Vx86: x86 Assembler Simulated in C Powered by Automated Theorem Proving

Stefan Maus[1], Michał Moskal[2], and Wolfram Schulte[3]

[1] Universität Freiburg, Freiburg, Germany
[2] European Microsoft Innovation Center, Aachen, Germany
[3] Microsoft Research, Redmond, WA, USA

Abstract. Vx86 is the first static analyzer for sequential Intel x86 assembler code using automated deductive verification. It proves the correctness of assembler code against function contracts, which are expressed in terms of pre-, post-, and frame conditions using first-order predicates. Vx86 takes the annotated assembler code, translates it into C code simulating the processor, and then uses an existing C verifier to either prove the correctness of the assembler program or find errors in it. First experiments on applying Vx86 on the Windows Hypervisor code base are encouraging. Vx86 verified the Windows Hypervisor's memory safety, arithmetic safety, call safety and interrupt safety.

1 Introduction

The correctness of operating systems is critical for the security and reliability of any computer system. However, debugging and testing operating systems is very difficult: kernel operations are hard to monitor, and algorithms are highly optimized and often concurrent. These factors suggest that one should verify operating systems. In fact, there are currently several projects that try to do just that [21,15,10,19]. However, such projects still leave us far from a practical methodology for verifying real systems.

One gap is in the verification targets. Existing verification projects often use idealistic sequential code written in clean programming languages. In contrast, modern system code is typically multithreaded, racy, written in C *and* assembler. Assembler is used (1) to access special instructions that are not available in C (like *CPUID*, which returns some important properties of the processor), and (2) to improve the performance of critical algorithms like interrupt dispatch, context switch, clearing pages, etc. While several verifiers for C exist [11,14,17], we think that it is imperative to verify the assembler portion of a verified operating system as well.

To address this gap, we developed an automatic static analysis tool, called Vx86, targeted towards the verification of the Windows Hypervisor [4,5]. Vx86 proves correctness of Intel x86 assembler code with AMD virtualization extensions against procedure contracts and loop invariants. It does so by building on top of other tools. First, Vx86 translates annotated assembler code to annotated C code. The C translation makes the machine model explicit and provides a meaning for the instructions by simulating the instructions on the machine

J. Meseguer and G. Roşu (Eds.): AMAST 2008, LNCS 5140, pp. 284–298, 2008.

state. The resulting C code is then passed to VCC, Microsoft's Verifying C Compiler [16]. VCC translates the annotated C programs into BoogiePL [9], an intermediate language for verification. Boogie [1] then generates logical verification conditions for the translated C code and passes them on to the automatic first-order theorem prover Z3 [8] to either prove the correctness of the translated assembler program or find errors in it.

We found that the simulation approach is a very good fit for our assembler verification effort. There are two reasons for that, one is technical and the other one is social. The technical reason is that C and assembler are in fact very closely related: both use arbitrary pointer arithmetic, both have a very weak type system (albeit a bit stronger in the case of C). So C verifiers, which are good enough for verifying low level OS code, should be good enough to deal with assembler code as well. Mapping assembler code to C thus obviates the need of implementing a full-blown assembler verifier. The social reason is that the users of the assembler verifier are likely to also use the C verifier for other parts of the code, therefore they can get familiar with only one tool instead of two.

This paper presents the design and use of Vx86. Our contributions are

- the development of a translator from annotated assembler code to C (see Subsection 3.1).
- the development of a semantics of x86 assembler with virtualization extensions by providing a simulator in C (see Subsection 3.2).
- the development of correctness criteria for assembler code (see Subsection 4.2).
- the application of the resulting verifier on the Windows Hypervisor code base (approximately 4,000 lines of assembler code) (see Subsection 4.3).

Section 2 introduces the challenges in assembler verification; furthermore it provides some background on VCC. Sections 5 and 6 discuss related work and conclude.

2 Background

2.1 Running Example: *SetZero*

We will explain the inner workings of Vx86 with the *SetZero* assembler code (see Figure 1 on the following page). It is literally taken from the Windows Hypervisor code base; it sets a memory block of 4096 bytes to zero.

This code is written in assembler, because it is optimized for branch prediction, cache lines and pipelines of the processor, something that the Microsoft C compiler cannot achieve.

2.2 Challenges in Assembler Verification

Verifying assembler code is challenging.

- Almost all assembler languages, including Microsoft's x86 assembler, are untyped; however, most of the automatic verification tools use type information to help with the problems of aliasing, framing, etc.

```
1       %LEAF_ENTRY SetZero, _TEXT$00
        db      066h, 066h, 066h, 090h
        db      066h, 066h, 066h, 090h
        db      090h
        ALTERNATE_ENTRY SetZero
6       xor     eax, eax
        mov     edx, X64_PAGE_SIZE / 64
@@:
        mov     [rcx], rax
        mov     8[rcx], rax
11      mov     16[rcx], rax
        add     rcx, 64
        mov     (24 - 64)[rcx], rax
        mov     (32 - 64)[rcx], rax
        dec     edx
16      mov     (40 - 64)[rcx], rax
        mov     (48 - 64)[rcx], rax
        mov     (56 - 64)[rcx], rax
        jnz     short @b
        ret
21      LEAF_END SetZero, _TEXT$00
```

Fig. 1. Original *SetZero* assembler code

- Assembler control flow is unstructured, therefore we need to find a place where to put loop invariants.
- Many assembler instructions have side effects not only on the mentioned registers but on the whole processor state. For faithful verification all of these effects have to be captured, since program logic may later depend on them; conditional jumps, for instance, depend on a flag set in earlier instructions.
- Assembler code often uses bitfields and words interchangeably. For example, the flag register is typically used as a bitfield, but when saved and restored, it is used as a single unit.
- The use of general purpose registers is even more demanding; they are not only used as bitfields and as integers, but also as pointers. A register mask, for example, is used to select the page of a pointer address.

VCC, a verifier that is currently being developed for verifying the C part of the Windows Hypervisor, is designed so that it can support weak type systems, bitvectors, as well as arbitrary goto-systems. This allows us to build Vx86 on top of VCC because it meets all necessary requirements.

2.3 Microsoft's Verifying C Compiler

VCC is a static analysis tool that uses automatic first order theorem proving to show formally that a given sequential C program, compiled for the Intel or AMD x86-32 or x86-64 processors, does what is stated in its specification.

VCC's specification language includes first-order predicates expressed in pre–/postconditions, loop invariants, assertions and assumptions. For modular

reasoning VCC introduces a "region-based" memory management using pure functions and abstract framing to guarantee that functions only write, read, allocate, and free certain locations. VCC also supports ghost state, that consists of specification-only variables that allow one to argue about values that were saved before.

VCC uses three formally related semantics. VCC's base memory model represents values as bit vectors and accesses memory in individual bytes, a simple abstraction represents values as mathematical integers and accesses memory in word sizes, the third model uses the C type system in addition to the second model to rule out many pointer aliases.

In this paper we show in detail how Vx86 uses VCC to verify the partial correctness of the $SetZero$ function, as well discuss the results of verification of a sizable chunk of assembler code from the Windows Hypervisor.

3 Translating Annotated Assembler Code to C

Vx86 processes annotated x86 assembler code, written in the input language for Microsoft's Macro Assembler 8.0 (MASM). Vx86 works as follows:

1. the annotated assembler code is preprocessed by MASM, this inlines all needed assembler macros and definitions;
2. the expanded assembler file is translated to annotated C code;
3. the annotated C code is extended with definitions that (a) make the machine state explicit, and (b) provide a meaning for the assembler instructions in terms of the machine state;
4. the annotated and now self-contained C code is passed to VCC for verification.

3.1 Specification and Syntax Translation of the Assembler Language

To simplify our translation task and to make the verification of the Hypervisor's C code as well as assembler code as uniform as possible, we decided to simply adopt VCC's specification language for Vx86. Specification constructs in Vx86 are introduced using special comments, e.g. pre, post conditions, writes clauses and local ghost variables appear after the assembler header, invariants appear after labels, assumptions, assertions and assignments to ghost variables can appear anywhere in the running code. Figure 2 on the next page describes the fully annotated $SetZero$ Code.

The $SetZero$ assembler procedure requires (1) that $X64\_PAGE\_SIZE$ bytes in the main memory starting at the address pointed to by *rcx* are valid, i.e. allocated (let us call that region the *page*), and that $rcx + X64\_PAGE\_SIZE$ is not allowed to overflow. $SetZero$ guarantees that it only writes the *page*, as well as the registers *rcx*, *edx*, *rax* and *rflags*. $SetZero$ ensures that the *page* is zero. In the body, there is a loop between the label @@ and the jump jnz back to it. After the alignment a ghost variable *count* of type signed 64 bit integer is

```
;^      requires(valid((U1*)rcx,X64_PAGE_SIZE))
;^      requires(rcx+X64_PAGE_SIZE < (U8)-1)
;^      writes(region((U1*)(old(rcx)),X64_PAGE_SIZE), rcx, edx, rax, rflags)
;^      ensures (forall(U4 i; (0 <= i && i < X64_PAGE_SIZE/8) ==>
;^                        *((U8*)(U1*)(old(rcx)) + i) == 0))
        LEAF_ENTRY SetZero, _TEXT$00

        db      066h, 066h, 066h, 090h  ; fill for alignment
        db      066h, 066h, 066h, 090h  ;
        db      090h                    ;

;^          spec(I8 count = 0);
; a ghost variable

        ALTERNATE_ENTRY SetZero
        xor     eax, eax
        mov     edx, X64_PAGE_SIZE / 64
@@:
;^          invariant (valid((U1*)(old(rcx)),X64_PAGE_SIZE))
;^          invariant  (8 * count == (U1*)rcx - (U1*)old(rcx))
;^          invariant (0 < edx && edx <= (X64_PAGE_SIZE/64))
;^          invariant ((U1*)(old(rcx))-64*edx+X64_PAGE_SIZE == (U1*)rcx)
;^          invariant ((U1*)(old(rcx)) <= (U1*)rcx)
;^          invariant ((U1*)rcx < (U1*)(old(rcx))+X64_PAGE_SIZE)
;^          invariant (rax == 0)
;^          invariant (forall(U4 i; (0 <= i && i < count) ==>
;^                                *((U8*)(U1*)(old(rcx)) + i) == 0));
        mov     [rcx], rax
        mov     8[rcx], rax
        mov     16[rcx], rax
        add     rcx, 64
        mov     (24 - 64)[rcx], rax
        mov     (32 - 64)[rcx], rax
        dec     edx
        mov     (40 - 64)[rcx], rax
        mov     (48 - 64)[rcx], rax
        mov     (56 - 64)[rcx], rax
;^          spec({ count += 8; })
        jnz     short @b
        ret

        LEAF_END SetZero, _TEXT$00
```

Fig. 2. With contracts annotated *SetZero* assembler code

introduced. This variable, which is updated as part of the loop body, is needed for describing the loop invariant.

Note that for verification purposes registers and the global memory are considered to be global variables, except that registers are treated specially; they are 64, 80 or 128 bit variables that lie outside the address range of global memory.

```
typedef unsigned long long U8;
typedef unsigned long U4;
typedef unsigned char U1;
typedef struct Flags_t {
        unsigned cf : 1; unsigned res1 : 3;
        unsigned af :1; unsigned res2 :1;
        unsigned zf :1; unsigned sf :1;
        unsigned res3 :3; unsigned of :1;
        unsigned res4 :20; U4 res6;
        } flags_t ;

//registers
register U8 rax, rcx, rdx;
//flags
flags_t rflags ;
#define eax rax //casts are introduced automatically where needed
#define edx rdx //casts are introduced automatically where needed
//eax and edx are the 32 bit versions of the 64 bit rax and rdx

//flag computations
#define zf_comp(a) rflags.zf = (unsigned)(a == 0)
#define sf_comp(a) rflags.sf = (a < 0)
//instructions
#define xor(a,b) a = (a^b); zf_comp(a); sf_comp(a); rflags.af=0; rflags.of=0
#define mov_U4(a,b) unchecked(a = (U8)((U4)b))
#define mov(a,b) unchecked(a = b)
#define add(a,b) a = (U8)((U1*)a + (b)); zf_comp(a); sf_comp(a)
#define dec(a) a = a - 1; zf_comp(a); sf_comp(a)
#define ret() return
#define jnz(a) if (!rflags.zf) goto a
```

Fig. 3. Vx86's machine state and instruction definitions in C

A challenging part of the syntax translation is the introduction of casts. In x86 assembler every register and memory access can be 8, 16, 32, or 64 bit wide. The access modes do not only differ in the number of bits they read or write, but also in the side effects to the higher bits. 8 and 16 bit access modes are just writing the bits given, leaving the rest of the register unchanged. On the other hand, 32 bit access mode extends with zeroes to a 64 bit register. For Vx86 we decided to model general purpose registers as 64 bit unsigned integers; all access modes other than 64 bit are represented in terms of their effect on 64 bit quantities. Due to VCC's ability to switch between different memory models, registers can be viewed as bitvectors or integers.

In assembler, jumps often depend on the value of the status register, i.e. one instruction sets the status register and the following jump is depending on the flags. In our running example, *dec* may set the zero flag and *jnz* performs the jump if the zero flag was set. We translate goto systems from assembler directly into goto systems in C, we only have to resolve symbolic assembler labels, e.g. @*b* and @@ in our example are resolved to a unique label.While most verifiers do not support unstructured goto's, VCC does. It translates goto systems into unstructured control

```
#include "vcc.h"
#include "Assembler.h"

//Page Size
#define X64_PAGE_SIZE 0x01000ULL

void SetZero()
 requires (...)  writes (...)  ensures  (...)
{
spec  (...);
SetZero:
        xor(eax, eax);
        mov_U4(edx, X64_PAGE_SIZE / 64);
_l0:
invariant  (...)
        mov(*(U8*)((U1*)rcx), rax);
        mov(*(U8*)((U1*)rcx + 8), rax);
        mov(*(U8*)((U1*)rcx + 16), rax);
        add(rcx, 64);
        mov(*(U8*)((U1*)rcx + (24 - 64)), rax);
assert  (...);
        mov(*(U8*)((U1*)rcx + (32 - 64)), rax);
        dec(edx);
        mov(*(U8*)((U1*)rcx + (40 - 64)), rax);
        mov(*(U8*)((U1*)rcx + (48 - 64)), rax);
        mov(*(U8*)((U1*)rcx + (56 - 64)), rax);
spec  (...);
        jnz( _l0 );
        ret ();
}
```

Fig. 4. Vx86 generated C code for *SetZero*

flow in Boogie. Next, Boogie translates unstructured goto systems into a system of equations that represent the verification condition. Vx86 does not make the PC explicit. In x86 assembler it is quite difficult to compute addresses instead of using labels to jump, because instructions do not have fixed size.

3.2 Simulator for Assembler

Figure 3 on the preceding page provides an excerpt of the simulator for x86 assembler. The figure only provides the definitions which are needed to verify *SetZero*. The full file has approximately 8,000 bytes of definitions.

The general purpose registers of the x86 processor are defined as 64 bit global variables in our C model. Special registers, like the flag register, are represented using bitfields. Other registers like the floating point registers, which are 80 bit wide and the multimedia registers, which are 128 bit wide, are modelled with the help of structs that have the same form as they would have on the real processor.

The meaning of each assembler instruction is provided by a simulation on these newly introduced memory locations. For providing the instruction semantics we relied on the instruction manuals from AMD and Intel. For example, the instruction "*add rax,rbx*" is not only translated to the C construct "*rax=rax+rbx*" but also into several statements to report the proper flag changes like "*rflags.zf = (rax == 0)*" and "*rflags.sf = (rax < 0)*". Note that such flag changes are performed by most of the assembler instructions.

Unfortunately, the processor instruction manual is not very precise when it comes to the virtualization extensions. For those instructions, the processor state after executing certain operations is only partially defined (see also Subsection 4.1). On the other hand, the Windows Hypervisor does not contain code that operates on floating point and multimedia values; they are only used for saving and restoring the processor state, therefore we do not need to model these instructions in detail.

Figure 4 contains the translation of the annotated *SetZero* assembler code into C (note that we do not include the contracts here, because they are pasted literally from the assembler code into the C code). This code is then passed to VCC; it verifies in less than a second. Alternatively, the code can be passed to a normal C compiler, and then executed or debugged using a regular C debugger.

4 Evaluation

The goal of our verification effort is to verify the assembler portion of the Windows Hypervisor. This section provides some background on Windows Hypervisor, explains the properties that we have verified so far and gives performance data.

4.1 The Windows Hypervisor

The Windows Hypervisor is a thin layer of software written in C and assembler that sits directly on x64 hardware, turning a real multi-processor (MP) x64 machine into a number of MP x64 virtual machines (VMs). These VMs provide additional machine instructions (hypercalls) to create and manage VMs, hardware resources, and inter-VM communication. VMs are viewed as a key enabling technology for a variety of services, such as server consolidation, sandboxing of device drivers, testing, running multiple OSs on a hardware machine, live VM migration, snapshotting/recovery, and high availability. Moreover, it provides such functionality in an OS-neutral way, with a trusted computing base 2-3 orders of magnitude smaller than that of a typical commercial operating system.

The Windows Hypervisor code base is separated into source files written in C and x86 assembler. Assembler code is mainly used to achieve performance optimizations, which cannot be expressed in C, and to access processor instructions that do not have corresponding C instructions. For both reasons, it is obvious that the code should not be changed just to be able to verify it.

The assembler code of the Hypervisor is located in different files, which means there is no inline assembler in the C code. This allows for modular reasoning: the assembler code can be verified separately against its specification.

If C code calls assembler functions, a C prototype for the assembler function has to be provided that expresses the assembler specification not in terms of registers but in terms of C's parameters and memory. We assume that the C compiler translates calls to assembler code using a standard register transfer protocol, where the function's first parameter is passed in register *rcx*, the next in *rdx*, etc. We also assume that every variable of the calling C function is marked as volatile, which means that the compiler is not allowed to store variables temporarily in registers but variables are always read from and written to main memory.

Intel and AMD have developed hardware support for hypervisor systems. For example, they can switch to the hypervisor if hardware interrupts occur and provide multi-stage page tables so that the operating systems do not see that they are working in translated mode. Unfortunately, both companies have their own virtualization instructions. Since the AMD instruction set is older and the implementation in the hypervisor thus (hopefully) has less errors in it, we decided to first support AMD. In future work, there will also be an implementation of the Intel virtualization hardware. Both hardware types can be supported at the same time because they have different instruction names and different processor states. Compared to standard assembler instructions the virtualization instructions are very complex. They are used for context switches between the hypervisor (host system) and the operating systems (guest systems). A typical scenario for a context switch consists of the following sequence of operations: (1) save the host state, (2) load the guest state, (3) run the guest, (4) save the guest state, (5) load the host state. Properties about those virtualization instructions include facts like "the state of the host after the restoring process is the same as it was at the point of the saving". Such a property does not only include the values of registers but also the stack that is administrated by the processor. If the stack has changed (either the place or the content) then the host will have a completely different state. Properties involving virtualization typically range over many registers and memory locations. Additionally, the processor state is usually available twice: once for the host system and once for the guest system. The verification tool then has to scale well to handle such complex functions and specifications, and we have seen verification times for virtualization function degrade (see below).

On the other hand, several functions in the Windows Hypervisor are only used for optimization reasons. The specifications for those functions are not too complicated as we have seen before. However, looking at an optimized implementation is often scary; algorithms are optimized for filling the pipeline most efficiently, to exploit branch prediction and caching. Verifiers however are good at keeping track of detail and so these algorithms are a great target for modern verification technology.

4.2 Well-Formedness Properties

So far we verified only assembler code that is guaranteed to be executed sequentially; this amounts to approximately 4,000 lines. For these 4,000 lines we verified memory safety, arithmetic safety, call safety, and interrupt safety.

Memory safety means that all memory accesses are only performed on valid (i.e., previously allocated and not freed) memory. Therefore, the precondition of an assembler function needs to include validity of all memory locations that are accessed. If a function tries to access memory that cannot be proven valid, VCC reports an error. Similarly, we specify explicitly the set of memory locations being written to, and it is an error to write to a memory location not listed in the writes clause. These properties are enforced to be transitive, i.e., if function f calls g then the writes set of g needs to be contained within the writes set of f, and also the precondition of g needs to follow from the context at the call site (including preconditions of f).

Arithmetic safety means absence of overflows, unless otherwise stated. For operations that can overflow (like addition, multiplication or signed division) VCC automatically adds assertions that check if the result is in the proper range. When an overflow behavior is desired, the user can specify this explicitly.

Call safety means that the stack is cleaned up after every function call and registers are saved before every function call. If f calls g and the postcondition of g does not guarantee that it restores values of registers, then f needs to save itself the registers it cares about. The registers are saved on the stack, therefore it is important to know that g will not modify stack locations above the current stack pointer (stacks are growing down on x86 architecture), and that g does not change the stack pointer. This is expressed using the postcondition $ensures(rsp == old(rsp))$ and by not including region starting with rsp in the writes clause of g (or for that matter in its validity preconditions). On the other hand, all accesses of g to the stack (like push) need to be specified in the way we usually specify memory safety, that is by a precondition like $requires(valid(rsp-40, 40))$.

Interrupt safety means that the stack is cleaned up after processing the whole interrupt. We cannot verify interrupt handlers like regular functions, because some of their subroutines push some registers on the stack, while only other subroutines pop them.

For a few functions we also verified functional correctness, like in the $SetZero$ example shown earlier.

4.3 Experimental Results

We analyzed all assembler files as given, i.e. without a single change except for adding contracts.

Sizes of Verification Task. Table 5 on the next page presents the size in bytes of different files in different processing stages. The column *Annotated ASM* denotes the size in bytes of the annotated assembler function, i.e. the files the user edits. The column *Preproc ASM* shows the file sizes after assembler macro expansion. The column *Translated C* shows the file size after the translation to C. At this point, we have the syntax translation but we do not have the simulator in the code, but only included as header file. Finally, column *Preproc C* presents the result of the C preprocessing phase. This is the file we finally give to *VCC* for

File	*Annotated ASM*	*Preproc ASM*	*Translated C*	*Preproc C*
zero	7,821	16,781	18,040	31,609
crashdump	5,625	17,791	20,217	31,028
GuestContext	1,422	16,673	18,136	29,297
Trap	82,316	420,854	444,865	486,510

Fig. 5. File sizes in bytes at different stage of the translation

Filename	*Verification Time[s]*
zero	1.45
crashdump	3.29
GuestContext	< 0.01
Trap	67.20

Fig. 6. Verification times for various files of the Windows Hypervisor

verification. Note that the translation is processing instruction by instruction. The time consumption is only mentionable for the largest file *Trap* and takes 2 seconds there.

Verification Times. Table 6 gives the verification time in seconds for different functions. We checked 17 files of the Windows Hypervisor for all previously mentioned properties, like memory safety, arithmetic safety, call safety, and interrupt safety. This corresponds to approximately 4,000 lines of assembler code, which is around 90% of the assembler code that is part of Windows Hypervisor. The remaining 10% might be executed concurrently, which we currently cannot handle.

The table shows only the functions of the four files we previously mentioned.

We observe that the time needed for verification increases linearly with size. This is due to the fact that our verification is modular, i.e. procedure by procedure. As long as the procedures do not grow too much in size and do not introduce too many control flow paths, we can verify substantial code. For instance, the procedure *ExceptionDispatch*, which is part of the *Trap* file, has approximately 300 instructions. These 300 instructions turn into several hundred C assignments. Nevertheless, it verifies in approximately 3 seconds. This shows that our approach cannot be used only for short toy functions but also for long and complex assembler implementations.

5 Related Work

The CLI stack project in the late 1980s [2] was the first project focussing on the pervasive verification of computer systems. In total the system consisted of four levels: starting from a verified FM 8502 microprocessor via a simple assembler language up to a verified operating system. Later Boyer and Yu [3] refined this

approach and verified MC68020 assembler programs. They used the theorem prover Nqthm as their verification tool; they formalized the MC68020 as Nqthm theories, thus in effect giving an interpreter for the processor; assembler programs are then translated into expressions over this special logic. Vx86 differs in various dimensions from this early work, Vx86 works on the much more complex x86, Vx86 incorporates contracts (including framing) into the assembler, Vx86 uses an automatic theorem prover (ATP), Vx86 has been used to verify parts of a real industrial strength operating system.

There are various projects to verify micro-kernels. Verisoft [12] is in spirit similar to the CLI project. Verisoft developed machine-models for assembler, small step and big step semantics for more abstract programming languages, and programs for devices, kernels, operating systems and applications. However, the Verisoft project only dealt with idealistic processors, inline assembler, and OS. The L4.verifed project [13] aims at the formal verification of an industrial strength implementation of a L4 micro-kernel, which is highly optimized for the ARM platform. While the L4 project tries to do low level C verification, it has – to the best of our knowledge – not yet started verifying assembler code. Verisoft and the L4 project use the same verification technology. Both systems use the interactive theorem prover Isabelle and a Hoare calculus embedded in Isabelle [20] to verify properties of the micro-kernel. The automation was slightly improved with the integration of automatic tools that can verify parts of the proof obligations [7]. However the resulting system does not yet achieve the automatization we achieved.

Another approach to guarantee that assembler programs are safe are Typed Assembly Languages (TAL) [6]. TALs are low-level, statically typed target languages. TALs guarantee type safety, which typically implies memory safety. However, TALs do not guarantee arithmetic safety, call safety, interrupt safety or other functional properties. Furthermore, TALs are often idealistic assembler languages, they are only used as target languages for compilers; as such they do not deal with the whole instruction set of the processor. We, however, also have to deal with instructions like *HLT* or *CPUID* and the virtualization instruction set.

Proof carrying code (PCC) has a similar goal [18]. Instead of defining type safety for assembler code, PCC adds proofs to untrusted assembler files, which establish certain properties. The receiver of the untrusted code is then able to use a simple and fast proof validator to check that the proof is valid and hence the untrusted code is safe to execute. Like TAL, PCC has focuses on memory safety; it is not a general verification architecture. However, we think that translating contracts from source level into assembler and then using Vx86 to discharge those contracts could be an interesting alternative to extend the reach of PCC.

To our knowledge, there is no other project that tries to verify an existing code base for an optimized hypervisor or which verifies assembler code using contract annotations and an ATP.

6 Conclusion and Future Work

Vx86 is a verifier for proving the correctness of sequential Intel x86 assembler code with AMD virtualization extensions against their contracts. Our approach has been to (1) provide a C simulator for Intel x86, and (2) to translate the annotated assembler code into C code for this simulator. Despite the fact that providing a C simulator seems to be a detour for verifying assembler code, it has turned out that this still allows us to verify the assembler portion of a complex industrial program, like Windows Hypervisor, in reasonable time.

In the process of developing Vx86 we heave learnt the following characteristics of handwritten assembler programs: they might have complex control flow, but they operate only on a few registers and the memory; the operations on registers are often low-level, in addition operations have many side effects. Recursive data structures, which typically need transitivity to describe effects on them, are rarely used in hand-written assembler. As a consequence, changes to registers and the memory can often easily be described by enumeration and quantification.

We have also learned that assembler code is particularly well suited for automated verification:

- Providing an assembler verifier is overdue – except for assemblers there are no tools to help assembly writers.
- Verifying assembler code is beneficial – if assembler code fails systems typically crash.
- Writing assembler contracts is feasible – the contracts often only mention a limited amount of objects, furthermore the contracts are often easier than the highly optimized implementation.
- Discharging assembler contracts is a sweet spot for ATPs – ATPs can deal well with lots of low level detail, since they often have specialized decision procedures for them, they can also deal well with quantifiers; however they often cannot deal well with complex heap structures; luckily user written assembler programs do not use them.

Vx86's simulator semantics is currently based on our understanding of the Intel and AMD instruction manuals. To be more reliable we need a review by hardware developers. If changes are necessary we should be able to incorporate them easily, we just need to change the simulator, the rest of the translation is unaffected.

Vx86 is not yet concurrency aware. Certain parts of the assembler code base are often concurrent. If the Windows Hypervisor, for example, shuts down the physical machine, it does so by stopping all its processors, or in more detail: the last processor which is alive finally has to shut everything down. As soon as VCC will support concurrency, we will investigate how Vx86 can reuse that model.

Vx86 is of course not restricted to consume only Windows Hypervisor code, it can be used to verify other code bases as well. Furthermore, we think that the presented approach is a viable way to quickly provide verifiers for other

processors. In fact, we were recently asked whether we could provide a verifier for an ARM assembler as well. We think that this should be possible in a couple of weeks.

Authors would like to thank Herman Venter for his help with getting Vx86 running and Peter Mueller for his very useful comments about this paper.

References

1. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
2. Bevier, W.R., Hunt Jr., W.A., Stroher Moore, J., Young, W.D.: An approach to systems verification. Journal of Automated Reasoning 5(4), 411–428 (1989)
3. Boyer, R.S., Yu, Y.: Automated correctness proofs of machine code programs for a commercial microprocessor. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 416–430. Springer, Heidelberg (1992)
4. Cohen, E.: Validating the Microsoft Hypervisor. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, p. 81. Springer, Heidelberg (2006)
5. Cohen, E., Hillebrand, M.A., Leinenbach, D., der Rieden, T.I., Moskal, M., Paul, W., Santen, T., Schirmer, N., Schulte, W., Tobies, S., Wolff, B.: The Microsoft Hypervisor verification project (to be published, 2008)
6. Crary, K., Gregory Morrisett, J.: Type structure for low-level programming languages. In: Wiedermann, J., Van Emde Boas, P., Nielsen, M. (eds.) ICALP 1999. LNCS, vol. 1644, pp. 40–54. Springer, Heidelberg (1999)
7. Daum, M., Maus, S., Schirmer, N., Seghir, M.N.: Integration of a software model checker into Isabelle. In: Sutcliff, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 381–395. Springer, Heidelberg (2005)
8. de Moura, L., Bjøner, N.: Z3: An efficient SMT solver. In: TACAS (2008)
9. De Line, R., Leino, K.R.M.: BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report 70, Microsoft Research (May 2005)
10. Dörrenbächer, J.: Vamos microkernel: formal models and verification. In: International Workshop on System Verification (2006)
11. Filliâtre, J.-C., Marché, C.: Multi-prover verification of C programs. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 15–29. Springer, Heidelberg (2004)
12. Gargano, M., Hillebrand, M.A., Leinenbach, D., Paul, W.J.: On the correctness of operating system kernels. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 1–16. Springer, Heidelberg (2005)
13. Heiser, G., Elphinstone, K., Kuz, I., Klein, G., Petters, S.M.: Towards trustworthy computing systems: Taking microkernels to the next level (2007)
14. Leinenbach, D., Paul, W.J., Petrova, E.: Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In: Aichernig, B.K., Beckert, B. (eds.) SEFM, pp. 2–12. IEEE Computer Society, Los Alamitos (2005)
15. Liedtke, J.: On microkernel construction. In: Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15), Copper Mountain Resort, CO (December 1995)
16. Moskal, M., Schulte, W., Venter, H.: Bits, words and types: Memory models for a Verifying C Compiler (2008)

17. Mürk, O., Larsson, D., Hähnle, R.: KeY-C: A tool for verification of C programs. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 385–390. Springer, Heidelberg (2007)
18. Necula, G.C.: Proof-carrying code. In: POPL, pp. 106–119 (1997)
19. Ni, Z., Yu, D., Shao, Z.: Using XCAP to certify realistic systems code: Machine context management. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 189–206. Springer, Heidelberg (2007)
20. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, Technische Universität München (2006)
21. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: Hoffmann, M., Felleisen, M. (eds.) POPL, pp. 97–108. ACM, New York (2007)