Verifying Branch-Free Assembly Code in Why3

Marc Schoolderman

Radboud University, Nijmegen, The Netherlands
m.schoolderman@science.ru.nl

Abstract. This paper discusses an approach to verification of assembly code using the Why3 platform. As a case study, we prove the functional correctness of hand-optimized routines for multiplying multiprecision integers on 8-bit microcontrollers which use an efficient version of Karatsuba's algorithm. We find that by carefully constructing an underspecified model of an instruction set architecture in Why3, and specifying a few simple lemmas, verification can succeed using a high degree of automation in a short amount of time. Furthermore, our approach is sensitive to subtle memory aliasing issues, demonstrating that formal verification of security-critical assembly code is not only feasible, but also effective.

Keywords: Why3, assembly language, Karatsuba multiplication

1 Introduction

Hand-optimized assembly code is usually hard to read and reason about. However, in application areas such as cryptographic engineering, the level of control over the code generation process it offers makes it a common means of implementing primitive operations. At the same time, the correctness of such implementations is critical for their security.

Formal verification often concerns itself with programs written in a higher level programming language. In these environments, the structured nature of programs aids reasoning about their properties, for example by allowing the formulation of loop invariants. In contrast, programs written in assembly language are more unstructured in nature. In these cases, the code itself does not facilitate structural reasoning, and so the structure of a correctness proof must be inferred from the code explicitly. Second, the semantics of assembly languages are more fine-grained, requiring more primitive operations to achieve a certain result, and consequently many more steps are needed in an accompanying deductive proof of its correctness.

This paper presents an approach to verifying optimized assembly code using the Why3 verification framework [8]. We show that this framework provides the tools to tackle both challenges, by allowing us to *logically partition* unstructured code to facilitate proofs, and enabling automation to take care of most intermediate steps needed in these proofs. We discovered that stepwise refinement of proofs — aimed at minimizing the need for user-supplied assertions, thus maximizing the utility of automated provers — allows this approach to scale.

Using this approach, we have verified the multiprecision multiplication routines for the 8-bit AVR microarchitecture presented in [9], up to a 96×96-bit multiplication routine that employs Karatsuba's algorithm. We have also found that formal verification exposes a potential issue in some of the larger routines that is not likely to be detected by testing alone.

Organization of this paper. The remainder of this section will provide an overview of Why3, and the selected case study. In Section 2, we will discuss the general approach we propose for the verification of assembly programs using Why3. Section 3 shows how this approach has been applied in constructing a model for the AVR instruction set architecture, with Section 4 describing how we have subsequently used this model to verify the assembly programs we selected. Section 5 and Section 6 conclude the paper by discussing related and future work.

1.1 The Why3 Verification Platform

Why3 [8] consists of two parts: a logical specification language, with libraries for reasoning about mathematical objects (such as integers, maps and sets) and a programming language in the form of WhyML. A verification condition generator extracts proof obligations from annotated WhyML programs; these are then discharged by either automated or interactive theorem provers. This process is illustrated in Figure 1.

WhyML's primary use is as an intermediate language for verification of structured programs. However, it also has features that are useful in the context of unstructured code. In particular, its support for *abstract blocks*, *type invariants* and *bit-vector theories* turn out to be highly useful.

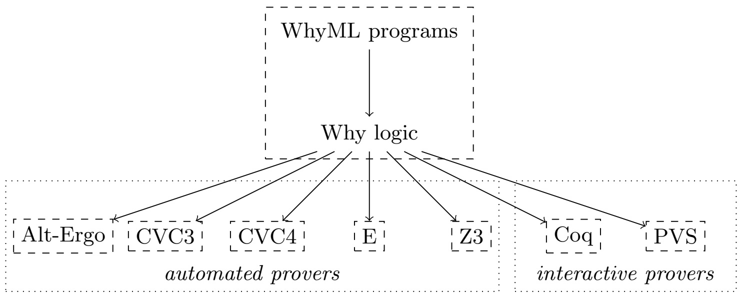

Fig. 1. Graphical representation of Why3 workflow

Using Why3 provides two major benefits. First, its verification condition generator and support for automated theorem provers allows the manual effort to focus more on *what* we need to prove, instead of *how* to prove it. Second, it allows using multiple theorem provers at no additional cost; allowing us to apply the state of the art in the field of automated (and interactive) theorem proving, and preventing a verification effort from getting hampered by the limitations of any single theorem prover.

1.2 Multiprecision Multiplication

Optimization of multiprecision arithmetic for the AVR ATmega family of 8-bit microcontrollers is an active research area, with many possible implementation strategies [12]. As a case study for this paper, we selected the hand-optimized multiplication routines developed by Hutter and Schwabe [9]. These routines currently hold the speed records for multiplication on this architecture, and are written in the context of cryptographic engineering, to serve as a primitive for efficiently implementing elliptic curve computations. This makes it an interesting target for formal verification for two reasons:

1. These routines are used as primitives in other cryptographic code [6], and so their correctness is security-critical. Bug attacks have been demonstrated to lead to practical exploits [4]. Exhaustive testing of a multiplication routine is not feasible — testing a routine which multiplies two 32-bit numbers would already mean we have to check 2^{64} cases, and random testing is known to fail to catch bugs that are triggered with low probability [5].
2. Hand-optimized assembly code is difficult to analyse; as it will contain various tricks that would not suggest themselves at a higher level, such as direct manipulation of the carry flag, and bit manipulations aimed at eliminating conditional jumps.

In fact, the programs we selected for our case study are examples of *branch-free* code. In cryptographic engineering, branch-free code is preferred to avoid side-channel attacks [11,13]. This has the effect of eliminating control flow structure from assembly programs entirely, making verification potentially even harder. In essence, branch-free code is a sequential flow program, and a correctness proof of such code is a long sequence of symbolic rewriting steps. Note that when verifying assembly code that *does* contain jumps, the first step is also to split a program into a set of branch-free fragments [14], which then need to be verified in turn. Therefore, in this paper we focus purely on this type of code.

2 Verification Approach

To verify assembly code, we need a model of the underlying instruction set architecture. Such a model consists of a representation of the machine memory (i.e., the organization of the register file, condition flags, and memory), and a set of WhyML functions corresponding to the necessary instructions. Such a model can employ underspecification, as we can safely ignore features of a microarchitecture that are not used. The usefulness of a model can be increased by building a richer logical infrastructure on top of it. For example, to allow reasoning about multiprecision integers, we introduce the function `uint` in Section 3.1.

By modelling each instruction as a single WhyML function, expressing an assembly fragment as a WhyML program conforming to our model is a simple transformation which does not require special tools.

2.1 Validation of the Formal Model

The trustworthiness of formally verified software is only as good as the tools used for verification. This paper tries to address that issue by enforcing *internal consistency* of our model. That is, we carefully avoid introducing any logical axioms in our Why3 formalization that are not present in its standard library (or supporting theorem provers).

Furthermore, since all instructions are modelled using WhyML functions, we can simply provide full implementations of these functions (in terms of WhyML primitives). This will ensure that Why3 actually checks that the postconditions are possible to satisfy, and that they are consistent with respect to any type invariants specified, which is not the case had we modelled instructions using abstract function prototypes.

This approach also allows the model to act as a bridge between the official specification of an instruction set architecture, and a Why3 specification that is most suited for reasoning about the correctness of a particular piece of code. Since we use Why3 to verify the function body of a modelled instruction with respect to its specification, this specification can be tailored to fit a certain problem domain. At the same time, the function body can be created to adhere closely to an official reference document. This can increase the confidence in the correctness of a machine model without exposing automated provers to unnecessary complexity when using this model to verify programs.

2.2 Logical Partitioning

In principle the WhyML code obtained by translating an assembly program can be verified as any other — for example, we can insert `assert` statements anywhere to indicate conditions that are needed to achieve a particular result. However, for the scalability of our approach, it is important that the demands placed on automated provers can be kept under control for larger programs.

Using Why3's `abstract` mechanism, it is possible to group any number of instructions together in an *abstract block* and specify the effect that they collectively have on the program. This will hide all computations inside the block from being seen by theorem provers. The usefulness of this strategy rests on two observations:

1. Automated provers find proofs more efficiently when presented with as little irrelevant information as necessary.
2. By grouping related operations together, their collective result can be effectively summarized (by the user) as a formula that brings us closer to the result we are trying to prove.

The benefits of logically partitioning a program in this way are maximized if we are able to verify large groups of instructions fully automatically. To achieve this, a model should be optimized (using stepwise refinement) by trying it out on small program fragments, with the aim of reducing the number of user-inserted `assert` statements needed.

3 Modelling the AVR Instruction Set

The AVR architecture is a RISC architecture, containing 32 general purpose 8-bit registers. Most instructions take two operands; with the first operand functioning both as a source and destination for the operation. Memory is only accessed via dedicated load (LD, LDD) and store (ST, STD) instructions, which require a memory address to be stored (as a 16-bit value) in one of three dedicated register pairs, denoted in mnemonic form as X for the register pair (R26,R27), Y for the pair (R28,R29) and Z for the pair (R30,R31). The register file itself is also accessible, residing at memory address 0. The instructions that are most important in our case study are instructions for adding and subtracting with and without carry (ADD, ADC, SUB, SBC) and multiplication (MUL). The MUL instruction deviates from the two-operand convention by always producing its result in the register pair (R0,R1). In Karatsuba multiplication, bitwise exclusive or (EOR) and arithmetic right shift (ASR) also play an important role, as well as the BST and BLD instructions, which transfer a single bit between a general purpose register and the T flag in the CPU status register.

We model the AVR address space in Why3 as a map of addresses to integers, where we restrict the allowed values in the range of this map to conform to those representable as an unsigned 8-bit value by adding a *type invariant*:

```
type address_space = { mutable data: map int int }
  invariant { forall i. 0 <= self.data[i] < 256 }
```

This type is embellished with some syntactic sugar for accessing elements of an address space, allowing us to write a succinct specification of the MOV instruction, which copies a value from one register to another:

```
type register = int
val reg: address_space
val mov (dst src: register): unit
  writes { reg }
  ensures { reg = old (reg[dst<-reg[src]]) }
```

Here, the notation m[*addr*<-*value*] means the address space m obtained by assigning *value* to the address *addr*, and leaving all other values unaltered.

The AVR also contains eight 1-bit status flags, of which only the carry flag and 'transfer bit' (T flag) are relevant to our case study. We model these flags as a boolean value, and provide a conversion operator to interpret them as integers:

```
type cpu_flag = { mutable value: bool }
function (?) (x: cpu_flag): int = if x.value then 1 else 0
val cf: cpu_flag
```

Reasoning about the carry flag as an integer allows for the specification of the ADD instruction as follows:

```
val add (dst src: register): unit
  writes { cf, reg }
  ensures { reg = old (reg[dst <- mod (reg[dst]+reg[src]) 256]) }
  ensures { ?cf = old (div (reg[dst]+reg[src]) 256) }
```

By specifying the result in terms of the `mod` and `div` operations, automated provers are easily able to use arithmetical theories to deduce that `?cf*256 + reg[dst]` is exactly equal to the sum of the input values.

As an alternative approach, we have also attempted to model the register file using Why3's *bit-vector theories*; as a practical benefit, this would allow us to get rid of the type invariant mentioned above. However, it appears that the additional conversion function needed to convert a register to an integer value (similar to the one needed for the 1-bit status flags) hampered the efficacy of SMT solvers.

The largest Karatsuba routine we verified also required the AVR stack to store values; we have similarly modelled this as an `address_space`:

```
val stack_pointer: ref int
val stack: address_space

let push (src: register): unit
  writes { stack, stack_pointer }
  reads { reg }
  ensures { stack = old(stack[!stack_pointer <- reg[src]]) }
  ensures { !stack_pointer = old !stack_pointer - 1 }
```

This implicitly assumes that the stack and the space for registers and memory does not overlap. In our case study (as in most code), that is something that can be easily statically verified, and so we have not focused on this property.

3.1 Representing Multiprecision Integers

Integers that are too large to be represented as a single 8-bit value are represented by multiple bytes stored consecutively in memory (or the register file), with the least significant byte occupying the lowest address. To enable reasoning about this, we define a function `uint` n A b, taken to mean the integer formed by examining the n bytes stored in address space A, starting at position b. We can define this function in Why3 as `uint` $n\ A\ b = \sum_{0 \leq i < n} 2^{8i} \cdot A[i+b]$.

Automated provers, however, work much better when given an explicit first-order expression of a multiprecision integer for a concrete value of n, for instance by fully expanding `uint 2` A b using the rule `uint 2` $A\ b = A[b] + 256 \cdot A[b+1]$. We can achieve this by adding these explicit rewrite rules as auxiliary lemmas, together with a set of `meta` directives instructing Why3 to expand matching occurrences of the `uint` function accordingly before handing a formula off to an SMT solver. This functionality is crucial, as it allows us to easily state properties about multiprecision integers in our model, while at the same time presenting them in a format that SMT solvers are able to cope with.

3.2 Using Ghost Code to Reduce Annotations

A drawback of modelling the AVR registers in combination with logical partitioning is that updates to individual registers will be hidden by the `abstract` block: even if only a single register was altered, the only thing known outside the `abstract` block is that the register file changed. This necessitates the specification of extra user-supplied annotations stating what part of the register file remained constant.

Modelling each register individually (as a `ref int`) would solve this problem, as Why3 will in this case keep of track of which registers changed inside the `abstract block`. However, this approach involves other drawbacks. For instance, Why3's type system does not allow creating an array of `ref int`, precluding the simple specification of the `uint` function of the previous section.

We solve this issue by combining both approaches, using *ghost code* [7]. In addition to modelling the register file as specified in Section 3, we also model it a second time using individual `ref int`'s, which are marked as `ghost` to prevent them from affecting the semantics of the code under verification. We can then specify as postcondition for each abstract block that the individual *ghost registers* must be equal to those in the actual register file. Ensuring that this postcondition holds is then simply done by updating only the registers that have actually been modified — information that can be obtained by a simple static analysis — in the *ghost register file*. We find that by using this trick Why3 is able to provide SMT solvers with enough information so that unnecessary annotations can be eliminated, reducing the total assertions needed by half.

3.3 Model Validation

As mentioned in Section 2.1, we are also interested in demonstrating the consistency and validity of our model. As an example of this, consider the `ADD` instruction, whose specification (as an abstract function prototype) was given in Section 3. The reference manual [1] defines the effects of the `ADD` ***`Rd, Rr`*** instruction in terms of operations on 8-bit registers, instead of the language of arithmetic:

$$\begin{aligned} \mathtt{Rd}' &\leftarrow \mathtt{Rd} +_8 \mathtt{Rs} \\ \mathtt{CF} &\leftarrow (\mathtt{Rd}_7 \wedge \mathtt{Rr}_7) \vee (\mathtt{Rr}_7 \wedge \neg\mathtt{Rd}'_7) \vee (\neg\mathtt{Rd}'_7 \wedge \mathtt{Rd}_7) \end{aligned}$$

Where $+_8$ means 8-bit addition, and $\mathtt{R}w_7$ denotes the most significant bit of $\mathtt{R}w$.

We can let the *implementation* of our model of the `ADD` instruction follow this specification closely by using Why3's *bit-vector theories*, which allow reasoning about both arithmetical and bitwise operations on bit-vectors of various sizes, and map onto the bit-vector theories of SMT solvers that support this reasoning (CVC4 and Z3). In this case, the `BV8` theory, which deals with 8-bit operations, allows the above specification to be followed closely. The function `BV8.add` can be used for performing the 8-bit addition, and the function `BV8.nth` for accessing individual bits:

```
let add (dst src: register): unit
  writes { reg, cf }
  ensures { reg = old (reg[dst <- mod (reg[dst]+reg[src]) 256]) }
  ensures { ?cf = old (div (reg[dst]+reg[src]) 256) }
= let rd  = BV8.of_int (Map.get reg.data dst) in
  let rr  = BV8.of_int (Map.get reg.data src) in
  let rd' = BV8.add rd rr in
  reg.data <- Map.set reg.data dst (BV8.to_uint rd');
  if BV8.nth rd 7 && BV8.nth rr 7 ||
     BV8.nth rd 7 && not BV8.nth rd' 7 ||
     not BV8.nth rd' 7 && BV8.nth rr 7
  then
    cf.value <- 1
  else
    cf.value <- 0
```

Using this definition, Why3 will generate proof obligations to show that the postconditions and the type invariant of the register file hold. The proof of the type invariant is immediately discharged by the prover ALT-ERGO, and the first postcondition is easily solved by CVC3. The second postcondition is more complex, but is discharged by CVC4 in less then a minute.

By constructing the model in this manner, we can be confident that our model is as internally sound as the Why3 platform itself. Furthermore, by keeping the model readable we can get a high degree of confidence that our model captures the relevant parts of the AVR instruction set architecture correctly.

It should in theory also be possible to *externally validate* the model by using Why3's code extraction feature to turn it into an executable AVR simulator, and testing that against a reference implementation. However, code extraction of programs involving bit-vectors turned out to not be possible with the version of Why3 we used. Furthermore, since AVR devices can only be re-programmed a limited number of times, such validation could be costly if we were to use actual hardware as a reference.

3.4 Underspecification

Since our case study only required a small subset of the entire AVR microarchitecture, we have used underspecification in our model in the following areas:

- We only model instructions that are actually needed in the case study.
- We have only modelled the flags that are actually needed; since none of the other flags are ever used as inputs during any of the computations.
- We have modelled the register file and memory contents as two separate address spaces, and disallow accessing the register file in the `LD` and `ST` instructions, since the code we verify does not use this feature.
- We assume that there is enough room on the stack for the `PUSH` and `POP` instructions to work safely.

To properly handle loads and stores to and from memory, it is also necessary to know what constitutes a valid address for accessing SRAM. However, the allowed range of addresses is specific to each type of AVR microcontroller, and so we cannot verify this in general. A possible approach would be two specify two abstract constants in our model, `ram_begin` and `ram_end`, and add as a precondition to the `LD` and `ST` instructions that addresses must fall inside the interval [`ram_begin`, `ram_end`).

However, in our case study, memory is only accessed using addresses supplied by the caller, and so we have no choice but to assume that these addresses are correct anyway. Therefore we have not added an address sanity check in the current model.

4 Verifying AVR Assembly Code

The target of our verification effort was the *branch-free* multiplication routines presented by Hutter and Schwabe [9]. These consist of routines using a quadratic complexity "schoolbook" method called *operand-scanning* for integer sizes from 24 bits to 48 bits, as well as routines using the subquadratic Karatsuba method for multiplying integers of sizes from 48 bits to 256 bits.

For verification, we targeted all "schoolbook" routines for multiplying integers of 48 bits and less, as well as the routines using only one application of Karatsuba's method, which are all the routines for argument sizes of 96 bits and less. The routines for larger argument sizes require multiple (recursive) applications of Karatsuba's method. These do seem to be in reach of our approach, but we have not yet finished verification of these versions.

4.1 Operand-Scanning Multiplication

The operand-scanning multiplication can be summarized as follows. Let $A = \sum_{0 \le i < n} 2^{8i} \cdot a_i$ and $B = \sum_{0 \le i < n} 2^{8i} \cdot b_i$. Their product can be computed as:

$$A \cdot B = \sum_{0 \le i,j < n} 2^{8(i+j)} \cdot a_i \cdot b_j$$

This can be implemented as an algorithm which iterates over the operands a_i and b_j, repeatedly multiplying and adding. This is exactly how the operand-scanning algorithms presented in [9] work. As an example, we show an AVR assembly version for multiplying two 16-bit values in Figure 2, where A is stored in the register pair (`R2`,`R3`), and B is in the pair (`R7`,`R8`), with the result rendered in the registers (`R12`,`R13`,`R14`,`R15`). This algorithm first computes $2^{16} \cdot a_1 b_1 + a_0 b_0$, and then adds in $2^8 \cdot a_0 b_1$ and $2^8 \cdot a_1 b_0$ using two sequences consisting of `ADD` and `ADC` instructions.[1]

During verification, it appears the difficult part in the code of Figure 2 is showing that after the last `ADC` in each such sequence, the carry flag is guaranteed

[1] As this version was not included in [9], we implemented it ourselves

```
CLR R23
MUL R3, R8
MOVW R14, R0
MUL R2, R7
MOVW R12, R0
MUL R2, R8
ADD R13, R0
ADC R14, R1
ADC R15, R23
MUL R3, R7
ADD R13, R0
ADC R14, R1
ADC R15, R23
```

```
let mul16()
ensures {
  uint 4 reg 12 = old(uint 2 reg 2*uint 2 reg 7)
}
= clr r23;
  mul r3 r8;
  movw r14  r0;
  mul r2 r7;
  movw r12 r0;
  mul r2 r8;
  add r13 r0;
  adc r14 r1;
  adc r15 r23;
  mul r3 r7;
  add r13 r0;
  adc r14 r1;
  adc r15 r23
```

Fig. 2. 16×16 bit multiplication in AVR assembly (left) and WhyML (right)

to be 0. At the end of the first sequence, this can still be easily manually asserted and automatically proven (by the prover Z3), since the pair (R14,R15) contained at most the value $255 \cdot 255 = 254 \cdot 2^8 + 1$ before this sequence is executed, meaning that R15 contains at most the value 254. However, to prove that the second sequence does not result in a carry requires showing that a product of two 16-bit values fits in 32-bits. Manual attempts at showing this for larger routines with many sequences of ADD and ADC instructions resulted in a substantive amount of assertions, initially slowing down the verification effort. However, by iterative improvements of the proof for the small routine of Figure 2, we discovered that adding a version of the aforementioned fact as a lemma in Why3 is beneficial:

Lemma *Let m be a map from addresses to integers, where for each address i we have $0 \leq m[i] < 256$. Then for all i, j we have $0 \leq m[i] \cdot m[j] \leq 255 \cdot 255$.*

Note that the condition in this lemma matches the type invariant in the definition of an address_space in Section 3 closely. Using this lemma, the prover CVC4 can automatically prove the code in Figure 2 to be correct; that is, in the WhyML translation, the postcondition `uint 4 reg 12 = old(uint 2 reg 2*uint 2 reg 7)` is automatically verified in less than a second. The lemma itself is instantly proven by the prover Alt-Ergo.

More importantly, it turns out that this single lemma, together with the rewrite rules for the uint function mention in Section 3.1, are enough to prove the correctness of *all* other operand-scanning algorithms presented in [9] in a short amount of time, as shown in Table 1. This demonstrates that if we optimize proofs on small examples, automated provers can be successfully utilized to verify scaled-up versions without requiring further user intervention.

4.2 Karatsuba Multiplication

Starting at 48-bit multiplications, *subtractive Karatsuba multiplication* is found to be more efficient on the AVR than operand-scanning multiplication. In this case, the process for multiplying two n-bit integers A and B, looks as follows (slightly adapted from [9]):

- Write $A = 2^{n/2}A_h + A_l$, and $B = 2^{n/2}B_h + B_l$
- Let $L = A_l \cdot B_l$
- Let $H = A_h \cdot B_h$
- Let $M = |A_l - A_h| \cdot |B_l - B_h|$
- If $A_l \geq A_h \iff B_l \geq B_h$, obtain the result as

$$A \cdot B = L + 2^{n/2}(L + H - M) + 2^n H$$

- Otherwise, obtain the result as

$$A \cdot B = L + 2^{n/2}(L + H + M) + 2^n H$$

Note that to implement this algorithm as a strictly sequential program, some assembly tricks are needed. In particular, to obtain the result, and in the computation of M, a value needs to be conditionally negated. To achieve that without using a conditional branch, the assembly program uses the fact that, in two's complement representation, $-x$ is equal to $\overline{x} + 1 = (x \oplus \overline{0}) - \overline{0}$, where $\overline{v}$ denotes the bitwise complement of v, and $\oplus$ the bitwise exclusive or. Accordingly, when computing $A_l - A_h$ the program stores the resultant carry flag in a register using the `SBC Rd, Rd` instruction. This produces a value w in the register `Rd`, which will be $\overline{0}$ if the subtraction generated a carry, and 0 otherwise. And because a carry is only generated if $A_l < A_h$, we then have that for any value x, $(x \oplus w) - w$ is equal to $-x$ if $A_l < A_h$, and equal to x otherwise.

Partitioning For verifying the Karatsuba implementations, we have used logical partitioning of the actual assembly program into *abstract blocks* by identifying (roughly) the following distinct computational steps in the implementation:

1. The computation of $L = A_l \cdot B_l$, using a *operand-scanning* multiplication.
2. The computation of $|A_l - A_h|$ and $|B_l - B_h|$.
3. A multiply-add step, which computes the product $H = A_h \cdot B_h$, and adds this to L to create $L + 2^{n/2}H$.
4. The computation of M by multiplying $|A_l - A_h|$ and $|B_l - B_h|$.
5. A processing step which prepares a bitmask $w = 0$ or $w = \overline{0}$ to control whether M will be negated in the next step, and which computes $(2^{n/2} + 1) \cdot (L + 2^{n/2}H) = L + 2^{n/2}(L + H) + 2^n H$.
6. Obtaining either M or $-M$ by computing $(M \oplus w) - w$.
7. Adding $2^{n/2}M$ or $-2^{n/2}M$ to the value from step 5 to obtain the desired result.

The steps mentioned are (in principle) documented in [9], which describes their design process, and they are also indicated by comments in the original source code. This structure is shared by all Karatsuba implementations (up to the 96×96-bit version). The main differences between these lie in the order in which data is loaded from memory into the available registers, and where the data is stored 'in flight'. For example, the 48×48-bit version does not use the 'T' flag, and only the 96×96-bit implementation used the CPU stack to store values. This did not alter the choice of partitioning. As an illustration, the WhyML code of the abstract block for step 2 for the 64×64-bit multiplication routine is shown in Figure 3. The `modify_r`N statements at the end of this block indicate which registers are modified by the code inside it, and are used to update the *ghost registers*, as explained in Section 3.2.

In general, if assembly code is hand-written (or generated using a simple process), such information is expected to be available for complicated routines, and should be sufficient to inform a partitioning of it into abstract blocks. If assembly code is written with verification in mind, it should also be possible to instruct programmers to indicate potential blocks, or even to employ verification already during the development process. On the other hand, it is probably much harder to find a useful partitioning in code emitted by an optimizing compiler.

Lemmas Needed As in Section 4.1, some simple lemmas were needed to help along the automated verification of some of the resultant abstract blocks. In the case of Karatsuba's routine, these concerned themselves primarily with the bit manipulation involved in computing absolute values, as explained in the previous section. For instance, a lemma was needed for the fact that $w \oplus 0 = w$, and $w \oplus \overline{0} = 255 - w$, where w is an 8-bit vector.

Ordering of Load and Stores As opposed to the operand-scanning routines, which multiply values already stored in the register file, the Karatsuba multiplication routine reads its input from SRAM (under control of the register pairs `X` and `Y`), and writes its result back to SRAM as well (under control of the register pair `Z`). In order to deal with register pressure, some of these loads and stores happen after some computation has already been performed. In particular, the lower three bytes of L are committed to memory before the values needed to compute H are loaded. In the 48-bit implementation, this happens in the following fragment:

```
STD Z+0, R8
STD Z+1, R9
STD Z+2, R10
LD R14, X+
LD R15, X+
LD R16, X+
LDD R17, Y+3
LDD R18, Y+4
LDD R19, Y+5
```

```
abstract
ensures { synchronized shadow reg }
ensures { uint 4 reg 2 = old (abs (uint 4 reg 2 - uint 4 reg 18)) }
ensures { uint 4 reg 6 = old (abs (uint 4 reg 6 - uint 4 reg 22)) }
ensures { ?tf = 0 <-> old((uint 4 reg 2 < uint 4 reg 18) <->
                          (uint 4 reg 6 < uint 4 reg 22)) }
  sub r2 r18;
  sbc r3 r19;
  sbc r4 r20;
  sbc r5 r21;
  sbc r0 r0;

  sub r6 r22;
  sbc r7 r23;
  sbc r8 r24;
  sbc r9 r25;
  sbc r1 r1;

  eor r2 r0;
  eor r3 r0;
  eor r4 r0;
  eor r5 r0;
  eor r6 r1;
  eor r7 r1;
  eor r8 r1;
  eor r9 r1;

  sub r2 r0;
  sbc r3 r0;
  sbc r4 r0;
  sbc r5 r0;
  sub r6 r1;
  sbc r7 r1;
  sbc r8 r1;
  sbc r9 r1;

  eor r0 r1;
  bst r0 0;
modify_r0(); modify_r1(); modify_r2(); modify_r3(); modify_r4();
modify_r5(); modify_r6(); modify_r7(); modify_r8(); modify_r9();
end;
```

Fig. 3. Abstract block for computing $|A_l - A_h|$ and $|B_l - B_h|$, where A_l is stored in the registers R2,R3,R4,R5, A_h is stored in the registers R18,R19,R20,R21; B_l is stored in the registers R6,R7,R8,R9 and B_h is stored in the registers R22,R23,R24,R25.

This code fragment caused several of the assertions relating the original memory contents to the contents of the register file to fail — the reason for this is that it cannot be proven that the memory contents read by the load instructions (`LD` and `LDD`) is not altered by the store (`STD`) instructions. This would indeed cause an invalid computation in the unlikely case where the memory pointed to by `Z` overlaps with the upper three bytes of either of the input values.

To prohibit this situations and allow the entire Karatsuba routine to be verified, preconditions are necessary to ensure the memory separation of the input and outputs, which take the form:

$$\texttt{uint 2 reg Z} + m \leq \texttt{uint 2 reg X} \vee \texttt{uint 2 reg Z} \geq \texttt{uint 2 reg X} + n$$
$$\texttt{uint 2 reg Z} + m \leq \texttt{uint 2 reg Y} \vee \texttt{uint 2 reg Z} \geq \texttt{uint 2 reg Y} + n$$

However, in the case of the 48-bit and 64-bit implementations, it was straightforward to rearrange the code fragment so that the `STD` instructions happen after the `LD` instructions, removing the problem (and the need for any special precondition) altogether. In the 80-bit implementation, improving the implementation in this manner is not possible due to the increased register pressure, and so the preconditions (with $m = 0$ and $n = 10$) were necessary. For the 96-bit implementation, this separation precondition was even more restrictive ($m = 6$ and $n = 12$), prohibiting most forms of aliasing input and output memory locations.

This leads to the observation that these routines are not perfect drop-in replacement for each-other: whereas the 80-bit (or smaller) implementations can safely be called if the output overlaps with either of the inputs, the 96-bit cannot. This could be a problem if the implementations are not used properly. While we do not consider this to be a bug in the original code, it does show that formal verification using our technique is indeed able to catch subtle issues like these in real-world complex assembly code.

Discovery of this problem was guided by automated provers and Why3: when several assertions fail to verify within a reasonable time frame, one approach is to insert more assertions in order to find the exact step that automated provers have difficulties with, and see what can be done to address this. In this case, this led to the discovery that the code did not satisfy the assertion.

Verification Results Using the partitioned and annotated WhyML program as input, we can use Why3 to generate verification conditions. All proof obligations that were generated, as well as the lemmas we needed, could be proven by a combination of the SMT solvers CVC3 and CVC4. In a handful of cases the automated E theorem prover was needed. As shown in Table 1, many of the smaller routines could be verified in less than a minute. We have also listed the number of annotations that had to be supplied by the user, in the form of `assert` statement or `ensures`-clauses for the abstract blocks described in Section 4.2. This count also includes the desired post-condition for the entire function, and any possible `requires`-clauses needed as described in Section 4.2.

Due to the growing complexity of the verification conditions, the total verification time does grow dramatically, especially for the 96×96 multiplication routine. On the other hand, the number of required annotations required per line of code actually decreases. It should be noted that we have also found that the verification time of the 96×96-bit routine can be brought down by adding extra annotations — however, we prefer to minimize the number of user-annotations required, since we have found these to be the largest bottleneck in verification.

5 Related Work

An early work on verifying machine code using an automated prover is given by Yu [16], who used the Nqthm theorem prover to produce formal proofs about object code generated for the Motorola 68020 microprocessor.

Pereira and Sousa [14] have used WhyML as an intermediate language for the verification of ARM code. Their work focuses on handling the unstructured control flow of assembly language: they have written the ARMY tool that splits an assembly language program containing jumps into a series of purely sequential basic blocks, where the user specifies pre- and postconditions for each block, and then hands these off to Why3. They also provide a cost model to enable complexity analysis. This work is therefore complementary to ours, since we focus on the effective verification of large purely sequential programs that are supposed to run in constant time. In essence we show that for verification, assembly programs can benefit from being split into more basic blocks than strictly necessary in their approach.

Chen et al. [5] have verified a *Montgomery ladder step* of an elliptic curve computation for the x86-64. architecture. The most complex step of this computation is a 256×256-bit multiplication, which is verified by translating (using a special-purpose converter) annotated QHASM [3] code to input for the SMT solver BOOLECTOR. Their approach appears to require more user intervention (in the form of annotations) to perform the actual proofs, as well as more com-

	program size	*annotations*	*verification time*	*provers used*
operand scanning				
16×16	13 lines	1	0s	CVC4
24×24	33 lines	1	1s	CVC4
32×32	59 lines	1	4s	CVC4
40×40	93 lines	1	10s	CVC4
48×48	136 lines	1	33s	CVC4
Karatsuba				
48×48	169 lines	23	52s	CVC4, CVC3, E
64×64	286 lines	21	96s	CVC4, CVC3, E
80×80	411 lines	31	215s	CVC4, CVC3, E
96×96	611 lines	39	906s	CVC4, CVC3, E

Table 1. Statistics of verifying optimized AVR multiplication routines

putational resources — requiring more than an hour of computational effort. On a 64-bit machine, a 256×256-bit multiplication is equivalent (in terms of instruction complexity) to a 32×32-bit multiplication on the 8-bit AVR, which we are able to verify using modest resources and without annotations. On the other hand, the code they verify is more extensive and diverse, as they also verify that their code performs other operations (such as an efficient reduction modulo $2^{255} - 19$), making the results hard to compare directly.

Schwabe and Schmaltz [15] first attempted to verify the 48-bit multiplication routines examined in this paper using ACL2 in combination with external SAT solvers. However, their approach focused on proving functional equivalence between two implementations, instead of proving correctness with respect to a simple specification, and required much more CPU time than the approach presented in this paper.

Branch-free assembly code provides many more opportunities for formal verification. For instance, Barthe et al. [2] present a type system to mark certain data as confidential, and provide an analysis that can be used to show that the control flow and sequences of memory accesses of programs does not depend on this confidential data. They also prove that x86 programs that are constant-time in this sense are safe from cache-based attacks, using the Coq interactive theorem prover.

6 Conclusion

The main contribution of this paper is the fact that by finding the right balance between manual verification and proof automation, verification of large and optimized assembly code is feasible. The Why platform offers the machinery for striking this balance. In particular, the key to using automated provers to prove large and complex properties efficiently is the ability to not only *summarize*, but also *prune* the proof context. Why3's `abstract` blocks provides a significant degree of control over this context.

Another advantage of using an off-the-shelf verification platform such as Why3 is that we have a higher confidence in the correctness of our result, as it can be proven to be internally consistent, assuming that Why3 itself is consistent. The validity of our model is also more easily examined than in the approach taken in [5].

The manual effort involved in our approach concerns itself with imposing a structure on an assembly language program by partitioning it into blocks, controlling the information that will be presented to an automated prover. This requires knowledge that is not necessarily easily deduced from the source code of an assembly program. The best way to reduce this manual effort would be to have programmers annotate assembly programs while writing them, indicating which groups of instructions form a block and specifying the pre- and postconditions of such a block.

6.1 Future Work

We would like to try our approach on even larger examples; the larger multiple-level Karatsuba routines presented in [9], up to 256 bits, are an obvious first choice. To verify assembly programs that contain control flow, combining our approach with the sequentialization technique of [14] seems an obvious next step, and should not pose much difficulties.

We are also interested in trying out the approach for other forms of unstructured code, such as the portable assembly language implemented in QHASM [3] or the Instruction List language used in programmable logic controllers. In these latter two cases, we expect the challenge to mainly consists of constructing a model that can be both validated, and has the right expressivity so that it is suitable for many different applications.

Finally, an interesting research question remains with the respect of identifying an appropriate *logical partitioning*. We expect that determining the point at which some proof context becomes irrelevant should be detectable by some automatic means. Why3 already has some support for automatically pruning a proof context in the poorly documented *bisect* feature [10], but this is computationally expensive and needs to be repeated for every individual verification condition.

Availability of code The Why3 code used in this paper can be obtained in full at `https://gitlab.science.ru.nl/sovereign/why3-avr`.

Acknowledgement This work is part of the research programme 'Sovereign' with project number 14319 which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO).

References

1. Atmel Corporation: AVR Instruction Set Manual, revision 0856L (2016)
2. Barthe, G., Betarte, G., Campo, J., Luna, C., Pichardie, D.: System-level non-interference for constant-time cryptography. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 1267–1279. CCS '14, ACM, New York, NY, USA (2014)
3. Bernstein, D.J.: qhasm: tools to help write high-speed software, `http://cr.yp.to/qhasm.html` (accessed 2016-12-01)
4. Brumley, B.B., Barbosa, M., Page, D., Vercauteren, F.: Practical realisation and elimination of an ECC-related software bug attack. In: Proceedings of the 12th Conference on Topics in Cryptology. pp. 171–186. CT-RSA'12, Springer-Verlag, Berlin, Heidelberg (2012)
5. Chen, Y.F., Hsu, C.H., Lin, H.H., Schwabe, P., Tsai, M.H., Wang, B.Y., Yang, B.Y., Yang, S.Y.: Verifying Curve25519 software. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS'14. pp. 299–309. ACM (2014)
6. Düll, M., Haase, B., Hinterwälder, G., Hutter, M., Paar, C., Sánchez, A.H., Schwabe, P.: High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. Des. Codes Cryptography 77(2-3), 493–514 (Dec 2015)
7. Filliâtre, J.C., Gondelman, L., Paskevich, A.: The spirit of ghost code. In: Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559. pp. 1–16. Springer-Verlag New York, Inc., New York, NY, USA (2014)
8. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Proceedings of the 22nd European Symposium on Programming. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (2013)
9. Hutter, M., Schwabe, P.: Multiprecision multiplication on AVR revisited. Journal of Cryptographic Engineering 5(3), 201–214 (2015)
10. Jackson, P., Schanda, F., Wallenburg, A.: Auditing User-Provided Axioms in Software Verification Conditions, pp. 154–168. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
11. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology. pp. 388–397. CRYPTO '99, Springer-Verlag, London, UK (1999)
12. Liu, Z., Seo, H., Großschädl, J., Kim, H.: Reverse Product-Scanning Multiplication and Squaring on 8-Bit AVR Processors, pp. 158–175. Springer International Publishing, Cham (2015)
13. Molnar, D., Piotrowski, M., Schultz, D., Wagner, D.: The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks, pp. 156–168. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
14. Pereira, M., de Sousa, S.a.M.: Complexity checking of ARM programs, by deduction. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing. pp. 1309–1314. SAC '14, ACM, New York, NY, USA (2014)
15. Schwabe, P., Schmaltz, J.: Verification of optimised 48-bit multiplications on AVR. Tech. rep., Radboud University (2015)
16. Yu, Y.: Automated proofs of object code for a widely used microprocessor. Tech. rep., University of Texas at Austin, Austin, TX, USA (1993)