Formally Verifying Exceptions for Low-level code with Separation Logic

Marco Paviotti and Jesper Bengtson

IT University of Copenhagen
Rued Langaards Vej 7, 2300, Denmark
{mpav,jebe}@itu.dk

Abstract

Exceptions in low-level architectures are implemented as synchronous interrupts: upon the execution of a faulty instruction the processor jumps to a piece of code that handles the event. Previous work has shown that assembly programs can be written, verified and run using higher-order separation logic [2]. However, execution of faulty instructions is then under specified by either being undefined or terminating with an error. In this work, we initiate the study of synchronous interrupts and prove an example of memory allocator, thus showing that it is possible to write positive specifications of programs that fault. All of our results are mechanised in the interactive proof assistant Coq.

1 Introduction

Assembly code is difficult to prove correct. Standard Hoare-logics make implicit assumptions about the control flow of programs and assume that the code c in a triple $\{P\}c\{Q\}$ has one entry point and one exit point, even though it may internally contain loops and method calls. In assembly programs we cannot make this assumption as the control flows of these languages are inherently unstructured. Control flow is altered primarily by two mechanisms – jump commands and interrupts. Jump commands allow developers to execute code stored nearly anywhere in memory; their use is an active choice, much like writing a loop or calling a method. Interrupts, on the other hand, occur either when something has gone catastrophically wrong (such as dividing by zero or reading from un-mapped memory) or when an action from the environment requires processing (such as the user pressing a key, a change to the file system is made, or the processor clock ticks). While some of the aspects of interrupts might resemble that of function calls, there are substantial differences: synchronous interrupts are not called explicitly but are dependent on a certain events that can occur at run-time, secondly, there cannot be infinitely many calls as after three interruptions the machine reboots. These interrupts are typically referred to as synchronous. Another denotation for synchronous interrupts is exceptions, due to their similarity with the exceptions encountered in languages like Java or ML, and we will use the terms interchangeably.

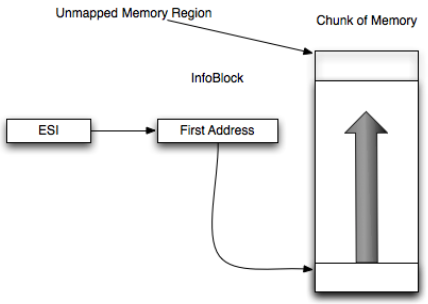
In this paper we extend Kennedy et al.'s semantics for the x86 machine [3] and Jensen et al.'s [2] program logic by adding support for synchronous interrupts. As a case study, we use it to verify a small memory allocator that uses exceptions.

The source code to our mechanisation can be found at `http://www.itu.dk/people/mpav/downloads/exp-tgc05.zip`. The increment to the previous development amounts to 1084 lines of code. The code is compiled with `coqc` version `8.4pl3` with `OCaml 4.00.1`.

2 Memory allocation using exceptions

We use the standard AT&T syntax for assembly notation. For this example, '$\mathsf{mov}\ r,\ v$' stores the value v in the register r, '$[r]$' dereferences a pointer stored in r and '$\mathsf{add}\ r,\ v$' adds the value v to the value stored in the register r.

Jensen et al. [2] implemented and verified a simple bound-and-check memory allocator. We verify an alternative version, whose behaviour and code is depicted in Figure 1. In our allocator there are no checks for overflow or memory bounds, instead, we mark the end of the available memory with an *unmapped location*. The code takes an argument *info* that is a single pointer to the start of memory and begins by moving the starting address of the information block to the ESI register and then by moving its value to the EDI register. After this preamble, EDI will eventually point at the beginning of the available memory. Now we write the value 0 in the memory pointed by EDI. By writing a value to the byte of memory we wish to allocate we will trigger an exception if that memory is unmapped, i.e. when the end of the memory available to the allocator has been reached. It is then up to the interrupt handler to catch the exception, but by jumping to the *fail* address it will mimic the behaviour of the handler in previous work [2]. If the memory is mapped, the control flow will go through and add four bytes to the EDI register to keep track advance the pointer to the free memory. At this point we update the information block by storing the value of the EDI register into the value pointed by ESI.



$$\begin{aligned}\mathsf{alloc}(\mathit{info}) \triangleq\ & \mathsf{mov}\ \mathsf{ESI}, \mathit{info};\\ & \mathsf{mov}\ \mathsf{EDI}, [\mathsf{ESI}];\\ & \mathsf{mov}\ [\mathsf{EDI}],\ 0;\\ & \mathsf{add}\ \mathsf{EDI}, 4;\\ & \mathsf{mov}\ [\mathsf{ESI}],\ \mathsf{EDI}.\end{aligned}$$

Figure 1: Allocator code snippet

3 Allocator with exceptions specification

In order to give the specification of the piece of code in Figure 1 we use Jensen's step-indexed variant of separation logic, but here we prefer to keep the presentation as simple as possible, thus using standard separation logic connectives as $\star$ for the usual *separating conjunction* in separation logic, $\mapsto$ for a points-to predicate for the registers and $\hookrightarrow$ as a points-to predicate for the memory. Moreover, we use the question mark $r?$ for registers and memory addresses as syntactic sugar for $\exists, v. r \mapsto v$ and similarly for $\hookrightarrow$. Here, we borrow the continuation passing style specification from previous work [2], thus, a specification has the following continuation-passing style form:

$$\vdash (\mathsf{safe} \otimes Q \implies \mathsf{safe} \otimes P) \oslash i..j \mapsto c \tag{1}$$

which states that a program c stored in the memory from the address i and the address j is safe to run from P *provided* that there is a continuation that runs safely from Q.

The specification for the example in Figure 1 follows the same pattern, but, since the program can succeed or fault we need two continuations, one stating what happens upon success and one stating what happens upon failure, a pre-condition and a invariant (omitted for space reasons) specifying that there exists a storage which ends are bounded by an unmapped memory region and that there exists and IDT containing the pointers to the handlers.

We define the specification allocSpec as the pre and post-conditions of the code in Figure 1 stating that, when the code is pointed by the range of addresses i to j, is safe to execute from a state

$$P \triangleq \mathsf{EIP} \mapsto i \star \mathsf{INTL} \mapsto 0 \star \mathsf{EDI}? \star \mathsf{ESP} \mapsto sp \star (sp{-}4..sp) \mapsto spval \tag{2}$$

where EIP points to the beginning of the code, INTL is the register keeping track of the level of interruptions, EDI is a temporary register and ESP is the stack pointer, *provided* that the program is safe in case an exception occurs, i.e. that there exist an handler which is going to take on the computation from the address $fail$ with the INTL set to 1 and the stack pointer containing the return address to the original code

$$Q_1 \triangleq \mathsf{EIP} \mapsto fail \star \mathsf{INTL} \mapsto 1 \star \mathsf{EDI}? \star \mathsf{ESP} \mapsto (sp{-}4) \star (sp{-}4..sp)? \tag{3}$$

and that there is a program which is safe run from the address j with the EDI register pointing to the end of the allocated memory and with the interrupt level set at zero in case the allocator succeeds

$$\begin{aligned} Q_2 \triangleq & \mathsf{EIP} \mapsto j \star \mathsf{INTL} \mapsto 0 \star \mathsf{ESP} \mapsto sp \star (sp{-}4)..sp \mapsto spval \star \\ & \exists p, \mathsf{EDI} \mapsto (p{+}4) \star (p..(p+4))? \end{aligned} \tag{4}$$

By wrapping up the tree formulas all together we obtain the allocSpec specification:

$$\mathsf{allocSpec} \triangleq \vdash ((\mathsf{safe} \otimes Q_1 \wedge \mathsf{safe} \otimes Q_2) \implies \mathsf{safe} \otimes P) \oslash i..j \mapsto c \otimes \mathsf{Inv}$$

Finally, we prove that implementation of the allocator respects the specification:

Theorem 1. *The specification* allocSpec *for the piece of code in Figure 1 is sound.* `[Coq proof]`

4 Conclusions and Future Work

We have extended an existing mechanisation of x86-assembly created by Jensen et al. to support synchronous interrupts. Jensen's model is expressive enough to reason about mutable code and we stay true to this design philosophy by storing the IDT and all handlers in memory, allowing them to be dynamically updated by the processor. Our extensions to the program logic are also very conservative. By allowing the memory points-to predicate to state that certain memory is unmapped (and not only what it contains), we obtain a logic that is expressive enough to verify programs that use synchronous interrupts. We believe that this is a testiment not only to the validity of our design decisions, but also of the quality of the original mechanisation.

References

[1] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 2 (2A, 2B & 2C): Instruction Set Reference*, January 2013.

[2] J. B. Jensen, N. Benton, and A. J. Kennedy. High-level separation logic for low-level code. In *Proceedings of POPL*, pages 301–314. ACM, 2013.

[3] Andrew Kennedy, Nick Benton, Jonas Braband Jensen, and Pierre-Évariste Dagand. Coq: the world's best macro assembler? In *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, pages 13–24, 2013.

[4] The Coq Development Team. *The Coq Reference Manual, version 8.4*, 2012.