'[i] \leftrightarrow \boldsymbol{s}[i])) & \wedge \\ \bigwedge_{i=0}^{7} \boldsymbol{r}[i] \leftrightarrow (\boldsymbol{d}[i] \oplus \boldsymbol{s}[i] \oplus \boldsymbol{c}[i]) & \wedge \\ \boldsymbol{c}[0] \leftrightarrow C & \wedge \\ \bigwedge_{i=0}^{6} \boldsymbol{c}[i+1] \leftrightarrow ((\boldsymbol{d}[i] \wedge \boldsymbol{s}[i]) \vee (\boldsymbol{d}[i] \wedge \boldsymbol{c}[i]) \vee (\boldsymbol{s}[i] \wedge \boldsymbol{c}[i])) & \wedge \\ H' \leftrightarrow ((\neg \boldsymbol{r}[3] \wedge \boldsymbol{s}[3]) \vee (\boldsymbol{s}[3] \wedge \boldsymbol{d}[3]) \vee (\boldsymbol{d}[3] \vee \neg \boldsymbol{r}[3])) & \wedge \\ S' \leftrightarrow (N' \oplus V') & \wedge \\ V' \leftrightarrow ((\boldsymbol{r}[7] \wedge \neg \boldsymbol{s}[7] \wedge \neg \boldsymbol{d}[7]) \vee (\neg \boldsymbol{r}[7] \wedge \boldsymbol{s}[7] \wedge \boldsymbol{d}[7])) & \wedge \\ N' \leftrightarrow \boldsymbol{d}[7] & \wedge \\ Z' \leftrightarrow \wedge_{i=0}^{7} \neg \boldsymbol{d}[i] & \wedge \\ C' \leftrightarrow ((\neg \boldsymbol{s}[7] \wedge \boldsymbol{s}[7]) \vee (\boldsymbol{s}[7] \wedge \boldsymbol{d}[7]) \vee (\boldsymbol{d}[7] \wedge \neg \boldsymbol{r}[7])) & \wedge \\ T' \leftrightarrow T & \wedge \\ I' \leftrightarrow I \end{cases}$$

where $\boldsymbol{d}$ is a bit-vector of intermediate variables that store the difference between r and $(s + C)$ and $\boldsymbol{c}$ is a bit-vector of carrys. We omit the details for $[\![\texttt{CPI}\ \ r\ \ c]\!]$, where c is a constant, since this formula is very similar to that given above.

Renaming can be applied to construct a single formula that expresses the semantics of the sequence. The registers r and s and the constant c are instantiated to obtain encodings for `CPC R30 69` and `CPI R31 R17` which, without loss of generality, can be renamed apart so they only share the variables $H, \ldots, I$ and $H', \ldots, I'$. Renaming is also applied to construct the formula $\psi = \rho_1([\![\texttt{CPC R30 69}]\!]) \wedge \rho_2([\![\texttt{CPI R31 R17}]\!])$ where $\rho_1 = \{H' \mapsto H'', \ldots, I' \mapsto I''\}$ and $\rho_2 = \{H \mapsto H'', \ldots, I \mapsto I''\}$. Then ψ is augmented with constraints that encode any invariants found through constant propagation, thereby obtaining $\psi' = \psi \wedge (\wedge_{i=0}^{7} \neg \boldsymbol{R17}[i])$. The force of construction is that it reduces the problem of synthesising branching predicates to that of dichotomic search [11, 15, 21]. In particular, maximal and minimal values can be found for `R30` and `R31` which are consistent with the formula $\psi' \wedge Z'$. By applying dichotomic search 4 times, we conclude that if the success branch is taken $\texttt{R30} \in [69, 255]$ and $\texttt{R31} \in [0, 0]$. Conversely, by considering $\psi' \wedge \neg Z'$, we infer that if the fail branch is selected then $\texttt{R30} \in [0, 68]$ and $\texttt{R31} \in [0, 0]$.

4.4. Refining Ranges using Congruences

Interval analysis can be refined using the branching predicates derived in the previous section. However, relational information, expressed with congruences can improve the quality of range information further. To illustrate this, suppose the code fragment given in Fig. 1 is entered with $\texttt{R8} \in [100, 103]$ and $\texttt{R9} \in [0, 0]$. Then an interval analysis, using branching predicate information, can infer that the following ranges bound the values that the registers take immediately before instruction at `0x5A` is executed:

$$\texttt{R8} \in [100, 103], \quad \texttt{R9} \in [0, 0], \quad \texttt{R26} \in [0, 255], \quad \texttt{R27} \in [0, 255], \quad \texttt{R30} \in [66, 69], \quad \texttt{R31} \in [0, 0]$$

so that no useful information is inferred for `R26` and `R27`. (Note that since the ranges are small, widening need not be applied, hence there is no scope for improving over these results by, for example, delaying the application of widening.) However, by inferring congruence relationships between `R26` and other variables which are bounded, and likewise for `R27`, useful ranges can be derived for both `R26` and `R27`.

To derive the congruence relationships, we follow [45] and express the behaviour of the fragment in terms of a flowchart program $\langle \mathcal{P}, \mathcal{V}, p_0, \mathcal{T} \rangle$, where $\mathcal{P}$ denotes the set of program locations, $\mathcal{V}$ is the set of program variables, $p_0 \in \mathcal{P}$ is the initial program location, and the transition relation $\mathcal{T} \subseteq \mathcal{P} \times \mathcal{P}$ defines the possible execution of the instructions as given by the control flow graph. Consequently, for the program in Fig. 1, we have:

$$\begin{array}{lcl} \mathcal{P} & = & \{\texttt{0x50}, \ldots, \texttt{0x5B}\} \\ \mathcal{V} & = & \{\texttt{R8}, \texttt{R9}, \texttt{R17}, \texttt{R26}, \texttt{R27}, \texttt{R30}, \texttt{R31}\} \\ p_0 & = & \texttt{0x50} \\ \mathcal{T} & = & \{\langle \texttt{0x50}, \texttt{0x51} \rangle, \langle \texttt{0x51}, \texttt{0x52} \rangle, \langle \texttt{0x52}, \texttt{0x53} \rangle, \langle \texttt{0x53}, \texttt{0x54} \rangle, \langle \texttt{0x54}, \texttt{0x55} \rangle, \langle \texttt{0x55}, \texttt{0x58} \rangle, \\ & & \langle \texttt{0x58}, \texttt{0x59} \rangle, \langle \texttt{0x59}, \texttt{0x5A} \rangle, \langle \texttt{0x5A}, \texttt{0x5B} \rangle, \langle \texttt{0x5A}, \texttt{0x56} \rangle, \langle \texttt{0x56}, \texttt{0x57} \rangle, \langle \texttt{0x57}, \texttt{0x58} \rangle\} \end{array}$$

Moreover, let $c_{i,j}$ denote a congruence relation which decorates the edge $\langle p_i, p_j \rangle \in \mathcal{T}$ and describes the concrete instruction between these two locations, as explained in Sect. 4.1.

As with conventional interval analysis, invariants are expressed as a least fixed point of a system of recursive equations. For interval analysis, such invariants express the range of values that a register can assume at a given program location. However, with congruence analysis, it is possible to derive invariants which express how the values stored in registers at a given location relate to those held in the registers at location p_0. Such an analysis can be formulated [45] as the least solution to the following recursive equations:

$$\mathsf{inv}(p_0) = \bigwedge_{v \in \mathcal{V}} \left(\bigwedge_{i=0}^{7} \boldsymbol{v}'[i] \equiv_{256} \boldsymbol{v}[i] \right) \qquad \mathsf{inv}(p_j) = \sqcup_{\mathsf{Cong}} \left\{ \mathsf{inv}(p_i) \circ c_{i,j} \,\middle|\, \langle p_i, p_j \rangle \in \mathcal{T} \right\}$$

where $\mathsf{inv}(p_i)$ denotes the relationship between the initial values of registers and their values at p_i. Recall that $\sqcup_{\mathsf{Cong}}$ denotes the join operator over systems of bit-wise congruences whereas $\circ$ denotes the composition of two congruence systems, as detailed in [45]. Applying the first equation gives:

$$\mathsf{inv}(\texttt{0x50}) \quad = \quad \left\{ \begin{array}{llll} \bigwedge_{i=0}^{7} \boldsymbol{r8}'[i] \equiv_{256} \boldsymbol{r8}[i] & \wedge & \bigwedge_{i=0}^{7} \boldsymbol{r9}'[i] \equiv_{256} \boldsymbol{r9}[i] & \wedge \\ \bigwedge_{i=0}^{7} \boldsymbol{r17}'[i] \equiv_{256} \boldsymbol{r17}[i] & \wedge & \bigwedge_{i=0}^{7} \boldsymbol{r26}'[i] \equiv_{256} \boldsymbol{r26}[i] & \wedge \\ \bigwedge_{i=0}^{7} \boldsymbol{r27}'[i] \equiv_{256} \boldsymbol{r27}[i] & \wedge & \bigwedge_{i=0}^{7} \boldsymbol{r30}'[i] \equiv_{256} \boldsymbol{r30}[i] & \wedge \\ \bigwedge_{i=0}^{7} \boldsymbol{r31}'[i] \equiv_{256} \boldsymbol{r31}[i] & & & \end{array} \right.$$

Note that details of the flags have been omitted for purposes of presentation (to reduce the number of iterates required to reach the fixpoint). Then, the congruence system $c_{\langle \texttt{0x50,0x51} \rangle}$, which describes the effects of the instruction `LDI R17 0`, is applied to $\mathsf{inv}(\texttt{0x50})$ to give:

$$\mathsf{inv}(\texttt{0x51}) \quad = \quad \left\{ \begin{array}{llll} \bigwedge_{i=0}^{7} \boldsymbol{r8}'[i] \equiv_{256} \boldsymbol{r8}[i] & \wedge & \bigwedge_{i=0}^{7} \boldsymbol{r9}'[i] \equiv_{256} \boldsymbol{r9}[i] & \wedge \\ \bigwedge_{i=0}^{7} \boldsymbol{r17}'[i] \equiv_{256} 0 & \wedge & \bigwedge_{i=0}^{7} \boldsymbol{r26}'[i] \equiv_{256} \boldsymbol{r26}[i] & \wedge \\ \bigwedge_{i=0}^{7} \boldsymbol{r27}'[i] \equiv_{256} \boldsymbol{r27}[i] & \wedge & \bigwedge_{i=0}^{7} \boldsymbol{r30}'[i] \equiv_{256} \boldsymbol{r30}[i] & \wedge \\ \bigwedge_{i=0}^{7} \boldsymbol{r31}'[i] \equiv_{256} \boldsymbol{r31}[i] & & & \end{array} \right.$$

Thereafter, the congruence system for this program point is stable. Likewise, the invariants $\mathsf{inv}(\texttt{0x52})$, $\mathsf{inv}(\texttt{0x53})$, $\mathsf{inv}(\texttt{0x54})$ and $\mathsf{inv}(\texttt{0x55})$ are computed, the latter being:

$$\mathsf{inv}(\texttt{0x55}) \quad = \quad \left\{ \begin{array}{llll} \bigwedge_{i=0}^{7} \boldsymbol{r8}'[i] \equiv_{256} \boldsymbol{r8}[i] & \wedge & \bigwedge_{i=0}^{7} \boldsymbol{r9}'[i] \equiv_{256} \boldsymbol{r9}[i] & \wedge \\ \bigwedge_{i=0}^{7} \boldsymbol{r17}'[i] \equiv_{256} 0 & \wedge & \bigwedge_{i=0}^{7} \boldsymbol{r26}'[i] \equiv_{256} \boldsymbol{r8}'[i] & \wedge \\ \bigwedge_{i=0}^{7} \boldsymbol{r27}'[i] \equiv_{256} \boldsymbol{r9}'[i] & \wedge & \sum_{i=0}^{7} 2^i \boldsymbol{r30}'[i] \equiv_{256} 66 & \wedge \\ \bigwedge_{i=0}^{7} \boldsymbol{r31}'[i] \equiv_{256} 0 & & & \end{array} \right.$$

Then, incrementing the 16-bit registers `Z` in location `0x56`, we obtain:

$$\mathsf{inv}(\texttt{0x57}) \quad = \quad \left\{ \begin{array}{llll} \bigwedge_{i=0}^{7} \boldsymbol{r8}'[i] \equiv_{256} \boldsymbol{r8}[i] & \wedge & \bigwedge_{i=0}^{7} \boldsymbol{r9}'[i] \equiv_{256} \boldsymbol{r9}[i] & \wedge \\ \bigwedge_{i=0}^{7} \boldsymbol{r17}'[i] \equiv_{256} 0 & \wedge & \bigwedge_{i=0}^{7} \boldsymbol{r26}'[i] \equiv_{256} \boldsymbol{r8}'[i] & \wedge \\ \bigwedge_{i=0}^{7} \boldsymbol{r27}'[i] \equiv_{256} \boldsymbol{r9}'[i] & \wedge & \sum_{i=0}^{7} 2^i \boldsymbol{r30}'[i] - \sum_{i=0}^{7} 2^i \boldsymbol{r26}'[i] + \sum_{i=0}^{7} 2^i \boldsymbol{r8}'[i] \equiv_{256} 67 & \wedge \\ \bigwedge_{i=0}^{7} \boldsymbol{r31}'[i] \equiv_{256} 0 & & & \end{array} \right.$$

In the next step, `X` is incremented, so as to obtain:

$$\mathsf{inv}(\texttt{0x58}) \quad = \quad \left\{ \begin{array}{llll} \bigwedge_{i=0}^{7} \boldsymbol{r8}'[i] \equiv_{256} \boldsymbol{r8}[i] & & \wedge & \bigwedge_{i=0}^{7} \boldsymbol{r9}'[i] \equiv_{256} \boldsymbol{r9}[i] \quad \wedge \\ \bigwedge_{i=0}^{7} \boldsymbol{r17}'[i] \equiv_{256} 0 & & \wedge & \bigwedge_{i=0}^{7} \boldsymbol{r27}'[i] \equiv_{256} \boldsymbol{r9}'[i] \quad \wedge \\ \sum_{i=0}^{7} 2^i \boldsymbol{r30}'[i] - \sum_{i=0}^{7} 2^i \boldsymbol{r26}'[i] + \sum_{i=0}^{7} 2^i \boldsymbol{r8}'[i] \equiv_{256} 66 & & \wedge & \bigwedge_{i=0}^{7} \boldsymbol{r31}'[i] \equiv_{256} 0 \end{array} \right.$$

A further iteration is required to detect that $\mathsf{inv}(\texttt{0x58})$ is stable and hence genuinely describes a relationship between the registers at location `0x58` and those held at p_0. Note that the chain length of bit-wise linear congruences constraining n registers, hence wn propositional variables, is $w^2 n$ [55] where w is the bit-width and 2^w the modulo. Thus, in general, widening is not needed to ensure termination.

The system, however, does not specify ranges on the values of R26 and R27 which constitute the X register. Improved bounds on these registers can be found, yet again, with dichotomic search. The invariant inv(0x58) is translated into a Boolean formula denoted ψ. Then ψ is augmented with range constraints that hold at 0x58 to obtain:

$$\psi' = \psi \wedge [\![100 \leq \boldsymbol{r8}' \leq 103]\!] \wedge [\![\boldsymbol{r9}' = 0]\!] \wedge [\![66 \leq \boldsymbol{r30} \leq 99]\!] \wedge [\![\boldsymbol{r31}' = 0]\!]$$

By applying the same strategy as described in Sect. 3.3, we derive minima and maxima for $\boldsymbol{r26}'$ subject to ψ', to conclude R26 $\in [100, 107]$. Repeating this chain of reasoning for $\boldsymbol{r27}'$ we infer R27 $\in [0, 0]$ thereby showing that X never coincides with the address of a register.

4.5. Closing the loop

Recall that the verification problem which motivated the analysis was that of showing that no indirect write operations can ever have a register as its target. The astute reader may therefore be perplexed as to why the store operation at location 0x57 has, thus far, not been discussed. In particular, the correctness of the interval analysis, and even the constant propagation analysis that precedes it, depends on the indirect write operation not perturbing any register. Hence there is a dependency between using analysis to argue the correctness of the indirect write and using the indirect write to argue the correctness of the analysis.

To close this loop, recall that Kleene iteration is classically applied to compute a least fixed point in which iteration commences with an under-approximation that is successively relaxed until the least fixpoint is found. Thus one can assume that the indirect write is safe until shown otherwise (by way of contrast, indirect loads are modelled as nondeterministic assignments). The interval and congruence analyses are applied under this assumption, in effect, ignoring the indirect write. This results an under-approximation. Yet, the under-approximation coincides with an upper-approximation if every target falls within a range which ensures that no register is mutated. This follows because the set of memory locations, which are written indirectly, is disjoint from the read memory locations characterised by the approximation [13, 68]. This argument is not dissimilar to that used to resolve other so-called chicken-and-egg dependencies in binary analysis [43]. Of course, if a suitable range cannot be inferred for an indirect write, then a fault has possibly been found.

5. Transfer functions as Congruence relations

In his seminal paper on abstract interpretation using linear congruences, Granger [37] lamented the difficulty of handcrafting transformers for the congruence domain. For microcontroller code, the problem is particularly acute since there are 131 different instructions for the ATMEL ATmega16 [1] alone which need to be separately abstracted. This suggests applying techniques for computing the best transformers for congruences [45].

These automatic techniques were proposed because of the desire to simulate a whole block of instructions with a single transfer function. Since blocks are program dependent, the transfer functions can only be derived once the program is itself known, necessitating automation [14, 15, 45]. The motivation is somewhat different when deriving congruences for individual instructions; the instruction set is known up-front. The problem is simply the number and the complexity of the instructions themselves, particularly in regard to how they side-effect status flags. We do not offer any novelty with regard to the abstraction process itself over [45]; the contribution of this section is in detailing transfer functions themselves and pointing out which microcontroller operations are amenable to accurate description with bit-level congruences and which are not. Note that the transfer functions are fully detailed below so as to save the reader from the need to apply automated abstraction themselves [45].

5.1. Load and move instructions

Instructions for the ATmega series of microcontrollers have either zero, one or two operands. Apart from indirect store operations with side-effects (such as pushing a register onto the stack using PUSH R0, which implicitly decrements the stack pointer), all instructions alter a single memory location or register.

The instruction MOV r s, with parameterised registers r and s, copies the contents of the source register s into a target register r; the value stored in s is not modified. Similarly, for fixed value $c \in \mathbb{Z}_{256}$, the instruction LDI r c

loads the constant c into r. These semantics of these two operations can be expressed in an analogous way to the EOR example of Sect. 4.1 and then abstracted to obtain:

$$\alpha_{\mathsf{Cong}}([\![\texttt{LDI}\ \ r\ \ c]\!]) \quad = \quad \begin{cases} \left(\wedge_{i=0}^{7} \boldsymbol{r}'[i] \equiv_{256} \boldsymbol{c}[i]\right) \wedge \\ S' \equiv_{256} S \wedge V' \equiv_{256} V \wedge N' \equiv_{256} N \wedge Z' \equiv_{256} Z \wedge \\ C' \equiv_{256} C \wedge H' \equiv_{256} H \wedge T' \equiv_{256} T \wedge I' \equiv_{256} I \end{cases}$$

$$\alpha_{\mathsf{Cong}}([\![\texttt{MOV}\ \ r\ \ s]\!]) \quad = \quad \begin{cases} \left(\wedge_{i=0}^{7} \boldsymbol{r}'[i] \equiv_{256} \boldsymbol{s}[i]\right) \wedge \left(\wedge_{i=0}^{7} \boldsymbol{s}'[i] \equiv_{256} \boldsymbol{s}[i]\right) \wedge \\ S' \equiv_{256} S \wedge V' \equiv_{256} V \wedge N' \equiv_{256} N \wedge Z' \equiv_{256} Z \wedge \\ C' \equiv_{256} C \wedge H' \equiv_{256} H \wedge T' \equiv_{256} T \wedge I' \equiv_{256} I \end{cases}$$

The ATmega series also features operations which set or clear a given bit of a register deterministically, namely SBR and CBR, and these operations can be encoded and abstracted in a way that is similar to the above. One may be forgiven for wondering why these congruence transfer functions are derived with a modulus of 256 since a modulus of 2 is sufficient for copy and load (and several other) instructions. This is not always so, as we will see in Sect. 5.4. However, doubling the modulus only adds to the expressiveness to the relations, hence choosing the modulus to match the register length is a safe and natural choice [55]. Apart from such considerations, one should note that bit-wise linear congruences are not canonical (even when triangularisation is applied [45]). For example, the set of 0/1 solutions of $\sum_{i=0}^{7} 2^i \boldsymbol{r}'[i] \equiv_{256} \sum_{i=0}^{7} 2^i \boldsymbol{c}[i]$ is the same as that of $\wedge_{i=0}^{7} \boldsymbol{r}'[i] \equiv_{256} \boldsymbol{c}[i]$ yet the former is expressed within a single equation whereas the latter requires a system of eight congruence equations.

5.2. Logical instructions

In terms of logical bit-wise instructions, as well as exclusive-or, the ATmega16 supports bit-wise and (AND), bit-wise and with a constant/immediate value (ANDI), bit-wise complement (COM), bit-wise or (OR) and bit-wise or with a constant value (ORI). Such instructions are frequently used, since the peripherals of the microcontroller are typically addressed in a bit-wise fashion to control their mode of operation; also, instruction such as EOR R0 R0 are frequently used to set the value in R0 to 0 whilst simultaneously setting the status flags. By modelling the effects of these operations propositionally and by applying abstraction we obtain the following:

$$\alpha_{\mathsf{Cong}}([\![\texttt{AND}\ \ r\ \ s]\!]) = \begin{cases} 128S' \equiv_{256} 128N + 128V & \wedge \\ V' \equiv_{256} 0 & \wedge \\ N' \equiv_{256} \boldsymbol{r}[7] & \wedge \\ C' \equiv_{256} C \wedge H' \equiv_{256} H \wedge T' \equiv_{256} T \wedge I' \equiv_{256} I \end{cases}$$

$$\alpha_{\mathsf{Cong}}([\![\texttt{ANDI}\ \ r\ \ c]\!]) = \begin{cases} \bigwedge_{i=0}^{7} (\text{if } \boldsymbol{c}[i] = 1 \text{ then } \boldsymbol{r}'[i] \equiv_{256} \boldsymbol{r}[i] \text{ else } \boldsymbol{r}'[i] \equiv_{256} 0) & \wedge \\ 128S' \equiv_{256} 128N + 128V & \wedge \\ V' \equiv_{256} 0 & \wedge \\ N' \equiv_{256} \boldsymbol{r}[7] & \wedge \\ C' \equiv_{256} C \wedge H' \equiv_{256} H \wedge T' \equiv_{256} T \wedge I' \equiv_{256} I \end{cases}$$

$$\alpha_{\mathsf{Cong}}([\![\texttt{OR}\ \ r\ \ s]\!]) = \begin{cases} 128S' \equiv_{256} 128N + 128V & \wedge \\ V' \equiv_{256} 0 & \wedge \\ N' \equiv_{256} \boldsymbol{r}[7] & \wedge \\ C' \equiv_{256} C \wedge H' \equiv_{256} H \wedge T' \equiv_{256} T \wedge I' \equiv_{256} I \end{cases}$$

$$\alpha_{\mathsf{Cong}}([\![\texttt{ORI}\ \ r\ \ c]\!]) = \begin{cases} \bigwedge_{i=0}^{7} (\text{if } \boldsymbol{c}[i] = 1 \text{ then } \boldsymbol{r}'[i] \equiv_{256} 1 \text{ else } \boldsymbol{r}'[i] \equiv_{256} \boldsymbol{r}[i]) & \wedge \\ 128S' \equiv_{256} 128N + 128V & \wedge \\ V' \equiv_{256} 0 & \wedge \\ N' \equiv_{256} \boldsymbol{r}[7] & \wedge \\ C' \equiv_{256} C \wedge H' \equiv_{256} H \wedge T' \equiv_{256} T \wedge I' \equiv_{256} I \end{cases}$$

$$\alpha_{\mathsf{Cong}}([\![\texttt{COM}\ \ r]\!]) = \begin{cases} \bigwedge_{i=0}^{7} (128\boldsymbol{r}'[i] \equiv_{256} 128\boldsymbol{r}[i] + 128) & \wedge \\ S' \equiv_{256} 128N + 128V & \wedge \\ V' \equiv_{256} 0 & \wedge \\ N' \equiv_{256} \boldsymbol{r}'[7] & \wedge \\ C' \equiv_{256} 1 & \wedge \\ H' \equiv_{256} H \wedge T' \equiv_{256} T \wedge I' \equiv_{256} I \end{cases}$$

Note that the constant c in an instruction such as `ANDI` r c is known statically, hence it is feasible to include a conditional within the definition of a congruence relation. This formulation summarises 256 different congruence relations, one for each constant $c \in \mathbb{Z}_{256}$, each of which needs to be synthesised separately. In actuality, it is only necessary to derive congruence relations for those instructions that occur in the program, so this replication of effort is not a problem. Observe that for `AND` and `OR` congruences are too weak to express any relationship between the bit-vectors $\boldsymbol{r}'$, $\boldsymbol{r}$ and $\boldsymbol{s}$. Nevertheless, the domain is sufficiently expressive to capture other key relations, notably those of `COM` and exclusive-or relationships on the status flags.

5.3. Shifting instructions

In binary code, bit-shifts are classically used to implement multiplication or division on integers by a power of two. The ATmega16 supports five different shifts, namely arithmetic shift right (`ASR`), logical shift left (`LSL`), logical shift right (`LSR`), rotate left through carry (`ROL`) and rotate right through carry (`ROR`). All these operations shift the value of the source register into the specified direction by a single position; shifts by a variable number of positions are not directly supported (such operations are realised with a loop whose body performs shift operations). `ASR` r shifts all bits in r to the right, the most significant bit (MSB) is held constant and the least significant bit (LSB) is shifted into the carry flag C. When the register r is interpreted as a signed integer, the instruction `ASR` r effectively divides r by two without changing its sign. The instruction `LSR` r behaves analogously for unsigned values. In a similar fashion, `LSL` r multiplies r by two; it shifts the MSB into the carry flag and clears the LSB. The rotate instructions `ROL` r and `ROR` r are used to multiply and divide multi-byte signed and unsigned values by two (if two 8-bit registers are interpreted as representing a single 16-bit integer). They copy the carry flag into the LSB/MSB and shift the value of the MSB/LSB into the carry flag. The semantics of these five instructions can straightforwardly be expressed in propositional logic, from which the following systems of congruences can be derived:

$$\alpha_{\mathsf{Cong}}([\![\texttt{ASR}\ \vec{r}]\!]) = \begin{cases} \bigwedge_{i=0}^{6}(\boldsymbol{r}'[i] \equiv_{256} \boldsymbol{r}[i+1]) & \wedge \\ \boldsymbol{r}'[7] \equiv_{256} \boldsymbol{r}[7] & \wedge \\ 128S' \equiv_{256} 128N + 128V & \wedge \\ 128V' \equiv_{256} 128N' + 128C' & \wedge \\ N' \equiv_{256} \boldsymbol{r}'[7] & \wedge \\ C' \equiv_{256} \boldsymbol{r}[0] & \wedge \\ H' \equiv_{256} H & \wedge \\ T' \equiv_{256} T \wedge I' \equiv_{256} I \end{cases}$$

$$\alpha_{\mathsf{Cong}}([\![\texttt{LSL}\ \vec{r}]\!]) = \begin{cases} \bigwedge_{i=0}^{6}(\boldsymbol{r}'[i+1] \equiv_{256} \boldsymbol{r}[i]) & \wedge \\ \boldsymbol{r}'[0] \equiv_{256} 0 & \wedge \\ 128S' \equiv_{256} 128N + 128V & \wedge \\ 128V' \equiv_{256} 128N' + 128C' & \wedge \\ N' \equiv_{256} \boldsymbol{r}'[7] & \wedge \\ C' \equiv_{256} \boldsymbol{r}[7] & \wedge \\ H' \equiv_{256} \boldsymbol{r}[3] & \wedge \\ T' \equiv_{256} T \wedge I' \equiv_{256} I \end{cases} \qquad \alpha_{\mathsf{Cong}}([\![\texttt{LSR}\ \vec{r}]\!]) = \begin{cases} \bigwedge_{i=0}^{6}(\boldsymbol{r}'[i] \equiv_{256} \boldsymbol{r}[i+1]) & \wedge \\ \boldsymbol{r}'[7] \equiv_{256} 0 & \wedge \\ 128S' \equiv_{256} 128N + 128V & \wedge \\ 128V' \equiv_{256} 128N' + 128C' & \wedge \\ N' \equiv_{256} 0 & \wedge \\ C' \equiv_{256} \boldsymbol{r}[0] & \wedge \\ H' \equiv_{256} H & \wedge \\ T' \equiv_{256} T \wedge I' \equiv_{256} I \end{cases}$$

$$\alpha_{\mathsf{Cong}}([\![\texttt{ROL}\ \vec{r}]\!]) = \begin{cases} \bigwedge_{i=1}^{7}(\boldsymbol{r}'[i] \equiv_{256} \boldsymbol{r}[i-1]) & \wedge \\ \boldsymbol{r}'[0] \equiv_{256} C & \wedge \\ 128S' \equiv_{256} 128N + 128V & \wedge \\ 128V' \equiv_{256} 128N' + 128C' & \wedge \\ N' \equiv_{256} \boldsymbol{r}'[7] & \wedge \\ C' \equiv_{256} \boldsymbol{r}[7] & \wedge \\ H' \equiv_{256} \boldsymbol{r}[3] & \wedge \\ T' \equiv_{256} T \wedge I' \equiv_{256} I \end{cases} \qquad \alpha_{\mathsf{Cong}}([\![\texttt{ROR}\ \vec{r}]\!]) = \begin{cases} \bigwedge_{i=0}^{6}(\boldsymbol{r}'[i] \equiv_{256} \boldsymbol{r}[i+1]) & \wedge \\ \boldsymbol{r}'[7] \equiv_{256} C & \wedge \\ 128S' \equiv_{256} 128N + 128V & \wedge \\ 128V' \equiv_{256} 128N' + 128C' & \wedge \\ N' \equiv_{256} \boldsymbol{r}'[7] & \wedge \\ C' \equiv_{256} \boldsymbol{r}[0] & \wedge \\ H' \equiv_{256} H & \wedge \\ T' \equiv_{256} T \wedge I' \equiv_{256} I \end{cases}$$

Note these bit-level congruences are expressive enough to capture all these shift relationships, even those through the carry flag, only dropping information for the zero flag.

5.4. Arithmetic instructions

Let us now consider the arithmetic instructions: incrementing a register by 1 (`INC`), summing two registers (`ADD`) and summing up two registers and the carry flag (`ADC`). Formulae for `INC` r, `ADD` r s and `ADC` r s can be derived by expressing addition using a cascade of full-adders defined using a bit-vector of auxiliary carry bits. Importantly, the actual formulation is incidental (as long as it faithfully models addition) since the same congruence system will be generated for different but equisatisfiable formulae [45].

$$\alpha_{\mathsf{Cong}}([\![\texttt{INC}\ \vec{r}]\!]) = \begin{cases} \sum_{i=0}^{7} 2^i(\boldsymbol{r}'[i] - \boldsymbol{r}[i]) \equiv_{256} 1 & \wedge \\ S' \equiv_{256} 128N + 128V & \wedge \\ N' \equiv_{256} \boldsymbol{r}'[7] & \wedge \\ H' \equiv_{256} H \wedge T' \equiv_{256} T \wedge I' \equiv_{256} I \end{cases}$$

$$\alpha_{\mathsf{Cong}}([\![\texttt{ADD}\ \vec{r}\ \vec{s}]\!]) = \begin{cases} \sum_{i=0}^{7} 2^i(\boldsymbol{r}'[i] - \boldsymbol{r}[i] - \boldsymbol{s}[i]) \equiv_{256} 0 & \wedge \\ S' \equiv_{256} 128N + 128V & \wedge \\ N' \equiv_{256} \boldsymbol{r}'[7] & \wedge \\ T' \equiv_{256} T \wedge I' \equiv_{256} I \end{cases}$$

$$\alpha_{\mathsf{Cong}}([\![\texttt{ADC}\ \vec{r}\ \vec{s}]\!]) = \begin{cases} \sum_{i=0}^{7} 2^i(\boldsymbol{r}'[i] - \boldsymbol{r}[i] - \boldsymbol{s}[i]) \equiv_{256} C & \wedge \\ S' \equiv_{256} 128N + 128V & \wedge \\ N' \equiv_{256} \boldsymbol{r}'[7] & \wedge \\ T' \equiv_{256} T \wedge I' \equiv_{256} I \end{cases}$$

Moreover, subtraction can be expressed using the same technique by merely observing that $\sum_{i=0}^{7} 2^i \boldsymbol{r}'[i] = \sum_{i=0}^{7} 2^i \boldsymbol{r}[i] - \sum_{i=0}^{7} 2^i \boldsymbol{s}[i]$ iff $\sum_{i=0}^{7} 2^i \boldsymbol{r}[i] = \sum_{i=0}^{7} 2^i \boldsymbol{r}'[i] + \sum_{i=0}^{7} 2^i \boldsymbol{s}[i]$. This provides a propositional encoding for decrement (`DEC`), substract (`SUB`) and subtract with carry (`SBC`), by way of which the following abstractions can be derived:

$$\alpha_{\mathsf{Cong}}([\![\texttt{DEC}\ \vec{r}]\!]) = \begin{cases} \sum_{i=0}^{7} 2^i(\boldsymbol{r}[i] - \boldsymbol{r}'[i]) \equiv_{256} 1 & \wedge \\ S' \equiv_{256} 128N + 128V & \wedge \\ N' \equiv_{256} \boldsymbol{r}'[7] & \wedge \\ H' \equiv_{256} H \wedge T' \equiv_{256} T \wedge I' \equiv_{256} I \end{cases}$$

$$\alpha_{\mathsf{Cong}}([\![\texttt{SUB}\ \vec{r}\ \vec{s}]\!]) = \begin{cases} \sum_{i=0}^{7} 2^i(\boldsymbol{r}'[i] - \boldsymbol{r}[i] + \boldsymbol{s}[i]) \equiv_{256} 0 & \wedge \\ S' \equiv_{256} 128N + 128V & \wedge \\ N' \equiv_{256} \boldsymbol{r}'[7] & \wedge \\ T' \equiv_{256} T \wedge I' \equiv_{256} I \end{cases}$$

$$\alpha_{\mathsf{Cong}}([\![\texttt{SBC}\ \vec{r}\ \vec{s}]\!]) = \begin{cases} \sum_{i=0}^{7} 2^i(\boldsymbol{r}[i] - \boldsymbol{r}'[i] - \boldsymbol{s}[i]) \equiv_{256} C & \wedge \\ S' \equiv_{256} 128N + 128V & \wedge \\ N' \equiv_{256} \boldsymbol{r}'[7] & \wedge \\ T' \equiv_{256} T \wedge I' \equiv_{256} I \end{cases}$$

The above systems are rearranged for brevity, but importantly note how the power of two multipliers on the coefficients are derived automatically from the propositional encodings which make no explicit reference to these values.

5.5. Conditional branching

On the ATmega series of microcontrollers, conditional branching is typically implemented by a series of compare instructions, followed by a conditional jump which selects a jump target based on the values of the status flags. Compare instructions take two arguments and subtract the second argument from the first; in contrast to the subtract instructions, they not store the results in a register, yet set the status flags according to the value of the difference. The ATmega16 supports three kinds of compare instructions: compare two registers (`CP` r s), compare register with constant (`CPI` r c) and compare two registers with carry (`CPC` r s). The sequence `CP R0 R2; CPC R1 R3`, for example, is then used to compare the values of `R0` and `R1`, when interpreted as a single 16-bit value, against the values of `R2` and `R3`, likewise interpreted as a 16-bit value.

Conditional branching instructions such as branch-if-not-equal (`BRNE` k) only alter the program counter, which is modelled by a transition across the edge of the CFG that is traversed when calculating invariants (see Sect. 4.4). Branching instructions increment the program counter by $k + 1$ iff the branching condition is satisfied, and do not

```
0x94: LDI R24 1              ; x := 1
0x96: RJMP 1                 ; jump to 0x98
0x97: ADD R24 R24            ; x := x * 2
0x98: DEC R18                ; c := c-1
0x99: BRPL -3                ; branch if positive
0x9a: COM R24                ; x := ~x
```

Figure 2: Assembly code corresponding to the assignment `x = ~(1 << c)`

mutate any registers or status flags. In case of `BRNE` k, the program counter is incremented by $k + 1$ iff the zero flag in the status register is cleared; otherwise, it is incremented by 1. Hence, congruent transfer functions for branching instructions are given as a set of identity constraints. However, the need to precisely model the branching semantics of binary code to derive accurate invariants further motivates the automatic recovery of branching conditions (see Sect. 6) and the combination of congruences with intervals (see Sect. 7).

5.6. Experimental Results

It is worth noting that generating these transfer functions requires surprisingly few SAT instances. For example, `INC r` requires 11 SAT instances and takes 0.16s to compute the resulting congruence system. The runtime for `ADD r s` is slightly higher, requiring an overall runtime of 0.23s for 18 SAT instances; the runtimes for `ADC r s`, `SUB r s` and `SBC` vary only slightly. For `ORI r 13`, the abstraction requires 0.12s and 10 SAT instances. The results for the other instructions are all in the same order.

6. Recovering Branching Conditions

As discussed previously, branching predicates in a high-level programming language such as C are often compiled into a sequence of instructions; the successor instruction then depends on a Boolean combination of status flags. While it is possible to cover most standard predicates using pattern matching [18], some assignments are compiled into a loop due to the lack of adequate instructions that coincide with the respective assignment. As an example, consider the C statement `x = ~(1 << c)`. Since the ATmega16 does not support shifts by a variable number of bits, this statement is compiled into the code given in Fig. 2. Here, the conditional jump `BRPL` at location `0x99` depends on the status flags set by `DEC r18` at location `0x98`. The code fragment thus deviates from more commonly used patterns, which demonstrates the necessity for a generalised method for recovering branching predicates. This section describes such an algorithm based on linear template constraints [64], though it is illustrated for intervals.

6.1. Approach

The key idea of our approach is to reuse the propositional semantics discussed so far and to adapt a technique we have developed for block-wise transfer function synthesis in the domain of affine equalities [14, 15]. To make affine relations amenable to program analysis over finite bit-vectors, it is then necessary to distinguish wrapping from non-wrapping behaviour. Since affine relations cannot directly express wrap-around arithmetic [55, Sect. 1], each instruction in a block is then modelled by at least one, and at most three Boolean functions, according to whether it overflows, underflows, or does neither (called exact). The modes of each instruction can be extracted from the AVR instruction set specification. One mode from one instruction can then be combined with a mode from another, providing it is co-satisfiable, to give mode combination. In general the modes from a sequence of instructions can be combined in this way to find all the mode combinations that are mutually satisfiable. Each mode combination — of which there exist exponentially many in the worst case — is then analysed on its own.

The techniques proposed previously in [14, 15] determine guards on the inputs of the block. The guards express properties of the input registers which must hold for the block to operate in the selected mode combination. Such guards expressed over linear template constraints can be computed using incremental SAT solving [15]. We adapt this idea to the setting of branching predicate recovery and use SAT solving to compute guards on the outputs of a sequence of instructions [11, 21], which either take the success or the fail branch.

6.2. Recovering Branching Predicates by Example

Consider recovering a branching predicate for the following implementation of conditional branching:

```
0x94: CP R0 R1
0x95: BRNE 3
```

The semantics of this block is to branch to instruction `0x99` iff `R0` $\neq$ `R1`. To do so, a compare instruction in location `0x94` first computes `R0` $-$ `R1` and alters the status flags accordingly. The instruction `BRNE 3` then tests the zero flag and branches relatively to `0x95` iff the zero flag is cleared. Note that the operation `CP R0 R1` has three modes of operation, whereas the branching instruction only alters the program counter. We thus derive one guard for each of the three modes. Let $\psi(\boldsymbol{X}) = [\![\texttt{CP R0 R1}]\!]$ encode the compare instruction over bit-vectors $\boldsymbol{X} = \{\boldsymbol{r0}, \boldsymbol{r1}, \boldsymbol{a}, I, \ldots, C, I', \ldots, C'\}$ (including the effects on the status word). Suppose that $\boldsymbol{a}$ stores the result of the subtraction. The semantics of the three modes can be expressed as three Boolean formulae:

$$\begin{array}{lrcl} (1) & \varphi_U(\boldsymbol{X}) & = & \psi(\boldsymbol{X}) \wedge (\boldsymbol{r0}[7] \wedge \neg \boldsymbol{r1}[7] \wedge \neg \boldsymbol{a}[7]) \\ (2) & \varphi_O(\boldsymbol{X}) & = & \psi(\boldsymbol{X}) \wedge (\neg \boldsymbol{r0}[7] \wedge \boldsymbol{r1}[7] \wedge \boldsymbol{a}[7]) \\ (3) & \varphi_E(\boldsymbol{X}) & = & \psi(\boldsymbol{X}) \wedge (\neg \boldsymbol{r0}[7] \vee \boldsymbol{r1}[7] \vee \boldsymbol{a}[7]) \wedge (\boldsymbol{r0}[7] \vee \neg \boldsymbol{r1}[7] \vee \neg \boldsymbol{a}[7]) \end{array}$$

In what follows, we study the case where `CP R0 R1` overflows, and `BRNE 3` takes the fail branch, i.e. we consider $\varphi_O(\boldsymbol{X}) \wedge \neg Z'$. The guards we aim to compute characterise those values of `R0` and `R1` which satisfy this path. To represent these values, we turn again to intervals. We search for the least $c_\ell, c_u \in \mathbb{Z}$ such that $c_\ell \leq \texttt{R0} \leq c_u$ on output of the branching instruction; likewise for `R1`. Clearly, we have: $-128 \leq c_\ell \leq c_u \leq 127$. We can now systematically explore the space that contains c_ℓ and c_u, respectively. To do so, we divide the interval $[-128, 127]$ into two equally-sized halves, and then perform binary dichotomic search to discover which half contains the constant. We first transform the constraint $-128 \leq c_u \leq 127$ into the equivalent form:

$$(-128 \leq c_u \leq -1) \quad \vee \quad (0 \leq c_u \leq 127)$$

Since c_u is uniquely determined, testing the formula

$$\psi'(\boldsymbol{X}) \quad = \quad (\varphi_O(\boldsymbol{X}) \wedge Z') \quad \wedge \quad (0 \leq \texttt{R0} \leq 127)$$

for satisfiability suffices to determine which of the two disjuncts holds. Satisfiability of $\psi('\boldsymbol{X})$ shows that $0 \leq c_u \leq 127$. Then put:

$$\psi''(\boldsymbol{X}) \quad = \quad \psi'(\boldsymbol{X}) \quad \wedge \quad (64 \leq \texttt{R0} \leq 127)$$

Again, satisfiability of $\psi''(\boldsymbol{X})$ refines the range of c_u, in this iteration to $64 \leq c_u \leq 127$. Repeating this step 6 more times yields only satisfiable formulae, and thus $c_u = 127$. By performing this form of binary search for the remaining inequalities so as to compute c_ℓ as well as lower and upper bounds for `R1`, we obtain the following interval constraints that describe the values of `R0` and `R1` on output of the fail branch if `CP R0 R1` overflows:

$$\begin{array}{lcccl} 0 & \leq & \texttt{R0} & \leq & 127 \quad \wedge \\ -128 & \leq & \texttt{R1} & \leq & -1 \quad \wedge \end{array}$$

This form of abstraction is performed for every possible combination of modes of `CP` and `BRNE`, giving $3 \cdot 2 = 6$ systems of constraints overall. However, some of the composed formulae, e.g. $\varphi_O(\boldsymbol{X}) \wedge Z'$, are unsatisfiable. This shows that the mode combination is inconsistent, i.e. there are no inputs to the block so that `CP R0 R1` overflows and `BRNE 3` takes the success branch.

6.3. Computing extremal values using dichotomic search

Given a formula φ that constrains a register v in some way, Alg. 2 presents an algorithm for computing the maximal value of v. The register v is assumed to be signed and lines 2–8 provide special treatment for the most significant bit. Lines 11–20 represent the heart of the algorithm. Since the goal is maximisation, the algorithm instantiates each bit $v[i]$ with 1, starting with $v[k-2]$, and then checks for satisfiability. If satisfiable, the bit $v[i]$ is fixed at 1, and then the next lower bit is examined. If unsatisfiable, the bit $v[i]$ can only take the value of 0, and the algorithm moves on to maximise the next bit. Minimisation can likewise be implemented by instantiating bits the other way round. Variants of this algorithm have been reported elsewhere [11, 21]. It has also been further extended to linear templates in [15].

Algorithm 2 Compute the least value d of $v = \langle v[k-1], \ldots, v[0] \rangle$ constrained by a Boolean formulae φ using binary search where v is interpreted as a signed bit-vector

Input: φ

1: {check the sign}
2: **if** $\varphi \wedge \neg v[k-1]$ is satisfiable **then**
3: $\quad d \leftarrow 0$
4: $\quad \varphi \leftarrow \varphi \wedge \neg v[k-1]$
5: **else**
6: $\quad d \leftarrow -2^{k-1}$
7: $\quad \varphi \leftarrow \varphi \wedge v[k-1]$
8: **end if**
9: {iterate over bits $k-2, \ldots 0$}
10: **for** $i = k-2 \rightarrow 0$ **do**
11: $\quad$ **if** $\varphi \wedge v[i]$ is satisfiable **then**
12: $\quad\quad d \leftarrow d + 2^i$
13: $\quad\quad \varphi \leftarrow \varphi \wedge v[i]$
14: $\quad$ **else**
15: $\quad\quad \varphi \leftarrow \varphi \wedge \neg v[i]$
16: $\quad$ **end if**
17: **end for**
18: **return** d

6.4. Algorithm

We conclude by discussing the key steps of the algorithm, which is presented in Alg. 3. In essence, given a conditional branching instruction p_{branch}, the algorithm consists of two key steps:

Step 1: Determine dependencies We determine those instructions and registers which influence the outcome of the branching condition p_{branch}. A def-use chain π is determined using a backward search [66, Sect. 5.4] using an auxiliary method $\mathsf{revDefUseChain}(p_{\mathsf{branch}})$ in line 1. The chain is terminated once an instruction depends only on the values of registers, but not on status flags. In the previous example, instruction `BRNE 3` depends on the zero flag, which is altered by `CP R0 R1`. This instruction, in turn, does not depend on status flags; the chain thus ends.

Step 2: Abstract mode combination For the success and fail branches, we generate two formulae $[\![\pi]\!] \wedge \mu(\mathsf{true})$ and $[\![\pi]\!] \wedge \mu(\mathsf{false})$, respectively, where μ is a predicate that denotes the branching condition. Further, $[\![\pi]\!]$ encodes the semantics of the reverse def-use path π. These two formulae describe the relations for the case that the program counter is incremented by `k` + 1 or 1, respectively. Each mode combination is then analysed once for each possible branching target in lines 3–8. For the abstraction, constraints that describe the respective mode are first encoded as a formula $[\![c]\!]$. All registers which are accessed in the relevant path π are then abstracted in line 6, say, using dichotomic search for intervals as presented in Alg. 2.

Algorithm 3 Recovering the branching conditions from a branching instruction p_{branch}

1: $\pi \leftarrow \mathsf{revDefUseChain}(p_{\mathsf{branch}})$
2: $\mu \leftarrow [\![p_{\mathsf{branch}}]\!]$
3: **for** $b \in \{\mathsf{true}, \mathsf{false}\}$ **do**
4: $\quad$ **for** each mode combination $c \in \pi$ **do**
5: $\quad\quad \psi \leftarrow [\![c]\!]$
6: $\quad\quad \mathsf{abstract}([\![\pi]\!] \wedge \mu(b) \wedge \psi, \mathsf{accessedRegs}(\pi))$
7: $\quad$ **end for**
8: **end for**

Algorithm 4 Refine an interval $[v_\ell, v_u]$ of a bit-vector v with a system $A\boldsymbol{x} \equiv_m \boldsymbol{b}$ of congruences

Input: $\varphi, v, [v_\ell, v_u], A\boldsymbol{x} \equiv_m \boldsymbol{b}$

1: $\kappa \leftarrow [\![v_\ell \leq v \leq v_u]\!]$
2: $\mu \leftarrow [\![A\boldsymbol{x} \equiv_m \boldsymbol{b}]\!]$
3: $v'_\ell \leftarrow \mathsf{minimise}(\varphi \wedge \kappa \wedge \mu, v)$
4: $v'_u \leftarrow \mathsf{maximise}(\varphi \wedge \kappa \wedge \mu, v)$
5: **return** $[v'_\ell, v'_u]$

7. Reducing Abstract Descriptions

Thus far, we have described two techniques that affect two different abstract domains: (1) synthesising transformers from propositional Boolean formulae allows us to derive invariants in the domain of bit-wise congruences; (2) automatically recovering conditions for conditional branching from the program under scrutiny provides us with a technique which allows for increased precision in interval analysis of binary code. The remaining step is thus to combine congruences and intervals more tightly so as to derive more precise abstract descriptions in both domains.

7.1. Refining intervals with congruences

To make the discussion tangible we shall consider a congruence system $A\boldsymbol{x} \equiv_{16} \boldsymbol{b}$ that describes equality relations between the bits of two unsigned 4-bit registers:

$$A = \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \end{array}\right] \qquad \boldsymbol{b} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

To refer to the registers individually let $\boldsymbol{u} = \langle \boldsymbol{x}[0], \boldsymbol{x}[1], \boldsymbol{x}[2], \boldsymbol{x}[3] \rangle$ and let $\boldsymbol{v} = \langle \boldsymbol{x}[4], \boldsymbol{x}[5], \boldsymbol{x}[6], \boldsymbol{x}[7] \rangle$. Observe that the congruence system asserts that $\boldsymbol{u}[i] = \boldsymbol{v}[i]$ for all $0 \leq i < 4$. Now suppose the congruence system is paired with the range constraints:

$$2 \leq \sum_{i=0}^{3} 2^i \boldsymbol{u}[i] \leq 15 \qquad 0 \leq \sum_{i=0}^{3} 2^i \boldsymbol{v}[i] \leq 7$$

The range refinement problem is that of maximally tightening the bounds on $\boldsymbol{u}$ and $\boldsymbol{v}$ given these congruence and range constraints. Dichotomic search is the natural candidate for computing such extrema and therefore we render $A\boldsymbol{x} \equiv_{16} \boldsymbol{b}$ as a Boolean formula as follows:

$$\begin{array}{rcl} \mu & = & [\![A\boldsymbol{x} \equiv_{16} \boldsymbol{b}]\!] \\ & = & (\boldsymbol{u}[0] \leftrightarrow \boldsymbol{v}[0]) \wedge (\boldsymbol{u}[1] \leftrightarrow \boldsymbol{v}[1]) \wedge (\boldsymbol{u}[2] \leftrightarrow \boldsymbol{v}[2]) \wedge (\boldsymbol{u}[3] \leftrightarrow \boldsymbol{v}[3]) \end{array}$$

In addition, the range constraints are also reduced to a propositional system:

$$\begin{array}{rcccl} \kappa & = & [\![2 \leq \sum_{i=0}^{3} 2^i \boldsymbol{u}[i] \leq 15]\!] & \wedge & [\![0 \leq \sum_{i=0}^{3} 2^i \boldsymbol{v}[i] \leq 7]\!] \\ & = & (\boldsymbol{u}[3] \vee \boldsymbol{u}[2] \vee \boldsymbol{u}[1]) & \wedge & \neg \boldsymbol{v}[3] \end{array}$$

Applying dichotomic search to maximise $\boldsymbol{u}$ subject to $\mu \wedge \kappa'$ we infer that $\sum_{i=0}^{3} 2^i \boldsymbol{u}[i] \leq 7$. Conversely dichotomic minimisation infers $2 \leq \sum_{i=0}^{3} 2^i \boldsymbol{u}[i]$. Minimisation and maximisation of $\boldsymbol{v}$ derives the same bounds. The overall algorithm is presented in Alg. 4.

7.2. Refining congruences with intervals

These tightened bounds can then, in turn, be used to constrain the congruence system $A\boldsymbol{x} \equiv_{16} \boldsymbol{b}$. To do so let,

$$\begin{array}{rcccl} \kappa' & = & [\![2 \leq \sum_{i=0}^{3} 2^i \boldsymbol{u}[i] \leq 3]\!] & \wedge & [\![2 \leq \sum_{i=0}^{3} 2^i \boldsymbol{v}[i] \leq 3]\!] \\ & = & (\neg \boldsymbol{u}[3] \wedge \neg \boldsymbol{u}[2] \wedge \boldsymbol{u}[1]) & \wedge & (\neg \boldsymbol{v}[3] \wedge \neg \boldsymbol{v}[2] \wedge \boldsymbol{v}[1]) \end{array}$$

Then each row of the system $A\boldsymbol{x} \equiv_{16} \boldsymbol{b}$ is separately rendered as a propositional formula to be abstracted in conjunction with the propositional constraint κ' as follows:

$$\begin{array}{lcl}
\alpha_{\mathsf{Cong}}(\kappa' \wedge [\![\boldsymbol{u}[0] \equiv_{16} \boldsymbol{v}[0]]\!]) & = & (\boldsymbol{u}[0] \equiv_{16} \boldsymbol{v}[0]) \\
\alpha_{\mathsf{Cong}}(\kappa' \wedge [\![\boldsymbol{u}[1] \equiv_{16} \boldsymbol{v}[1]]\!]) & = & (\boldsymbol{u}[1] \equiv_{16} 1 \wedge \boldsymbol{v}[1] \equiv_{16} 1) \\
\alpha_{\mathsf{Cong}}(\kappa' \wedge [\![\boldsymbol{u}[2] \equiv_{16} \boldsymbol{v}[2]]\!]) & = & (\boldsymbol{u}[2] \equiv_{16} 0 \wedge \boldsymbol{v}[2] \equiv_{16} 0) \\
\alpha_{\mathsf{Cong}}(\kappa' \wedge [\![\boldsymbol{u}[3] \equiv_{16} \boldsymbol{v}[3]]\!]) & = & (\boldsymbol{u}[3] \equiv_{16} 0 \wedge \boldsymbol{v}[3] \equiv_{16} 0)
\end{array}$$

A new congruence system $A'\boldsymbol{x} \equiv_{16} \boldsymbol{b}'$ can then be reassembled from these equations like so:

$$A' = \left[\begin{array}{cccc|cccc}
1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}\right] \qquad \boldsymbol{b}' = \left[\begin{array}{c} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{array}\right]$$

to give a system $A'\boldsymbol{x} \equiv_{16} \boldsymbol{b}'$ that confers strictly less solutions than $A\boldsymbol{x} \equiv_{16} \boldsymbol{b}$. Alg. 5 presents the generalised algorithm. Here, $\mathsf{row}(i, A\boldsymbol{x} \equiv_m \boldsymbol{b})$ is an auxiliary function that delivers the i^{th} row of a congruence system $A\boldsymbol{x} \equiv_m \boldsymbol{b}$. This row is then encoded in a formula $[\![r]\!]$ and the congruence closure of $[\![r]\!]$ in conjunction with the interval constraints is computed. The resulting system is eventually put into upper triangular form. Observe that a different, straightforward tactic for refinement would be to recompute α_{Cong} based on $\varphi \wedge \kappa$. However, this is computationally more expensive: it is as expensive as computing the congruence closure of the original block augmented with interval constraints.

Algorithm 5 Refine a congruence sytem $A\boldsymbol{x} \equiv_m \boldsymbol{b}$ with a set $[v_{i,\ell}, v_{i,u}]$ of interval constraints

Input: $\{[v_{1,\ell}, v_{1,u}], \ldots, [v_{n,\ell}, v_{n,u}]\}, A\boldsymbol{x} \equiv_m \boldsymbol{b}$

1: $A'\boldsymbol{x} \equiv_m \boldsymbol{b}' \leftarrow \top$
2: $\kappa \leftarrow \bigwedge_{i=1}^{n} [\![v_{i,\ell} \leq v_i \leq v_{i,u}]\!]$
3: **for** $i = 1 \rightarrow \mathsf{numRows}(A\boldsymbol{x} \equiv_m \boldsymbol{b})$ **do**
4: $\quad r \leftarrow \mathsf{row}(i, A\boldsymbol{x} \equiv_m \boldsymbol{b})$
5: $\quad r' \leftarrow \alpha_{\mathsf{Cong}}(\kappa \wedge [\![r]\!])$
6: $\quad A'\boldsymbol{x} \equiv_m \boldsymbol{b}' \leftarrow A'\boldsymbol{x} \equiv_m \boldsymbol{b}' \wedge r'$
7: **end for**
8: **return** $\mathsf{triangular}(A'\boldsymbol{x} \equiv_m \boldsymbol{b}')$

7.3. Refining intervals with strides

As a final point on refinement, it is interesting to observe that congruences can be applied to refine a range to include stride information, that is, the greatest common divisor of the difference between consecutive elements of an ordered set of integers. For example, the stride of { 1, 3, 7, 9 } is 2. Stride information is pertinent to the analysis of binaries since it has been applied to reason about memory alignment [6]. To illustrate how stride information can be extracted, let $w = 4$ and $n = 2$ and suppose $A\boldsymbol{x} \equiv_{16} \boldsymbol{b}$ is defined as

$$A = \left[\begin{array}{cccc|cccc}
1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0
\end{array}\right] \qquad \boldsymbol{b} = \left[\begin{array}{c} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{array}\right]$$

Furthermore, let $\boldsymbol{u} = \langle \boldsymbol{x}[0], \boldsymbol{x}[1], \boldsymbol{x}[2], \boldsymbol{x}[3] \rangle$ and $\boldsymbol{v} = \langle \boldsymbol{x}[4], \boldsymbol{x}[5], \boldsymbol{x}[6], \boldsymbol{x}[7] \rangle$ as before, but now assume that the ranges are prescribed as follows:

$$0 \leq \sum_{i=0}^{3} 2^i \boldsymbol{u}[i] \leq 15 \qquad 0 \leq \sum_{i=0}^{3} 2^i \boldsymbol{v}[i] \leq 15$$

Observe that the congruence system contains equations $\boldsymbol{\nu}[0] \equiv_{16} 1$ and $\boldsymbol{\nu}[1] \equiv_{16} 0$. Thus the bottom two bits of $\boldsymbol{\nu}$ are fixed, hence $\sum_{i=0}^{3} 2^i \boldsymbol{\nu}[i]$ must draw values from the set $\{1, 5, 9, 13\}$ and therefore have a stride of 4.

8. Related Work

Defining and computing transformers for (relational) abstract domains has been a field of active research for thirty years. Classically, transfer functions have been designed by hand, though the design principles that shape transformers can be tracked back to the seminal works that founded the field [23]. The process of designing transfer functions manually, however, is not straightforward; something that has been noted in the past. As examples, consider the discussion by Cousot and Halbwachs [26] on different ways to model multiplication in the polyhedral domain, or the comments of Granger [37] on the complexity of handcrafting transformers for congruences. The difficulty of designing transfer functions can be explained by the fact that the descriptions that constitute the abstract domain are typically high-level geometric concepts, which contrasts with the low-level nature of the concrete data objects themselves; this motivates the desire to synthesise transfer functions automatically from the concrete semantics of the programming language under consideration.

8.1. Automatic abstraction

It took several decades until Reps and his colleagues [62] observed that transformers of optimal precision can be derived for any abstract domain that satisfies the finite ascending chain condition. The key idea of their work is to replace the application of an abstract transformer by a series of successive calls to a decision procedure; they employ an automatic theorem prover for this purpose. Since these calls are executed at runtime, during the analysis itself, this approach does not achieve the desired level of efficiency. Yet, their work demonstrates the existence of a procedure that generates transfer functions in a fully automatic way. Contemporaneously, the need for automated transfer function synthesis in binary code analysis was investigated by Regehr and Reid [57], following the observation that the imprecision of abstract interpretation of binary code stems, in part, from the omni-presence of both logical and arithmetic operations. They infer transfer functions for interval analysis using BDD-based Boolean encodings, though the time required to synthesise a single instruction typically exceeds 24 hours. Only recently has automatic abstraction [14, 15, 44, 45, 51, 52] become a practical proposition, due to the emergence of robust decision procedures and efficient quantifier elimination techniques [17, 47, 53]. The domain of bit-wise linear congruences [45], and the closure algorithm that is used to construct transfer functions, is an example of an analysis that particularly benefits from the efficiency of state-of-the-art SAT solvers. This algorithm is used in this paper to derive transfer functions for the instruction set of the microcontroller; it has been used elsewhere to derive relationships that hold at the word-level [14, Sect. 3.2]. The step beyond the automatic abstraction of individual instructions and straight line blocks is that of abstracting whole control structures such as loops. This is known as colloquially as loop leaping [8] or loop leaping [46] and recent advances suggest that compositionally, hence scalability, can be obtained by composing Boolean formulae in a bottom-up fashion [12].

8.2. Abstract interpretation of machine arithmetic

Over- and underflows, which are natural to programs whose semantics is defined over bit-vectors, inevitably manifest themselves within analysis. A classical approach in abstract interpretation is to perform the analysis on unbounded integers, and then check that no wraps can possibly occur, otherwise a warning message is emitted. The Astree static analyser follows this strategy [24]. However, it is important to note that Astree verifies code for a different architecture than the ATmega16 on the level of ANSI-C. In their setting, over- and underflows are undesirable (or rather disallowed). By way of comparison, wrap-arounds are commonplace in microcontroller code where many 16-bit integer operations are implemented as a composition of 8-bit operations. One approach is to deploy congruences where the modulus is 2^8 so as to precisely model the wrapping behaviour of machine arithmetic. This can be realised either on the level of words [54], where an entire register is abstracted as a whole [55], or on the level of bits [16, 44, 45]. Recently, the relative precision of these approaches has been compared [31], observing that the expressiveness of these two classes of abstract domains is strictly incomparable. Another approach is to reformulate the concretisation map for polyhedra [71] so that the most frequently occurring operations, such as evaluating a linear expression, need not require any additional machinery to support wrapping. Nevertheless, some operations, notably guards need to

be revised to reflect the effects of overflow arithmetic. A tactic that has been more recently proposed is to formulate transfer functions as guarded updates [14, 15] so that different wrapping modes can be considered in the synthesis of the transfer function and supported during its evaluation.

8.3. Static analysis of binary code

In recent years, there has been increasing interest in developing and implementing abstract interpreters that operate directly on the binary code (or an equivalent intermediate representation). Only recently, however, has infrastructure emerged to represent binary programs in interchangeable formats [9, 19, 50, 69]. These approaches, decouple the dependencies between the hardware platform and the analysed binary, at least to some extent. The most prominent representative of this class of binary code analysers is CodeSurfer/x86 [4–7] for x86 binaries. This analyser uses congruences to track strides in the value-set abstract domain, in much the same way that Astree uses congruences for ANSI-C verification [24, 25]. For example, an interval $[40, 44]$ augmented with a congruence value of 4 corresponds to the value-set $\{40, 44\}$. Congruences are also used by Debray et al. [28] who approximate addresses by sets of congruence values and thereby construct an alias analysis. However, this combination of domains is not necessarily well-suited for all applications. For example, this approach produces insufficiently precise results when applied to control-flow recovery [10, 33, 43, 59]. This is because *"there is no reason why all valid targets of a dynamic jump should follow a nice regular pattern"* [10, p. 45], hence, in this context, sets are more suitable abstractions than strided intervals [10, 59].

Model checking has also been investigated for binary code [74], based on directed proof generation [38]. The McVeto checker [74] starts with a coarse initial abstraction of the state space, which is gradually refined in parallel to disassembly. Their approach, however, is based on a possibly infinite graph representation of the state space. Furthermore, weighted pushdown automata are used, rather than classical abstraction interpretation techniques, which contrasts to the philosophy of our work. Interpolant-based approaches are also applicable to microcontroller code and for a discussion of their relationship to abstract interpretation, the reader is referred to [77].

8.4. Recovery of branching conditions

To the best of our knowledge, our approach presented in Sect. 6 is the first technique to automatically recover branching conditions from binary code. The method is influenced by our previous work on transfer function synthesis for weakly relational domains [14, 15], yet applies backwards search for a relevant path by reasoning about def-use chains. There is no reason why these techniques could not be integrated with others proposed techniques for binary code analysis, such as the work on control-flow reconstruction by Flexeder et al. [33], or the control-flow splitting method of Simon [70]. The problem of recovering branching conditions is not dissimilar to that of recovering relational information, which has been studied by Sepp et al. [69] in the context of reverse engineering.

9. Concluding Discussion

9.1. Synopsis

This paper advocates using pre-synthesised transformers for microcontroller instructions. A synthesis algorithm can be realised thanks to the bit-level semantics of the instruction set and the SAT-based congruent closure algorithm of King and Søndergaard [45]. Branching conditions are also recovered from binary code by adapting an abstraction technique originally proposed for weakly relational domains [14, 15]. This improves the precision of interval analysis. A distinguishing feature of our analysis, however, is the combination of congruences and intervals beyond the integration of branching conditions. To do so, we propose an algorithm that passes information between both components of the domains. This algorithm is based on triangularisation for refining the congruence equations and SAT solving for pruning the intervals. In summary, this combination of techniques allows us to prove memory safety for many examples of microcontroller code. Although we have described the technique for the ATmega16 microcontroller instruction set, the approach is not bound to this specific platform. We have recently ported the same technique to the programming language Instruction List for programmable logic controllers, which are frequently used in safety-critical systems. The only significant difference in the implementation is the propositional encoding of the instructions.

9.2. Experiments

Timing results for generating congruence transformers with respect to runtime and precision have been reported in [16, Sect.7]. These numbers were obtained from the [MC]SQUARE framework, which is written in JAVA, using the SAT4J solver [49]. We have applied the techniques discussed in this article to six different programs for the ATMEL ATmega16 microcontroller, ranging from 148 to 383 instructions. Details of these programs are described in [66, Sect. 7]. The technique for recovering branching conditions is an adaptation of our work presented in [15]. The runtime requirements for this step are comparable to those reported in [15, Sect. 5], being below $0.5s$ for all our benchmarks. The reverse def-use chains, which are determined prior for bit-blasting, are very short, i.e. no more than 5 instructions are involved in any of our benchmarks. Range analysis [18] by itself is frequently not sufficient to show that indirect stores are safe, as is illustrated in the example given in Sect. 1.2. However, when coupled with linear congruences and branch recovery, range analysis was able to show that no indirect store accesses could affect any register for any of our benchmarks [18, Sect. 7]. Range analysis with congruences was able to show that all indirect read accesses to program memory (e.g. to access constants), too.

9.3. Future Work

An issue that calls for future research is the handling of indirect stores. Although our analysis can effectively predict the target range (or set) of an indirect store operation, the effects of such operations are modelled as weak updates. Consequently, the invariants that express the contents of various target memory locations are merged with a commensurate loss of precision. It is more desirable to apply a strong update, which completely replaces the invariant that expresses the contents of a target with a new information. Yet it is interesting to observe that with relational domains it is possible, at least in principle, to derive a relationship between indirectly written memory locations and their contents. For instance, suppose an invariant was established between the address `X` and the contents of `R0` for the indirect store operation `ST X R0`. Then, providing the memory cell is not over-written, it follows that there is a relationship between an address and its contents, at least for some range of address values, namely those written by the store operation. More generally, we believe that recent work by Gulwani et al. [39] on lifting abstract interpreters to quantified abstract domains could serve as a general basis for this type of reasoning.

It is also interesting to note that even though bit-wise congruences cannot capture the precise semantics of logical or, they can be augmented with fresh variables to do so. Observe $\boldsymbol{r0}[i] \vee \boldsymbol{r1}[i] = (1 + \boldsymbol{r0}[i] + \boldsymbol{r1}[i]) \gg 1$, hence by introducing two fresh propositional variables to represent $(1 + \boldsymbol{r0}[i] + \boldsymbol{r1}[i])$, it is possible to derive a congruence system that precisely captures disjunction, albeit at the cost of increasing the dimension of the congruence system. It will be interesting to further study the effects of such transformations and the additional expressiveness they bring. A comparable transformation, though introduced to capture range information, has been introduced in [45, Sect. 6] using so-called witness variables that monitor whether range constraints hold or not. Similar ideas can also be found in the work of Laviron and Logozzo [48], who introduced auxiliary slack variables in the domain of subpolyhedra.

Acknowledgements. Jörg Brauer was supported, in part, by the DFG research training group 1298 *Algorithmic Synthesis of Reactive and Discrete-Continuous Systems*. Furthermore, the work of Jörg Brauer and Stefan Kowalewski was supported, in part, by the DFG Cluster of Excellence on Ultra-high Speed Information and Communication, German Research Foundation grant DFG EXC 89. Andy King was funded, in part, by a Royal Society travel grant, reference TG092357, and a Royal Society Industrial Fellowship, reference IF081178. The authors want to thank Edd Barrett, Sebastian Biallas, Chris Coppins, Bastian Schlich, Harald Søndergaard and Axel Simon for interesting discussions.

References

[1] Atmel Corporation, July 2010. 8-bit AVR Instruction Set.
URL `http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf`

[2] Bagnara, R., Dobson, K., Hill, P., Mundell, M., Zaffanella, E., 2006. Grids: A domain for analyzing the distribution of numerical values. In: Logic-Based Program Synthesis and Transformation (LOPSTR). Vol. 4407 of Lecture Notes in Computer Science. Springer, pp. 219–235.

[3] Baier, C., Katoen, J.-P., 2008. Principles of Model Checking. The MIT Press.

[4] Balakrishnan, G., August 2007. WYSINWYX: What you see is not what you execute. Ph.D. thesis, University of Wisconsin, Madison, Wisconsin, USA.

[5] Balakrishnan, G., Reps, T., Kidd, N., Lal, A., Lim, J., Melski, D., Gruian, R., Yong, S.-H., Chen, C. H., Teitelbaum, T., 2005. Model checking x86 executables with CodeSurfer/x86 and WPDS++. In: Computer Aided Verification (CAV). Vol. 3576 of Lecture Notes in Computer Science. Springer, pp. 158–163.

[6] Balakrishnan, G., Reps, T. W., 2010. WYSINWYX: What You See Is Not What You eXecute. ACM Trans. Program. Lang. Syst. 32 (6).
[7] Balakrishnan, G., Reps, T. W., Melski, D., Teitelbaum, T., 2005. WYSINWYX: What You See Is Not What You eXecute. In: VSTTE 2005. Vol. 4171 of Lecture Notes in Computer Science. Springer, pp. 202–213.
[8] Ball, T., Kupferman, O., Sagiv, M., 2012. Leaping Loops in the Presence of Abstraction. In: CAV. Vol. 4590 of Lecture Notes in Computer Science. Springer, pp. 491–503.
[9] Bardin, S., Herrmann, P., Leroux, J., Ly, O., Tabary, R., Vincent, A., 2011. The BINCOA Framework for Binary Code Analysis. In: CAV. Vol. 6806 of Lecture Notes in Computer Science. Springer, pp. 165–170.
[10] Bardin, S., Herrmann, P., Védrine, F., 2011. Refinement-based CFG reconstruction from unstructured programs. In: VMCAI. Vol. 6538 of Lecture Notes in Computer Science. Springer, pp. 54–69.
[11] Barrett, E., King, A., 2010. Range and Set Abstraction Using SAT. Electronic Notes in Theoretical Computer Science 267 (1), 17–27.
[12] Biallas, S., Brauer, J., King, A., Kowalewski, S., 2012. Loop Leaping with Closures. In: SAS. Lecture Notes in Computer Science. Springer.
[13] Bozga, M., Fernandez, J.-C., Ghirvu, L., 1999. State space reduction based on live variables analysis. In: Static Analysis Symposium (SAS). Vol. 1694 of Lecture Notes in Computer Science. Springer, pp. 164–178.
[14] Brauer, J., King, A., 2010. Automatic Abstraction for Intervals using Boolean Formulae. In: Static Analysis Symposium (SAS). Vol. 6337 of Lecture Notes in Computer Science. Springer, pp. 167–183.
[15] Brauer, J., King, A., 2011. Transfer Function Synthesis without Quantifier Elimination. In: European Symposium on Programming (ESOP). Vol. 6602 of Lecture Notes in Computer Science. Springer, pp. 97–115.
[16] Brauer, J., King, A., Kowalewski, S., 2010. Range Analysis of Microcontroller Code using Bit-level Congruences. In: Formal Methods for Industrial Critical Systems (FMICS). Vol. 6371 of Lecture Notes in Computer Science. Springer, pp. 82–98.
[17] Brauer, J., King, A., Kriener, J., 2011. Existential Quantification as Incremental SAT. In: CAV. Vol. 6806 of Lecture Notes in Computer Science. Springer, pp. 191–207.
[18] Brauer, J., Noll, T., Schlich, B., 2010. Interval Analysis of Microcontroller Code using Abstract Interpretation of Hardware and Software. In: Software and Compilers for Embedded Systems (SCOPES). ACM Press.
[19] Brumley, D., Jager, I., Avgerinos, T., Schwartz, E. J., 2011. BAP: A Binary Analysis Platform. In: CAV. Vol. 6806 of Lecture Notes in Computer Science. Springer, pp. 463–469.
[20] Clarke, E. M., Grumberg, O., Peled, D., 2001. Model Checking. MIT Press.
[21] Codish, M., Lagoon, V., Stuckey, P. J., 2008. Logic programming with Satisfiability. Theory and Practice of Logic Programming (TPLP) 8 (1), 121–128.
[22] Corbett, J. C., Dwyer, M. B., Hatcliff, J., Robby, 2000. Bandera: A source-level interface for model checking Java programs. In: International Conference on Software Engineering (ICSE). ACM Press, pp. 762–765.
[23] Cousot, P., Cousot, R., 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages (POPL). ACM Press, pp. 238–252.
[24] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Mine, A., Monniaux, D., Rival, X., 2005. The Astrée analyser. In: Programming Languages and Systems (ESOP). Vol. 3444 of Lecture Notes in Computer Science. Springer, pp. 21–30.
[25] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Mineé, A., Rival, X., 2009. Why does Astrée scale up? Formal Methods in System Design 35 (3), 229–264.
[26] Cousot, P., Halbwachs, N., 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In: Principles of Programming Languages (POPL). ACM Press, pp. 84–97.
[27] Das, M., Lerner, S., Seigle, M., 2002. ESP: Path-sensitive program verification in polynomial time. In: Programming Language Design and Implementation (PLDI). ACM Press, pp. 57–68.
[28] Debray, S., Muth, R., Weippert, M., 1998. Alias Analysis of Executable Code. In: Principles of Programming Languages (POPL). ACM Press, pp. 12–24.
[29] Eide, E., Regehr, J., 2008. Volatiles are miscompiled, and what to do about it. In: Embedded Software (EMSOFT). ACM Press, pp. 255–264.
[30] Elder, M., Lim, J., Sharma, T., Andersen, T., Reps, T., 2011. Abstract Domains of Affine Relations. In: SAS. Vol. 6887 of Lecture Notes in Computer Science. Springer, pp. 198–215.
[31] Elder, M., Lim, J., Sharma, T., Andersen, T., Reps, T. W., 2011. Abstract Domains of Affine Relations. In: SAS. Vol. 6887 of Lecture Notes in Computer Science. Springer, pp. 198–215.
[32] Engler, D. R., Chelf, B., Chou, A., Hallem, S., 2000. Checking system rules using system-specific, programmer-written compiler extensions. In: Operating Systems Design and Implementation (OSDI). ACM Press, pp. 1–16.
[33] Flexeder, A., Mihaila, B., Petter, M., Seidl, H., 2010. Interprocedural Control Flow Reconstruction. In: Asian Symposium on Programming Languages and Systems (APLAS). Vol. 6461 of Lecture Notes in Computer Science. Springer, pp. 188–203.
[34] Godefroid, P., 1990. Using partial orders to improve automatic verification methods. In: CAV. Vol. 531 of Lecture Notes in Computer Science. Springer, pp. 176–185.
[35] Gopan, D., Reps, T., Sagiv, M., 2005. A Framework for Numeric Analysis of Array Operations. In: Principles of Programming Languages (POPL). ACM Press, pp. 338–350.
[36] Granger, P., 1989. Static Analysis of Arithmetical Congruences. International Journal of Computer Mathematics 30 (13), 165–190.
[37] Granger, P., 1991. Static Analysis of Linear Congruence Equalities among Variables of a Program. In: TAPSOFT. Vol. 493 of Lecture Notes in Computer Science. Springer, pp. 169–192.
[38] Gulavani, B. S., Henzinger, T. A., Kannan, Y., Nori, A. V., Rajamani, S. K., 2006. SYNERGY: A New Algorithm for Property Checking. In: SIGSOFT FSE. ACM Press, pp. 117–127.
[39] Gulwani, S., McCloskey, B., Tiwari, A., 2008. Lifting abstract interpreters to quantified logical domains. In: Principles of Programming Languages (POPL). ACM Press, pp. 235–246.
[40] Havelund, K., Pressburger, T., 2000. Model checking Java programs using Java PathFinder. STTT 2 (4), 366–381.
[41] Huuck, R., Fehnker, A., Seefried, S., Brauer, J., 2008. Goanna: Syntactic software model checking. In: Automated Technology for Verification and Analysis (ATVA). Vol. 5311 of Lecture Notes in Computer Science. Springer, pp. 216–221.

[42] Karr, M., 1976. Affine relationships among variables of a programs. Acta Informatica 6, 133–151.
[43] Kinder, J., Veith, H., Zuleger, F., 2009. An abstract interpretation-based framework for control flow reconstruction from binaries. In: Verification, Model Checking, and Abstract Interpretation (VMCAI). Vol. 5403 of Lecture Notes in Computer Science. Springer, pp. 214–228.
[44] King, A., Søndergaard, H., 2008. Inferring Congruence Equations Using SAT. In: Computer Aided Verification (CAV). Vol. 5123 of Lecture Notes in Computer Science. Springer, pp. 281–293.
[45] King, A., Søndergaard, H., 2010. Automatic Abstraction for Congruences. In: Verification, Model Checking, and Abstract Interpretation (VMCAI). Vol. 5944 of Lecture Notes in Computer Science. Springer, pp. 281–293.
[46] Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C. M., 2008. Loop summarization using abstract transformers. In: ATVA. Vol. 5311 of Lecture Notes in Computer Science. Springer, pp. 111–125.
[47] Kroening, D., Strichman, O., 2008. Decision Procedures. Springer.
[48] Laviron, V., Logozzo, F., 2009. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In: VMCAI. Vol. 5403 of Lecture Notes in Computer Science. Springer, pp. 229–244.
[49] Le Berre, D., 2010. SAT4J: Bringing the power of SAT technology to the Java platform. `http://www.sat4j.org/`.
[50] Lim, J., Reps, T. W., 2008. A System for Generating Static Analyzers for Machine Instructions. In: CC. Vol. 4959 of Lecture Notes in Computer Science. Springer, pp. 36–52.
[51] Monniaux, D., 2009. Automatic Modular Abstractions for Linear Constraints. In: POPL. ACM Press, pp. 140–151.
[52] Monniaux, D., 2010. Automatic Modular Abstractions for Template Numerical Constraints. Logical Methods in Computer Science 6 (3).
[53] Monniaux, D., 2010. Quantifier Elimination by Lazy Model Enumeration. In: CAV. Vol. 6174 of Lecture Notes in Computer Science. Springer, pp. 585–599.
[54] Müller-Olm, M., Seidl, H., 2005. Analysis of Modular Arithmetic. In: European Symposium on Programming (ESOP). Vol. 3444 of Lecture Notes in Computer Science. Springer, pp. 46–60.
[55] Müller-Olm, M., Seidl, H., August 2007. Analysis of Modular Arithmetic. ACM Trans. Program. Lang. Syst. 29 (5).
[56] Plaisted, D. A., Greenbaum, S., September 1986. A Structure-Preserving Clause Form Translation. Journal of Symbolic Computation 2 (3), 293–304.
[57] Regehr, J., Reid, A., 2004. HOIST: A System for Automatically Deriving Static Analyzers for Embedded Systems. Operating Systems Review 38 (5), 133–143.
[58] Regehr, J., Reid, A., Webb, K., 2003. Eliminating Stack Overflow by Abstract Interpretation. In: Embedded Software (EMSOFT). pp. 306–322.
[59] Reinbacher, T., Brauer, J., 2011. Precise Control Flow Reconstruction Using Boolean Logic. In: EMSOFT. ACM Press, pp. 117–126.
[60] Reps, T., Balakrishnan, G., 2008. Improved Memory-Access Analysis for x86 Executables. In: Compiler Construction (CC). Vol. 4959 of Lecture Notes in Computer Science. Springer, pp. 16–35.
[61] Reps, T., Horwitz, S., Sagiv, M., 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In: POPL. ACM Press, pp. 49–61.
[62] Reps, T., Sagiv, M., Yorsh, G., 2004. Symbolic Implementation of the Best Transformer. In: Verification, Model Checking, and Abstract Interpretation (VMCAI). Vol. 2937 of Lecture Notes in Computer Science. Springer, pp. 252–266.
[63] Robby, Dwyer, M. B., Hatcliff, J., 2006. Bogor: A Flexible Framework for Creating Software Model Checkers. In: TAIC PART. IEEE Computer Society Press, pp. 3–22.
[64] Sankaranarayanan, S., Sipma, H., Manna, Z., 2004. Constraint Based Linear Relations Analysis. In: SAS. Vol. 3148 of Lecture Notes in Computer Science. Springer, pp. 53–68.
[65] Schlich, B., 2010. Model Checking of Software for Microcontrollers. ACM Trans. Embedded Comput. Syst. 9 (4), 1–27.
[66] Schlich, B., Brauer, J., Kowalewski, S., 2011. Application of Static Analyses for State-space Reduction to the Microcontroller Binary Code. Sci. Comput. Program. 76 (2), 100–118.
[67] Schlich, B., Kowalewski, S., 2009. Model checking C source code for embedded systems. STTT 11 (3), 187–202.
[68] Schlich, B., Noll, T., Brauer, J., Brutschy, L., 2009. Reduction of interrupt handler executions for model checking embedded software. In: Haifa Verification Conference. Vol. 6405 of Lecture Notes in Computer Science. Springer, pp. 5–20.
[69] Sepp, A., Mihaila, B., Simon, A., 2011. Precise Static Analysis of Binaries by Extracting Relational Information. In: WCRE. IEEE Digital Library, to appear.
[70] Simon, A., 2008. Splitting the Control Flow with Boolean Flags. In: Static Analysis Symposium (SAS). Vol. 5079 of Lecture Notes in Computer Science. Springer, pp. 315–331.
[71] Simon, A., King, A., 2007. Taming the Wrapping of Integer Arithmetic. In: Static Analysis Symposium (SAS). Vol. 4634 of Lecture Notes in Computer Science. Springer, pp. 121–136.
[72] Simon, A., King, A., Howe, J. M., 2010. The Two Variable Per Inequality Abstract Domain. Higher-Order and Symbolic Computation 23 (1), 87–143.
[73] Song, D. X., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., Liang, Z., Newsome, J. Poosankam, P., Saxena, P., 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In: ICISS. Vol. 5352 of Lecture Notes in Computer Science. Springer, pp. 1–25.
[74] Thakur, A. V., Lim, J., Lal, A., Burton, A., Driscoll, E., Elder, M., Andersen, T., Reps, T. W., 2010. Directed Proof Generation for Machine Code. In: CAV. Vol. 6174 of Lecture Notes in Computer Science. Springer, pp. 288–305.
[75] Tseitin, G. S., 1968. On the Complexity of Derivation in the Propositional Calculus. In: Slisenko, A. O. (Ed.), Studies in Constructive Mathematics and Mathematical Logic. Vol. Part II. pp. 115–125.
[76] Valmari, A., 1996. The state explosion problem. In: Petri Nets. Vol. 1491 of Lecture Notes in Computer Science. Springer, pp. 429–528.
[77] Weißenbacher, G., 2010. Program Analysis with Interpolants. Ph.D. thesis, Magdalen College, Oxford University, `http://www.georg.weissenbacher.name/publications.html`.
[78] Yang, X., Chen, Y., Eide, E., Regehr, J., 2011. Finding and Understanding Bugs in C Compilers. In: PLDI. ACM Press, pp. 283–294.