Bitfields and Tagged Unions in C – Verification through Automatic Generation

David Cock

Sydney Research Lab., NICTA*, Australia

Abstract. We present a tool for automatic generation of packed bitfields and tagged unions for systems-level C, along with automatic, machine checked refinement proofs in Isabelle/HOL. Our approach provides greater predictability than compiler-specific bitfield implementations, and also provides a basis for formal reasoning about these typically non-type-safe operations. The tool is used in the implementation of the seL4 microkernel, and hence also in the lowest-level refinement step of the L4.verified project which aims to prove the functional correctness of seL4. In the seL4 implementation, it has eliminated the need for unions entirely.

1 Introduction

In this paper we present a tool that automatically generates inline-able C functions to implement tagged unions of packed bitfield types, based on a simple domain-specific language specification. We then generalise, and suggest a technique to exploit the desires of systems programmers to ease program verification.

The motivation for this work was the C implementation of the seL4 microkernel, and the needs of the associated L4.verified project [8, 3]. The seL4 microkernel [5, 7] is an evolution of the L4 family [17] for secure, embedded devices. The L4.verified project aims to prove its functional correctness. The need to produce code that can be verified with reasonable effort requires the disciplined use of 'ugly' programming idioms, those which violate the basic abstractions of the underlying semantic model. In our case, these are heap type aliases, i.e. unions, non-type-safe pointer accesses, and sub-machine-word manipulations. These violations occur commonly together, in the tagged union and bitfield construct. See Fig. 1 for an example from the OKL4 kernel [19], a current commercial implementation of L4. This pattern is very regular, and an obvious target for automation. Generation of this code is desirable for two reasons: First, via controlled tagged unions, it adds functionality to C in a disciplined, type-safe way. Second, bitfield implementations vary widely, both in performance and in actual behaviour between compilers, and even different versions of the same compiler. As a result, they are usually mistrusted by kernel programmers. In contrast, our generated code is fast, predictable and formally correct.

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

```
union {
    struct {
        BITFIELD2(word_t,
                type    : 2,
                tcb_p   : BITS_WORD - 2
        );
    } x;
    word_t raw;
};
```

Fig. 1. Example of combined union/bitfield usage. From OKL4 2.1, "include/caps.h", lines 92-100.

Our approach is to provide an opaque, abstract type, implementing the tagged-union/bitfield semantics, together with generated accessor functions. The tool automatically provides the proofs that the functions behave as expected. Whilst this is not a radically new idea, our approach is successful precisely because we target the regular low-level functions, which nonetheless comprise roughly 10% of the code within seL4. This tool also provides a case study for the use of the C semantics of Tuch et al. [26], and Dawson's Isabelle/HOL library for machine-word [4]. The remainder of the paper is laid out as follows: Section 2 introduces the specification language used to describe the bit-level layout of structures, Section 3 shows the C code generation framework, and Section 4 explains the framework of automatically generated proofs to allow reasoning without descending to the level of pointers and bit manipulation.

2 Specification Language

The tagged-union/packed-bitfield structure is useful in a number of contexts e.g. Hardware-dictated page-table layouts, hardware register mapping, and highly optimised data structure storage. As a running example, we will consider a subset of the seL4/ARM capability representation. Capabilities are used as a proxy for authority, and we consider only two capability types: Null caps (`null_cap`) which function as placeholders, and Untyped caps (`untyped_cap`) which convey authority over a power of two sized block of memory. The capability is represented as a two word (64 bit) bitfield: Null caps contain no data other than the type tag, whereas Untyped caps have two fields: `capBlockSize` and `capPtr`, a pointer aligned on a 16-byte boundary.

The cap representations are specified as follows (the full grammar is included in Fig. 5). First the machine word size (32 bits for ARM) is specified with the `base` keyword:

```
base 32
```

Next, bitfield blocks are specified. Fields are listed from most-significant to least-significant bit (Fig. 2). The `padding` keyword introduces anonymous

padding space, to achieve the desired alignment, and the `field` keyword reserves space for a named field. In the `null_cap` example, `padding 32` reserves one empty machine word, `field capType 4` allocates a 4 bit field at the top of the second word, and `padding 28` explicitly fills the remainder of the second word. The trailing padding is mandatory, where the fields do not fill the lower bits of the last word.

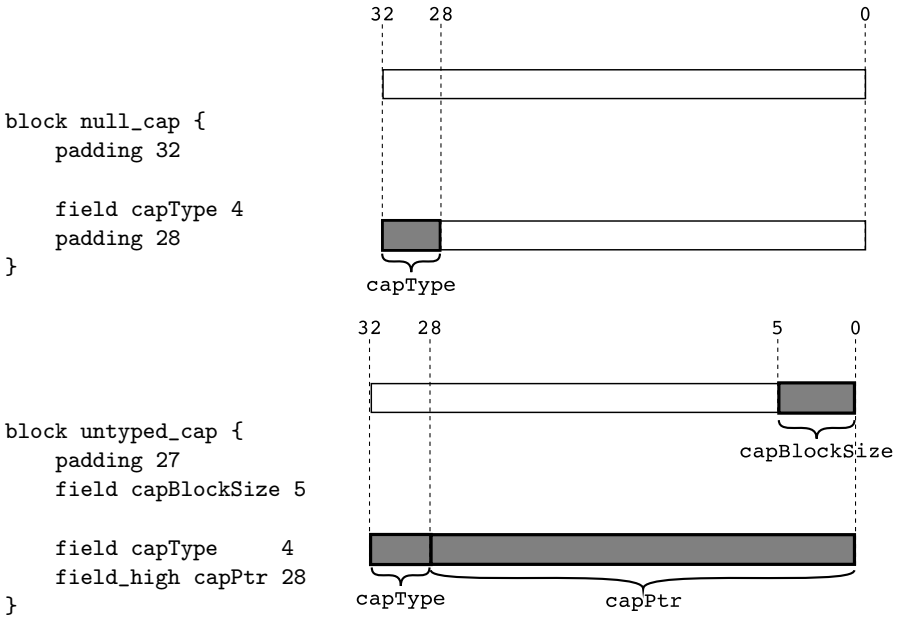

Fig. 2. Packed bitfield layout

Padding between fields is inserted explicitly, fields are forbidden to cross word boundaries, and the size of each block must be a multiple of the base word size. These restrictions ensure that the implementation maps efficiently onto common machine operations, and present no difficulties in practice. The `field_high` keyword specifies that a field should be left-aligned to the word size when read or written, padded on the right with zero bits, see Fig. 3.

Finally, blocks are grouped together into tagged unions (Fig. 4).

The `tagged_union` keyword is followed by the name of the union, and the name of the tag field, then a list of block names, together with their associated tag values. All blocks in the union must be the same size, and each must contain a tag field. All tag fields must be the same size, and at the same location within the block.

Fig. 3. `field_high` implementation

```
tagged_union cap capType {
    tag null_cap 0
    tag untyped_cap 1
}
```

Fig. 4. Tagged union specification

3 Generated Code

This section gives a brief overview of the code generated from the specifications above. Each block and union in the specification language is translated to a C representation with appropriate access and update functions. Each object is represented by a struct containing simply an array of machine words, and for each union, an enum of tag values. The wrapping struct allows pass and return by value in C.

```
struct cap {
    uint32_t words[2];
};

typedef struct cap cap_t;
```

```
enum cap_tag {
    cap_null_cap = 0,
    cap_untyped_cap = 1,
};
typedef enum cap_tag cap_tag_t;
```

For each block and union, the tool generates create, access and update functions. Each such function is generated in a purely functional version, which passes and returns a stack object of appropriate struct type:

```
static inline cap_t CONST
cap_untyped_cap_set_capBlockSize(cap_t cap, uint32_t v) {
    assert(((cap.words[0] >> 28) & 0xf) ==
           cap_untyped_cap);

    cap.words[1] &= ~0x1f;
    cap.words[1] |= (v << 0) & 0x1f;
    return cap;
}
```

Also generated is a pointer lifted version, which operates indirectly on heap values through a supplied pointer:

```
entity_list ::= empty
              | entity_list block
              | entity_list tagged_union
              | entity_list base

base ::= "base" INTLIT

block ::= "block" IDENTIFIER "{" fields "}"

fields ::= empty
         | fields "padding" INTLIT
         | fields "field_high" IDENTIFIER INTLIT
         | fields "field" IDENTIFIER INTLIT

tagged_union ::= "tagged_union" IDENTIFIER IDENTIFIER "{" tags "}"

tags ::= empty
       | tags "tag" IDENTIFIER INTLIT
```

Fig. 5. Specification language grammar

```
static inline void
cap_untyped_cap_ptr_set_capBlockSize(cap_t *cap_ptr,
                                     uint32_t v) {
    assert(((cap_ptr->words[0] >> 28) & 0xf) ==
           cap_untyped_cap);

    cap_ptr->words[1] &= ~0x1f;
    cap_ptr->words[1] |= (v << 0) & 0x1f;
}
```

The prototypes for the remaining functions are given in Fig. 6.

Note that the generated API only provides functions that read the tag field through the union type, and no function to write it directly. This imposes a class-like behaviour on the types. The subtype is set implicitly at creation time, and can only be modified by overwriting with an object of a different type. This will turn out to be an important property for verification.

In practice the output is automatically pruned, so that only those functions actually used in the source are generated. This speeds the proof process. As the specification language is highly focussed and carefully limited, the generated code is simple and fast, highly predictable, and easily inlined by the compiler.

4 Generated Specifications

The final and most novel part of the approach consists of the automatically generated, machine checked function specifications together with their automated

```
static inline uint32_t CONST
cap_get_capType(cap_t cap);

static inline uint32_t PURE
cap_ptr_get_capType(cap_t *cap_ptr);

static inline cap_t CONST
cap_untyped_cap_new(uint32_t capBlockSize,
                    uint32_t capPtr);

static inline void PURE
cap_untyped_cap_ptr_new(cap_t *cap_ptr,
                        uint32_t capBlockSize,
                        uint32_t capPtr);

static inline uint32_t CONST
cap_untyped_cap_get_capBlockSize(cap_t cap);

static inline uint32_t PURE
cap_untyped_cap_ptr_get_capBlockSize(cap_t *cap_ptr);

static inline cap_t CONST
cap_untyped_cap_set_capBlockSize(cap_t cap,
                                 uint32_t v);

static inline void
cap_untyped_cap_ptr_set_capBlockSize(cap_t *cap_ptr,
                                     uint32_t v);
```

Fig. 6. Generated function prototypes for the example

proofs. The function of the generated proofs is to not only to show implementation correctness, but also to provide sufficient reasoning power to allow any statement involving the generated functions to be rephrased in terms of simple operations on abstract, high-level types in the theorem prover. This means that we can avoid invoking bit manipulations and pointer dereferences when reasoning about the packed structures as part of a larger proof. We can instead reason about higher-level types, for which there is well established support.

This abstract representation is expressed in terms of Isabelle's record types, which behave much like struct or record constructs in typical programming languages, providing access and update of disjoint fields. The bitfields from the C level are represented as records of fields on the abstract level. For example, the `untyped_cap` block is represented as follows:

```
record cap_untyped_cap_CL =
    capBlockSize_CL :: "word32"
    capPtr_CL :: "word32"
```

Tagged unions are represented by an algebraic datatype wrapping the records corresponding to the component bitfields, with one constructor for each. Tag fields are not included in the record representation, but are implied by the choice of constructor within the union type. Any bitfields which are empty after the removal of the tag field are represented simply by a naked constructor, with no associated record. The `cap` union translates thus:

```
datatype cap_CL =
    Cap_null_cap
  | Cap_untyped_cap cap_untyped_cap_CL
```

The name convention is that the C types and identifiers, when parsed into Isabelle, are tagged by appending `_C`, whereas the lifted types are tagged with `_CL`. The connection between the C level and the abstract level will be provided by two functions in the example below: *cap_lift* and *cap_untyped_cap_lift*. The former lifts any *cap_C* to a *cap_CL*, and the latter lifts a *cap_C* with the `untyped_cap` tag directly to a *cap_untyped_cap_CL*. It is under-specified in all other cases.

The properties of the generated functions are expressed as strongest-postcondition Hoare rules. Specifically, we use Schirmer's [23] verification environment for imperative programs in Isabelle/HOL. It contains a verification condition generator (VCG) which automates reasoning about Hoare-triples. Tuch's et al. [26] instantiation to C parses the generated code directly into Isabelle/HOL and into Schirmer's representation language SIMPL.

As an example, we will take the generated specification of the also generated C function `cap_untyped_cap_set_capBlockSize`. It takes two arguments, an untyped capability *cap* and a new block size *v*. It returns the original capability with the new block size. The formal specification below translates this into a record update:

```
"Γ ⊢ ⦃s. cap_get_tag ´cap = cap_untyped_cap⦄
     ´ret__struct_cap_C :==
       PROC cap_untyped_cap_set_capBlockSize(´cap, ´v)
     ⦃cap_untyped_cap_lift ´ret__struct_cap_C =
      cap_untyped_cap_lift ˢcap (|capBlockSize_CL := ˢv AND (mask 5)|) ∧
      cap_get_tag ´ret__struct_cap_C = cap_untyped_cap⦄"
```

The specification above reads as follows. For all program contexts Γ, if the tag of the C-struct *cap* in the current state (indicated by ´) equals the value *cap_untyped_cap*, and we execute the function *cap_untyped_cap_set_capBlockSize* with parameters *cap* and *v*, storing its return value in *ret__struct_cap_C*, we will arrive at the following post condition: lifting the return value to the abstract record type is the same as lifting the value of *cap* in the initial state *s*, and then performing an update of the abstract record field *capBlockSize_CL* with the value *v* had in state *s*. Additionally, as a convenience for automated methods in the larger proof, we provide that the tag of the return value remains *cap_untyped_cap*. A separate specification states (and the tool proves) that the function is side-effect free, i.e. that no global variables, including the heap, are changed. The term *AND (mask 5)* carries the additional information that the field has a size of 5 bits. This form proved more convenient so far than the alternative of having

an abstract field of word length 5, because casting between word lengths often introduces additional proof obligations.

The meaning of the rule can also be expressed by means of the commuting diagram in Fig. 7.

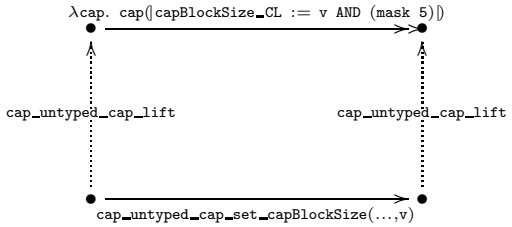

Fig. 7. Refinement picture for field update.

For Fig. 7, consider `cap_untyped_cap_set_capBlockSize` as a function from `cap_t` (its first argument) to `cap_t` (its return value). Control flows left to right, and *r* (| a := x |) is the Isabelle syntax for the record *r*, with field a updated with value *x*. This makes it clear that the function of the rule is to allow us to transform a function call into a record update, by commuting it with a lift. We can therefore take any precondition of the form *P (cap_untyped_cap_lift cap)*, and commute it past any number of field updates, to produce a postcondition of the form *P (f (cap_untyped_cap_lift cap)*, where *f* is the composition of a number of record updates.

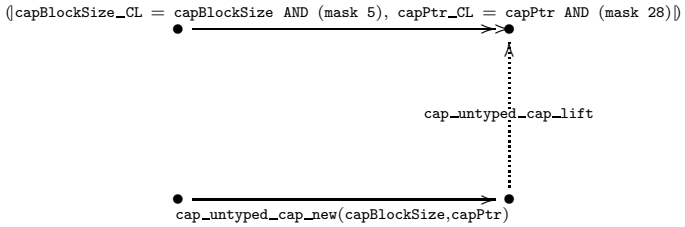

Fig. 8. Refinement picture for initialisation

That this is be useful becomes clear when we consider the equivalent diagram for the *cap_untyped_cap_new* function (Fig 8), which returns a new untyped capability. This provides a starting point for our chain of reasoning, by providing the identity *cap_untyped_cap_lift cap = (|capBlockSize_CL=capBlockSize AND (mask 5),capPtr_CL=capPtr AND (mask 28)|)*. We can base our argument on

such a case, as long as bitfield objects are only initialised via the appropriate `*_new` functions, and the type tags are never externally modified. This justifies the API restriction introduced in Section 3, which is adhered to by the seL4 kernel implementation without loss of convenience or performance.

Equivalent rules are proved automatically for all the generated functions, and their pointer-lifted versions. The latter involve direct heap access to record fields and automate the interactive reasoning Tuch provides [25]. Additionally to what Tuch provides, we make use of the concept of packed records that allows us to ignore padding in record implementations, and derive more precise properties of the corresponding memory layout. Packed records are represented by a type class in Isabelle that simply states that all fields in the record have sufficient size to make padding unnecessary.

The proofs for the specifications above are fully automated and generally consists of two to three automated method invocations in Isabelle. The first of these is a call to the C-level VCG mentioned above. The second and possibly third, first reduce the remaining proof obligation from variable, heap, and struct-updates to a goal on bit-vectors only. This is then solved automatically with a carefully designed set of a generic, algebraic rewrite rules for the bit operations involved in the generated functions. The direct proof script for one specification of a typical C-function is about 10 lines of Isabelle script only.

5 Related Work

Earlier work on OS verification includes PSOS [10] and UCLA Secure Unix [27]. Later, KIT [2] describes verification of process isolation properties down to object code level, but for an idealised kernel far simpler than modern microkernels. The Verisoft project [11, 12] is attempting to verify a whole system stack, including hardware, compiler, applications, and a simplified microkernel VAMOS. The VFiasco [14] project is attempting to verify the Fiasco kernel, another variant of L4 directly on the C++ level. These verification projects do not use generated C code for automating parts of their proof obligations. In the Verisoft case, there is no reason to distrust the compiler as it is verified as well [20, 15]. Directly using C or dropping down to assembly code to implement the desired features does not have the benefit of the high-level reasoning support and API the tool presented here provides, though.

The verified proofs in this work build directly on Tuch's et al. memory model for C [26, 25, 24] which in turn builds on work by Schirmer [23, 22] that provides a generic framework, verification condition generator, and Hoare-logic [13] for imperative programs. Both are intended for interactive verification. This paper uses the predictable structure of the generated code to completely automate the pointer level proofs on the C implementation.

This work also builds directly on Dawson's machine-word library [4] for Isabelle/HOL. Despite recent progress in tools like Yices [6], bit-vector proofs for machine words remain hard to automate. Traditional SAT solvers are usually too slow to handle the resulting proof obligations on realistic word sizes. Again

due to the predictable nature of the generated code, the tool is able to fully automate the bit-level verification conditions with a set of algebraic rewrite rules. This means that switching to different, say 64-bit based, architectures should not result in any noticeable slowdown of the generated proofs.

General translation validation [21] and compiler correctness like Leroy's et al. work [16] are related to the topic. As mentioned above, the tool presented here can exploit the known, predictable nature of the application domain to provide a convenient interface to and integration into client proofs.

Related to generated correctness proofs is also the idea of proof-carrying code [18], which usually focusses on the machine level and on specific properties like memory safety or resource constraints. Functional correctness is not usually targeted, because it is impossible to automate completely. Barthe et al. [1] come close by automatically transforming certificates from source code to machine code and, similarly to the work presented here, generating proofs for generated code. In contrast to this, the bitfield generator here does not require any source-level proof as input. It generates a full functional correctness statement automatically.

Denney et al. [9] automatically prove properties about generated aerospace software. The generated code appears more complex than the one presented here, but the semantics they are using is not foundational and the properties do not cover full functional correctness, but specific safety properties only.

6 Conclusion

This paper has summarised a generator for tagged unions of packed bitfields in the C programming language as they are used in low-level systems code and operating system kernel implementations. In theory, this data structure can be implemented with C primitives without resorting to a generator. However, compiler implementations of bitfields seem to be so unpredictable in memory layout and performance over different platforms and compilers that kernel programmers distrust this compiler feature more than others.

The tool generates efficient, predictable, and above all correct C code from a short, high-level description that is detailed enough to provide precise memory layout specification which is important to map data, for instance, to memory mapped hardware device registers. The generated code includes the data type itself as well as an API for convenient, high-level access on the stack and on the heap.

This work shows that an automatic correctness proof of generated code for controlled environments is not hard to achieve, even if this code contains bit-level reasoning and pointer access. The proof is foundational in the sense that it assumes no specific axioms on the application domain, but is built directly on the semantics of the C programming language. The proof is machine-checked in the theorem prover Isabelle/HOL and provides an example of translation-validation: Instead of proving the correctness of the generator, the correctness of the generated code is proven instead.

The usefulness of the tool reaches further than a stand-alone correctness proof. The Hoare-triples proven integrate directly, in the same formal model, with larger implementation proofs of client-code using the generated bitfields. The Hoare-triples are designed such that client-proofs and code have no need to reason about the internal representation or bit-level operations that are carried out. They provide an abstract interface. Translation validation has likely made this easier achieve than in a generator correctness proof. No meta-level reasoning or switching of formal models is required.

The tool is expected to automate an estimated 10% of the C implementation proofs in the L4.verified project, covering 5,000 lines of C code with 7,000 lines of generated proof. The seL4 kernel, including all generated inline functions is 14,000 lines of code. The tool is generally applicable to code that needs to have direct, reliable control over the memory layout of data structures. The technique of generating proofs that integrate well into interactive environments should generalise easily to reasonably constrained application domains where the structure of the generated proof is highly predictable.

References

1. G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. In *Proceedings of the 13th International Static Analysis Symposium (SAS)*, volume 4134 of *LNCS*, pages 301–317, Seoul, Korea, August 2006. Springer.
2. William R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.
3. David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In Cesar Munoz and Otmane Ait, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs'08)*, LNCS. Springer, 2008. To appear.
4. Jeremy E. Dawson. Isabelle theories for machine words. In *Seventh International Workshop on Automated Verification of Critical Systems (AVOCS'07)*, Electronic Notes in Computer Science, Oxford, UK, September 2007. Elsevier. To appear.
5. Philip Derrin, Kevin Elphinstone, Gerwin Klein, David Cock, and Manuel M. T. Chakravarty. Running the manual: An approach to high-assurance microkernel development. In *Proc. ACM SIGPLAN Haskell Workshop*, Portland, OR, USA, September 2006.
6. Bruno Dutertre and Leonardo de Moura. The yices smt solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, 2006. Link visited June 2008.
7. Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone. A memory allocation model for an embedded microkernel. In *Proc. 1st MIKES*, pages 28–34, Sydney, Australia, 2007.
8. Kevin Elphinstone, Gerwin Klein, Philip Derrin, Timothy Roscoe, and Gernot Heiser. Towards a practical, verified kernel. In *Proc. 11th Workshop on Hot Topics in Operating Systems*, San Diego, CA, USA, May 2007.
9. Bernd Fischer Ewen Denney and Johann Schumann. Using automated theorem provers to certify auto-generated aerospace software. In *Proc. 2nd International Joint Conference on Automated Reasoning (IJCAR'04)*, volume 3097 of *LNCS*, pages 198–212. Springer, 2004.

10. Richard J. Feiertag and Peter G. Neumann. The foundations of a provably secure operating system (PSOS). In *AFIPS Conf. Proc., 1979National Comp. Conf.*, pages 329–334, New York, NY, USA, June 1979.
11. Mauro Gargano, Mark Hillebrand, Dirk Leinenbach, and Wolfgang Paul. On the correctness of operating system kernels. In Joe Hurd and Thomas F. Melham, editors, *Proc. TPHOls'05*, volume 3603 of *LNCS*, pages 1–16, Oxford, UK, 2005. Springer.
12. Mark A. Hillebrand and Wolfgang J. Paul. On the architecture of system verification environments. In *Hardware and Software: Verification and Testing*, volume 4899 of *LNCS*, pages 153–168, Berlin, Germany, 2008. Springer.
13. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
14. Michael Hohmuth and Hendrik Tews. The VFiasco approach for a verified operating system. In *Proc. 2nd ECOOP-PLOS Workshop*, Glasgow, UK, October 2005.
15. Dirk Leinenbach and Elena Petrova. Pervasive compiler verification—from verified programs to verified systems. In Ralf Huuck, Gerwin Klein, and Bastian Schlich, editors, *Proceedings of the 3rd international Workshop on Systems Software Verification (SSV'08)*, Electronic Notes in Computer Science, Sydney, Australia, February 2008. Elsevier. To appear.
16. Xavier Leroy. Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In J. G. Morrisett and S. L. P. Jones, editors, *33rd symposium Principles of Programming Languages (POPL'06)*, pages 42–54, New York, NY, USA, 2006. ACM.
17. J. Liedtke. On μ-kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, December 1995.
18. George C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM.
19. Open Kernel Labs, Inc. OKL web site. http://www.ok-labs.com, 2007. Visited May 2008.
20. Elena Petrova. *Verification of the C0 Compiler Implementation on the Source Code Level*. PhD thesis, Saarland University, Computer Science Department, Saarbrücken, Germany, 2007.
21. Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In Bernhard Steffen, editor, *Proc. 4th Intl. Conf. Tools and Algorithms for Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *LNCS*, pages 151–166. Springer, 1998.
22. Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, AI, and Reasoning*, volume 3452 of *LNAI*, pages 398–414. Springer, 2005.
23. Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.
24. Harvey Tuch. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, School for Computer Science and Engineering, University of New South Wales, Sydney, Australia, 2008.
25. Harvey Tuch. Structured types and separation logic. In Ralf Huuck, Gerwin Klein, and Bastian Schlich, editors, *Proceedings of the 3rd International Workshop on Systems Software Verification (SSV'08)*, Electronic Notes in Computer Science, Sydney, Australia, February 2008. Elsevier. To appear.

26. Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 97–108, Nice, France, 2007. ACM.
27. Bruce Walker, Richard Kemmerer, and Gerald Popek. Specification and verification of the UCLA Unix security kernel. *Commun. ACM*, 23(2):118–131, 1980.