Science of Computer Programming 78 (2013) 862–883

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Abstract interpretation of microcontroller code: Intervals meet congruences

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

[a] *Embedded Software Laboratory, RWTH Aachen University, Germany*
[b] *Portcullis Computer Security Limited, Pinner, UK*
[c] *Verified Systems International GmbH, Bremen, Germany*

ARTICLE INFO

Article history:
Received 11 February 2011
Received in revised form 28 May 2012
Accepted 3 June 2012
Available online 21 July 2012

Keywords:
Embedded systems
Binary code
Abstract interpretation
Linear congruences
Intervals

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 accidentally mutate a register. To prove this and related properties, this article advocates using the domain of bitwise 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.

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 bitwise 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

* Corresponding author at: Embedded Software Laboratory, RWTH Aachen University, Germany. Tel.: +49 241 80 21156; fax: +49 241 80 22150.
E-mail addresses: brauer@embedded.rwth-aachen.de (J. Brauer), a.m.king@kent.ac.uk (A. King), kowalewski@embedded.rwth-aachen.de (S. Kowalewski).
URL: http://www.embedded.rwth-aachen.de (J. Brauer).

0167-6423/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2012.06.001

```
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
```

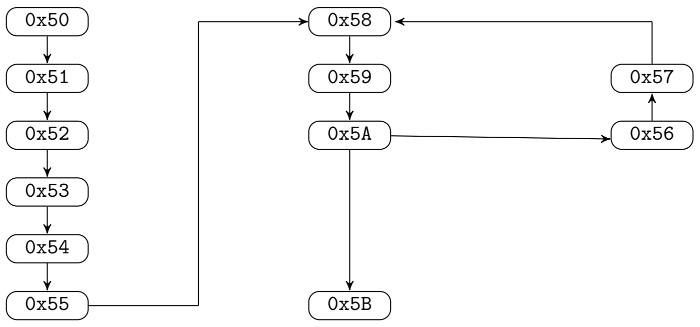

Fig. 1. `Memcopy` loop for the ATMEL ATmega16.

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.

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. Fig. 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.