Modular Verification for Computer Security

(Invited Paper)

Andrew W. Appel
Princeton University
Princeton, NJ 08540

29th IEEE Computer Security Foundations Symposium,
June 2016

***Abstract*—For many software components, it is useful and important to verify their security. This can be done by an analysis of the software itself, or by isolating the software behind a protection mechanism such as an operating system kernel (virtual-memory protection) or cryptographic authentication (don't accepted untrusted inputs). But the protection mechanisms themselves must then be verified not just for safety but for *functional correctness*. Several recent projects have demonstrated that formal, deductive functional-correctness verification is now possible for kernels, crypto, and compilers. Here I explain some of the *modularity principles* that make these verifications possible.**

I. Introduction

Computer systems are built from numerous software and hardware components. These components may be buggy or malicious, so it is standard practice to have a trusted kernel that enforces protection mechanisms, to guarantee that bugs (or maliciousness) of one component cannot propagate to others.

But a trusted kernel itself may be very large. It may include

1) the operating-system kernel that implements (e.g.) virtual-memory protection and process preemption;
2) the network communication stack, such as TCP/IP;
3) crypto primitives, such as AES or RSA;
4) crypto protocols built atop these primitives;
5) compilers that translate all of the above to machine language;
6) RTL (register-transfer language) implementations of that machine language;
7) netlist (logic gates) implementations of the RTL.

The systems we use today are built in a modular way from components, software/hardware processes, and layers such as I describe. An engineer working or an expert analyzing at one of these layers (or in one of these components) will not want to know much, will not be able to know much, about the internals of the next component. For example, the network-stack experts don't want to be optimizing-compiler experts, who don't want to be cryptographers. That fractioning of knowledge is essential, otherwise we could not put together the big and capable systems that we enjoy today.

But how can we come to trust such a big system built from diverse components? Must there inevitably be bugs and vulnerabilities, so that the hacker or the Nation-State Attacker can always get in?

I will argue that:

1) We can gain a substantial basis for trust—we can substantially improve the security of our systems—by using *end-to-end, modular, deductive program verification* of all of these components.
2) The verifications must be done in different domains: *program logic or program refinement* to prove that concrete programs correctly implement abstract algorithms; *application-domain reasoning* to prove that abstract algorithms accomplish our goals.
3) A *higher-order pure functional programming language* is a particularly effective way of expressing functional specifications for modular program verification.
4) One can connect these different styles of verification together with no *specification gaps,* by embedding all of the different verification methods into a single general-purpose logic.

II. Static analysis vs. functional correctness

Many static analysis tools—commercially available or in academic/industrial research—check safety or liveness properties of programs. Some are *sound* (if they assert a safety property then every execution of the program must indeed be safe), some are unsound. Such tools can be very useful in the software engineering process to reduce the bug count and increase reliability. Especially the sound tools have application in computer security: for example, the Java typechecker (bytecode verifier) or a software fault isolator (SFI) can guarantee that untrusted code cannot escape its sandbox even in the absence of hardware memory protection.[1] Sound static analysis tools are one kind of formal-methods verification.

Should we static-analyze the OS kernel, crypto library, bytecode verifier? Perhaps, but it's not enough. These protection mechanisms exist to separate client programs from each other, and to protect system code from client programs. To do this, the protection mechanisms must be not only *safe* but also *correct.* It is not enough that the OS kernel or crypto library has no buffer overruns: it must also compute the right answer.

Therefore, an important application of formal machine-checked functional-correctness verification is for systems code that implements protection mechanisms.

III. Functional specifications in functional languages?

A *functional specification* of a program characterizes the program's observable behavior. In this section I will discuss

[1] One can even do machine-checked verification of the soundness of a type-based [1] or SFI-based [17] sandboxer.

two different approaches to functional specs: mathematical relations, and functional programs. Consider this C program,

```
int minimum(int a[ ], int n) {
  int i, min;
  min=a[0];
  for (i=0; i<n; i++)
     if (a[i]<min) min=a[i];
  return min;
}
```

with these three different functional specifications. The notation array $a\ \sigma$ means, "program variable a is an array whose current contents are the sequence σ."

Specification A :

$\forall\ \sigma : \text{list}(\mathbb{Z})$.
precondition : $\{|\sigma| = n > 0\ \wedge\ \mathsf{array}\ a\ \sigma\}$
postcondition : $\{\forall j.\ 0 \le j < n\ \rightarrow\ ret \le \sigma_j\}$

Specification B :

$\forall\ \sigma : \text{list}(\mathbb{Z})$.
precondition : $\{|\sigma| = n > 0\ \wedge\ \mathsf{array}\ a\ \sigma\}$
postcondition : $\{\exists i.\ 0 \le i < n \wedge ret = \sigma_i$
$\wedge \forall j.\ 0 \le j < n\ \rightarrow\ ret \le \sigma_j\}$

Specification C :

```
LET
function fold (f: α → β → β) (b: β) (al: list(α)) : β =
  match al with nil ⇒ b | a::ar ⇒ f a (fold f b ar) end

function min (i: ℤ) (j: ℤ) : ℤ =
  if i<j then i else j

function hd (d: α) (al: list(α)) : α =
  match al with nil ⇒ d | a::ar ⇒ a end
IN
```

$\forall\ \sigma : \text{list}(\mathbb{Z})$.
precondition : $\{|\sigma| = n > 0\ \wedge\ \mathsf{array}\ a\ \sigma\}$
postcondition : $\{ret = \mathsf{fold\ min\ (hd\ 0\ }\sigma)\ \sigma\}$

Specs A and B characterize some mathematical relations between the input and the output. You might notice that Spec B is stronger. Spec A just ensures the return value is $\le$ all elements of the array, without necessarily being one of the elements. Does that mean Spec B is better than Spec A? Perhaps not; perhaps property A is all that the client needs.

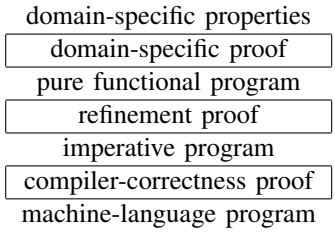
Spec C says, "the program computes exactly this function." Now, any program that satisfies Spec C will also satisfy Specs A and B; that's straightforward to prove, using the simple semantics of the ML-like functional language in which *fold, min,* and *hd* are written.

That is, Spec C is at least as strong as A or B. In fact, any specification written in this style must have a strongest-possible postcondition. That's because the functional language we use is *deterministic* and *total*—if it typechecks then it can't go wrong and must compute a single unique result.

Is that a good thing? One might think, not necessarily:

- Perhaps property A is all the client needs; are B and C overkill?
- Specification C looks verbose.
- What if my program doesn't have a deterministic semantics?

Even so, experience in many projects has shown that *writing functional specs in functional languages* is very effective indeed. These projects use a three-level approach:



The reason this strategy often works so well is that it modularizes the proof effort. The refinement proof can focus on *programming,* the relation between some clumsy low-level language (such as C, Java, or assembly language) and how it implements a given function. In this proof, domain knowledge about the mathematics of the application domain is often not necessary; all the domain knowledge can go in the high-level proof about properties of the functional program.

With this method, it's crucial to have a functional language that's easy to reason about formally, i.e., a language with a clean proof theory. Successful examples are,

Gallina, the functional language embedded in the CiC logic of the Coq proof assistant.

HOL-fun, (the name I will use to refer to) the pure functional language embedded in the higher-order logic of the Isabelle/HOL proof assistant.[2]

ACL, Applicative Common Lisp, the pure functional Lisp used in ACL2 (A Computational Logic for Applicative Common Lisp).

Later (§VI) I will argue that first-order logics such as ACL2 are insufficiently modular.

IV. Examples

Researchers around the world have accomplished full functional correctness verifications of protection mechanisms, employing principles of end-to-end, modular, deductive program verification with respect to functional specifications. The results I list here all use a *functional program* as a refinement layer in the proof.

CertiKOS: Hypervisor kernel, implemented in C and proved correct in Coq, by multilayer contextual refinement proofs between Gallina functional specifications and C operational semantics [14]. Then the Gallina specs are shown to enjoy correctness properties: process (VM) isolation [10], VM liveness, OS availability, and correct

[2] "The language doesn't have a name. We speak of Isabelle/HOL's function definition facility/command." Tobias Nipkow, e-mail of April 11, 2016.

execution of guest processes (they execute as if running solo on the bare machine).

HMAC: OpenSSL SHA-256 and HMAC, implemented in C and proved correct in Coq [8]. Proved by Separation Logic proofs between C and Gallina functional specifications. The Gallina specs are direct transcriptions of the FIPS 180 and FIPS 198 standards for SHA and HMAC respectively. Then the Gallina specs are shown to enjoy the correctness property that a computationally limited adversary not in possession of the session key cannot distinguish the HMAC from a PRF (pseudorandom function). That proof is by reasoning in a computational monad of probability distributions.

L4.verified: The seL4 operating-system kernel, implemented in C and proved correct in Isabelle/HOL [16]. Proved by operational refinement proofs between the C program and an "abstract functional specification," that is, a program in HOL-fun. Then the functional program is shown to enjoy correctness properties such as integrity and confidentiality.

Raft: The Raft consensus protocol, implemented as a state machine translated by the Verdi system (in Coq) to Gallina for extraction to OCaml [28]. Correctness properties proved in Coq include: state machine safety, election safety, log matching, leader completeness, and linearizability. The relation between the Gallina spec and the low-level imperative program (assembly language) is by compilation in OCaml; this part is not proved correct.

In this short paper I cannot possibly survey all the excellent results in related areas; I pick just these few to illustrate some principles of their construction.

V. Fun with functional program specs

The advantage of a two-level verification—low-level implementation to functional program, then functional program to domain-specific mathematical/relational properties—is that *functional programs are easy to reason about*. I will use our HMAC verification as an example.

SHA-2 (Secure Hash Algorithm) is an iterated block compression: in 256-bit mode, a 512-bit *block* is mixed with a 256-bit *hash* yielding a new 256-bit hash. Blocks can be chained together, and odd-length strings are handled by a padding+length suffix.

In 1996, Bellare *et al.* [7] described the HMAC algorithm for symmetric-keyed cryptographic authentication based on any (good-enough) iterated block compression (such as SHA). They proved that it provides cryptographic security: in particular, a polynomial-time adversary who lacks the session key cannot distinguish HMAC from a random function.[3] (In short, HMAC is a PRF, a pseudorandom function.)

[3]This proof was subject to the hypothesis that the underlying block-compression function (e.g., the core of SHA-256) is a PRF; a more precise characterization of the assumption is in footnote 12 of Appel [5]. No one knows how to prove this assumption about SHA, but it is generally accepted as a "true enough" property of SHA.

In 2002 the FIPS 180-2 standard for SHA-2 was promulgated, and in 2008 the FIPS 198 standard for HMAC.

Bellare's proof is not about a *program*, but about a mathematical presentation (in English and LaTeX math) of an algorithm. FIPS 180-2 and FIPS 198 comprise a similar but not identical English+LaTeX presentation of an algorithm that should be an instance of what Bellare *et al.* describe.

In 2015 we proved, with machine-checked proofs in Coq, that OpenSSL's SHA+HMAC, as actually compiled to assembly language, is cryptographically secure (is a keyed PRF) [8]. The four authors of our paper worked on modularly separable tasks, with little communication except at specification interfaces. The specification interfaces are written in Coq, typically in the form of functional programs in Gallina.

- Appel proved the OpenSSL SHA-256 C program implements (our Coq formalization of) FIPS 180.
- Beringer proved the OpenSSL HMAC C program implements (our Coq formalization of) FIPS 198.
- Petcher implemented in Coq the Bellare *et al.* proof that (our Coq formalization of) Bellare *et al.*'s presentation of HMAC/SHA is a keyed PRF.
- Ye proved that (our Coq formalization of) FIPS 180 + FIPS 198 is equivalent to (our Coq formalization of) Bellare *et al.*'s presentation of HMAC/SHA.

Our SHA/HMAC proof would have been impractical without a modular separation of concerns. In particular, Petcher's proof has no interaction with the C language or with techniques for proving correctness of C programs. Appel's and Beringer's proofs have no interaction with Petcher's computational monad over probability distributions, or with techniques for proving correctness of probabilistic programs, or with crypto proof techniques about distinguishability by computationally limited adversaries. Ye's proof concerned neither the C language nor the probabilistic techniques.

This proof of a cryptographic primitive is just one example. The OS kernel proofs cited above also enjoy this modularity—the separation of concerns between *implementation in C* and *properties of the algorithm*—by using a functional-language specification interface between these two levels.

VI. Program modules and data abstraction

SHA-256 and HMAC really are functions: given an input (message + key) there is a unique output. So it's easy to see how a C program can implement a functional spec written in a pure functional language. But an operating system, or *one module* of an operating system or of any large software system, is not simply a function from an input to an output. A module has internal state, and interface operations (methods) that operate on the external state. No problem! We can represent this in the functional language as a dependent record of a *representation type*, a *value* of that type that represents the "current internal state," and a series of pure *functions* that take the representation type and other inputs, and produce the representation type and other outputs [14].

The essence of data abstraction is that the private representation of an abstract data type (ADT) can be modeled by

existential quantification [22]. Not only the *type* of the private variables, but the *representation invariant* (a predicate), must be quantified over. Let's do an example:

Counter.h

```
struct counter;
struct counter * make(void);
void inc(struct counter *p);
int get(struct counter *p);
```

Counter.c

```
#include "Counter.h";
struct counter {int x3; int x2;};   /* deliberately baroque */
struct counter *make(void) {
  struct counter *p = (struct counter *)malloc(sizeof(*p));
  p->x3 = 0; p->x2=0;
  return p;
}
void inc (struct counter *p) {
  p->x3+=3; p->x2+=2;
}
int get (struct counter *p) {
  return p->x3 - p->x2;
}
```

Part of the module-interface's specification is (and don't be intimidated by the notation, just read on),

$$\begin{array}{l} \Sigma(\tau : \mathrm{Type}).\Sigma(\phi : \ \tau \to \mathrm{Prop}). \\ \quad (1 \to \Sigma(x:\tau).\phi(x), \qquad\qquad\qquad\qquad make \\ \quad\ \Pi(x:\tau).\phi(x) \to \Sigma(y:\tau).\phi(y), \qquad\quad inc \\ \quad\ \Pi(x:\tau).\phi(x) \to \mathbb{Z} \times \Sigma(y:\tau).\phi(y)) \qquad get \end{array}$$

This says there is a hidden representation whose *type* is τ (in the example implementation, $\tau = \mathbb{Z} \times \mathbb{Z}$, a pair of integers representing the x3 and x2 fields of the struct). Then, ϕ is a *representation invariant* (in the example, $\phi(x_3, x_2) = \exists i.\ x_3 = 3i \wedge x_2 = 2i$). Then, within the scope of the existential quantification of τ and ϕ, there are characterizations of the three interface functions. For example, *inc* takes input x of type τ such that $\phi(x)$, and produces output y such that $\phi(y)$.

Both the "deliberately baroque" implementation shown above, and the obvious straightforward implementation, are instances of this representation-hiding specification.

There is more to this specification: I haven't shown that x is actually represented as a C struct, and I haven't shown how we represent the claim that *inc* actually increments rather than decrements (or adds 6). But the point I want to make is: quantification over types (τ) and predicates (ϕ) is essential to *data abstraction*, that is, reasoning about program modules with private data representations.

Therefore, when doing large compositional proofs about modular programs (such as CertiKOS), it's important to have a higher-order logic, which permits quantification over predicates. Thus I focus on HOL and Coq/Gallina, and it's not surprising that the OS kernel verifications L4.verified [16] and CertiKOS [14] were done in these (respective) logics rather than a first-order logic such as ACL2.

VII. Program Synthesis

Instead of (laboriously, interactively) proving the correctness of a C program w.r.t. a functional specification, one might wish to *compile* the functional spec directly to an imperative (C, or assembly language) program. That is, replace the box labeled refinement proof with one labeled synthesize or compile. In that case, one would need a formally verified functional-language compiler. Several approaches have been demonstrated.

One could compile the ML-like language used in the logic (such as Gallina or HOL-fun) using standard techniques for ML compilation (with heap-allocated data structures and function closures), but with a compiler verified with a machine-checked proof. **CakeML** is such a compiler, proved correct w.r.t. formalized operational semantics of (a variant of) Standard ML source language and (x86-64) machine language [18]. It is written in (HOL4's version of) HOL-fun, proved correct in HOL4, and translated through CakeML itself (with careful attention to bootstrapping the proof so this is not too circular).

CertiCoq is an ongoing project at Princeton and Cornell to write, in Gallina, a proved-correct compiler from Gallina to CompCert C light. In the Raft example cited in §IV, CertiCoq+CompCert could be used to obtain formally verified compile layers.

A disadvantage of that approach is that heap-allocated data and closures require garbage collection. Some kinds of low-level security-relevant software, such as OS kernels and crypto primitives, cannot easily tolerate garbage collection.

But most software *can* tolerate garbage collection—just consider the huge amount of code written in Java, Perl, Python, Ruby, Go, and so on. If this software is insecure, it is not because of garbage collection! In fact, just the opposite, since explicit malloc/free is the source of many bugs and security vulnerabilities. Probably most software—even most operating-system and crypto-protocol software—should be written in garbage-collected languages. For example, Mirage OS kernels (written in garbage-collected OCaml) are layered over over the Xen hypervisor (written in C) [21].

But still, can the Gallina or HOL-fun functional specification be automatically translated to an "imperative style" non-garbage-collected low-level program?

Cogent is a linear-typed polymorphic first-order pure functional language. It is not as general as ML (or Gallina or HOL-fun), but it is at least sufficient for implementing (verified functionally correct) file systems [3]. From a Cogent program, the compiler generates a HOL-fun program, a C program, and a refinement proof that the C program implements the HOL-fun program [23]. The linear types ensure that malloc/free can be used instead of garbage collection.

VIII. Compiler correctness

To show correctness of the SHA/HMAC assembly language programs that result from compilation, we rely on two more self-contained modular proofs in Coq:

- Leroy's proof [19] that the CompCert C compiler is correct with respect to the operational semantics of C and of x86 assembly language (both formalized as small-step operational semantics).
- Appel *et al.*'s proof [6] that our *Verifiable C* program logic (used in the SHA and HMAC proofs above) is sound with respect to Leroy's operational semantics for C.

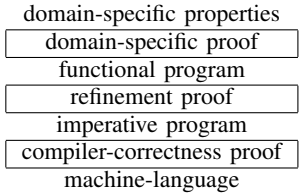
There is not much overlap between these two proofs. However, there has been much work (2006–2016) on the *specification interface* between them, that is the Coq presentation of a memory model and operational semantics for CompCert C light [20], [9].

Similarly, CertiKOS enjoys an end-to-end connection between its contextual refinement proofs (about C programs) and the CompCert compiler-correctness proof.

In connecting to a verified compiler, it is advisable to bypass the compiler's lexer and parser to connect directly via abstract syntax tree (AST) representations of parsers. The high-level tools (such as Verifiable C or CertiKOS's Certified Abstraction Layers) can reason directly on ASTs, using the formal operational semantics as a reasoning principle. The compiler semantics phases can reason on the same ASTs, using the same operational semantics, in the same logic. There is no need to reason formally about parsing and pretty-printing.

IX. Tool linkage

The pretty picture repeated at right can get rather messy, depending on how one specifies and proves compiler correctness. What works well is to have concrete abstract-syntax tree (AST) data structures



as the specification interface for compiler correctness. This is how the HMAC (Verifiable C) and CertiKOS proofs connect to the CompCert proofs.

L4.verified, the correctness proof in Isabelle/HOL of the seL4 kernel, connects to its compiler using a somewhat weaker semantic specification. It was built originally using an untrusted "C parser" that generates a (partly) shallowly embedded functional semantics, and compiled using `gcc`, an unverified compiler. Certainly at the time the L4.verified project started (in 2005 or so [13]) there was no available verified C compiler, nor any trustworthy formal specification of an operational semantics for C.

By 2012, CompCert did exist; so why not connect it to the L4.verified proofs? The seL4 team tried this but unfortunately,

> "We have used [CompCert] on the verified C source of the seL4 microkernel, but in this case found the result unsatisfactory: There is a remaining chance of a mismatch between CompCert's interpretation of the C standard in the theorem prover Coq and the interpretation of the C standard the seL4 verification uses in the theorem prover Isabelle/HOL, esp. in cases where the standard is purposely violated to implement machine-dependent, low-level operating system (OS) functionality. Reconciling these two semantics is non-trivial. Firstly the logics of Coq and Isabelle/HOL are not directly compatible. Secondly, since the seL4 semantics is largely shallowly embedded, an equivalence proof would have to be performed for each program, ..." [26]

Instead they take a different approach: they use ad-hoc translation validation of a `gcc` compilation. Proof checking now relies on two proof assistants (Isabelle/HOL and HOL4) and two SMT solvers (Z3 and SONOLAR). The validation processes uses both tactical proof search and SMT solving. The primary specification interface is the *control-flow graph*. From the C program there is refinement-based proof search *down* to the graph; from the ARM binary there is heuristic-based decompilation (with proof) *up* to the graph; then SMT-based proof is used to match the two graphs.

Unlike a standard deeply embedded SOS[7], the partly shallow embedding obstructs some kinds of reasoning about the syntactic C program in the proof assistant.[4] In particular, one cannot reason *in the logic* about transformation functions from one program to another–that is, compilers or domain-specific language processors.

So therefore: L4.verified does indeed use machine-checked formal verification to connect the functional spec to the C program and thence to the machine-language program. That's good. But their system has several disadvantages. The trusted base is huge; validation is slow and must be done on every recompile of the C program; it's hard to reason generally about program transformers.

This case study illustrates several issues regarding the specification interface to the compiler:

- Abstract syntax trees and structural operational semantics are *good.* The programming-languages research community has 35 years experience formalizing them for use as interfaces between compilers and programmers [25]. Failing to use this abstraction layer is a missed opportunity for modularity. Not only for direct refinement proofs on C programs one at a time: ASTs + SOS permit cleaner specifications and proof of domain-specific languages that target C. Perhaps the Cogent compiler [23] would be simpler if it could have targeted an AST rather than heuristic-based decompilation.

[4] "[The] C-parser uses Norbert Schirmer's Simpl language, which is deeply embedded with respect to statements, but just uses HOL functions (with lambdas) for expressions. The parser also produces types in Isabelle which match input types from C, e.g. an Isabelle record per `struct` in the C source. All of this is designed to produce an AST which the user can manipulate conveniently like it was the C program.

"However, for automatic or reflected methods, this is a headache. Because the expressions are just state transformers, you can write a VCG or other proof method, but you can't write a translator to another language (in HOL). You can write a translator at the ML level, but then the translator has to be validating rather than validated, e.g. AutoCorres. [Another reason the translator has to be validating rather than validated is:] Because some of the types are unknown until the parser runs, you can't obviously use the AST as a target language for some conversion. ... you can't show that you produced the same result as some external C program." [Thomas Sewell, e-mail of April 23, 2016.]

- Compilation is complete, decompilation is incomplete. Programming languages such as C are designed so that there is an algorithm for translating a legal source program to a target program. (It almost goes without saying.) Sewell *et al.*'s decompilation heuristics must be tuned to each application; for example, they do not handle nested loops, because these were unnecessary for seL4.
- *Reflection.* Sewell *et al.*'s refinement from C to control-flow graphs, and decompilation from machine-language to graphs, is driven by tactical proof search, which is slow. CompCert's translation from C to assembly language is simply the execution of an ML-like functional program; within an order of magnitude, it is approximately as fast as running `gcc`. That is, CompCert takes advantage of *proof by reflection,* the ability to write a functional program inside the logic, prove it correct, then use it in further proofs. CompCert is proved, once and for all in Coq, to preserve observable behavior between source program and target program. Each time a C program is compiled, it is *not* necessary to do a tactical translation-validation proof.
- When possible, one should connect proof modules together at specification interfaces *in the same logic and proof system.* Then we don't have to worry whether the program proof and the compiler proof interpret the operators of the language semantics in the same way: they both import the same specification in logic, and that's the end of it.

Still, formal top-to-bottom functional correctness verification is much better than top-to-middle verification. Both CertiKOS and L4.verified guarantee that the next bug to show up in `gcc` cannot undetectedly compromise the security of their operating systems. Neither can the next misunderstanding of the C semantics (see the next section), since the top-to-middle proof connects *at the specification interface* with the middle-to-bottom proof. If that specification in the middle is not exactly "C," no matter; the proofs still compose.

X. The rise of the C language lawyers

In the first decades of the 21st century, the discussions leading up to (and past) the C11 standard made the semantic definition of C more precise. This definition is still in natural language and "litmus tests," rather than in formal mechanized logic. Even so, the characterization of *undefined behavior* is clearer than before, as is the understanding that compilers are responsible only for the defined behavior of the C language.

This increased precision had an unexpected and paradoxical effect. The *language lawyers*[5] who write the optimizing compilers are "taking advantage" of legalisms in order to make programs misbehave! Wang *et al.* explain this with two examples [27].

[5]From Wikipedia (Rules_lawyer): A rules lawyer is a participant in a rules-based environment who attempts to use the letter of the law without reference to the spirit, usually in order to gain an advantage within that environment. [*Shared Fantasy,* by Gary Alan Fine, University of Chicago Press, 2002, pp. 109–113.] The term "language lawyer" is used to describe those who are excessively familiar with the details of programming language syntax and semantics. [*The Jargon File* ed. by Eric S. Raymond, v.4.4.8, October 2004.]

Example 1

```
char *buf = ...;
char *buf_end = ...;
unsigned int len = ...;
if (buf + len >= buf_end) return; /* len too large */
if (buf + len < buf) return; /* overflow */
/* write to buf[0..len-1] */
```

In recent years `gcc` *deletes the second if-statement.* The language lawyers who maintain the optimizing C compiler point out that if buf+len overflows, then (buf + len < buf) is not a defined computation (it "goes wrong" in the Milner sense); and the compiler is allowed to compile undefined computations however it wants. The operating-system hackers complain that in the "good old days" when the C compiler did not delete the second if-statement, the undefined computation did not actually cause buggy behavior in practice; the second if-statement executed its *then* clause as intended.

Example 2

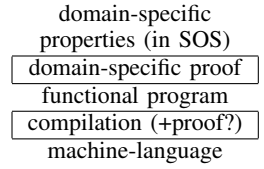
```
struct tun_struct *tun = ...;
struct sock *sk = tun->sk;
if (!tun) return POLLERR;
/* write to address based on tun */
```

If tun==NULL then the dereference tun->sk will have been undefined ("wrong"); therefore the compiler (recently) *deletes the if-statement.* What's the consequence? In the Linux kernel (from which this example was taken), if tun==NULL then tun->sk does not segfault (as it would in user mode), it just dereferences page zero (which is mapped in the kernel). This fetches a nonsense value; but fortunately the if-statement prevents the nonsense value from being used. When the *if* is deleted, then there's a bug.

I'm a language lawyer myself, but ethical concerns make me ask: How can the poor operating-system hacker cope with the language lawyers? The best defense is *never compile a program that exhibits undefined behavior.* A program proved functionally correct will never exhibit undefined behavior.[6]

XI. When a function won't do

Compilers *seem* to be an exception to the general rule that specifications should be functional programs. Verified compilers such as CompCert, CakeML, and CertiCoq have specifications in the form of structural operational semantics (SOS). These are not functions, they are inductive relations. That is, the source and target languages are *deeply embedded* into the object logic, not shallowly embedded as semantic functions.[7]



However, all these verified compilers are pure functional programs. Instead of refinement proof between functional program and executable program, they are written directly in the logic's functional programming language, and extracted automatically for compilation instead.

[6]Wang *et al.* [27] propose a different solution: a static analysis to detect C code that behaves differently when the compiler does or does not optimize based on undefined behavior.

These examples support my thesis rather than disprove it: it's useful to have a functional program as one of the specification layers. In the diagram, (+proof?) indicates that Coq's extractor/compiler is not proved correct; in HOL with CakeML or (eventually) Coq with CertiCoq this phase will be proved.

These examples do show the importance of reasoning in a logic that supports higher-order reasoning about inductive relations. Proving compiler-correctness in a first-order (SMT-compatible) logic can be done for toy examples [11], but I am not aware of attempts to do it at scale.

Randomness.[8] Consider a program that is nondeterministic, or that explicitly uses randomness. For example, a quicksort implementation may use a random-number generation to choose a pivot element. We would like to reason about both the correctness and the average-case complexity of the program.

We *could* model this as a deterministic program that consumes an input from an external stream of random numbers. Then we *could* model the program as a simple pure function. But it is not a very natural model. In particular, in what order does the function consume coin-flips from the external stream? Really it should not matter; but the determinized program is considered a *different function* if we reorder the two recursive calls to quicksort. And this determinized program, which threads the random stream explicitly through the recursive calls, is difficult to parallelize.

Perhaps this is an application domain where a pure functional program does *not* serve well as a functional specification. Instead, perhaps some sort of probabilistic Hoare logic [12] is appropriate.

Graph algorithms. Imperative programs with pointer data structures with mutation can be refinements of pure functional programs. Our Verifiable C program logic (in which we did HMAC) uses *separation logic* for such proofs. But when the data structures go beyond lists and trees, to graphs with dynamic patterns of sharing, the going gets rough even in separation logic: "Programs manipulating mutable data structures with intrinsic sharing present a challenge for modular verification." [15] Graph algorithms are not trivial to model with functional programs, especially concurrent algorithms.

So perhaps my thesis of "functional programs are a good specification layer" applies only to *easy* domains like compilers, operating systems, and cryptography.

XII. Limitations

In the Introduction I talked about the software/hardware stack all the way down past the ISA (instruction-set architecture) to the RTL (register-transfer language) and netlist. Indeed, from the RTL to the netlist, formal machine-checked functional-equivalence verification (FEV) tools are heavily used in industry. But there is a big gap between the ISA and the RTL: ISA specifications have not been heavily formalized, especially in industry and especially in details such as their cache coherence models [4]. Closing this verification gap is an interesting question, in which there is active academic research; but it is beyond the scope of this paper. However, I will remark that the operational-semantic description of the ISA, along with an axiomatic description of the cache coherence model [2], should be an excellent kind of modular specification interface between the software and the hardware.

Functional correctness is a practical necessity for those system components that implement protection mechanisms: OS kernels, file systems, crypto libraries. Program testing, even of the most organized and principled variety, cannot catch all the bugs that a clever and determined attacker will be looking for. The corollaries of functional correctness are important too, including *immunity to buffer-overrun vulnerabilities* and general *absence of undefined behaviors*.

Still, functional correctness does not automatically imply *freedom from timing channels* or *resilience in the presence of hardware faults.* Formal machine-checked program analyses of such properties would be useful in conjunction with the methods I have described in this paper.

And what about the bootloader, did it actually install our verified components? And does the chip actually implement that netlist, or were there fab-level hardware trojans? A proof of functional correctness does not make an entire security assurance case! But known cases of deliberate trojans installed by insiders are rare: most rootkits are installed by outsiders exploiting unintentional functional-incorrectness vulnerabilities. Therefore, functional-correctness verification would yield significant improvements in security.

XIII. Conclusion

Modular machine-checked functional-correctness verification of significant system security components is practical now. I recommend three important modularity principles: (1) Use a functional specification written as a functional program, to separate domain-specific reasoning from low-level program verification. (2) Use operational semantics of the low-level programming language (such as C or assembly) to separate program verification from compiler verification (or machine-architecture verification). (3) To verify modular

[7] An *inductive data type* is a type associated with a finite set of *constructors*. The constructors build instances, possibly recursively, and also permit traversal ("parsing") of the data. Examples are lists (with constructors *nil* and *cons*), trees with (*internal node* and *leaf*). Direct support for inductive data types is found in some languages (such as ML and Haskell) and logics (such as Coq and some implementations of HOL, higher-order logic). Inductive data types are a bit like context-free grammars, and they work very well to express abstract-syntax trees (AST) that represent programs inside compilers.

Functions in Gallina or HOL-fun have direct semantic meaning, *as functions,* in their respective logics (CiC and HOL); such a program is called *shallowly embedded*. In contrast, the AST representation of a program is just syntax—it doesn't mean anything by itself except a static tree. Such a program is called *deeply embedded*. Structural operational semantics (SOS) is a way of giving meaning to deep embeddings by means of inference rules, showing the transformations on ASTs that can "calculate" the results of program execution. These inference rules are also a kind of inductive construction, but they construct logical propositions rather than data; they are an example of an *inductive relation.*

All of this is described in standard textbooks such as Pierce [24]. The point here is that in modular layered proofs about the correctness of security-kernel implementations, it is *very* useful to work in a logic capable of expressive reasoning about inductive relations.

[8] Thanks to my student Qinxiang Cao for this observation.

programs, use a higher-order logic that permits abstraction by quantification over predicates.

Finally, to minimize semantic gaps at specification interfaces, it is helpful if all the verifications can be done (and tools can be embedded) in a common framework—such as a general-purpose higher-order logic proof assistant.

Acknowledgements. This research was supported in part by DARPA agreement number FA8750-12-2-0293 and by NSF Grant CCF-1407794. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. Semantic foundations for typed assembly languages. *ACM. Trans. on Programming Languages and Systems*, 2009.

[2] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, July 2014.

[3] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, Atlanta, GA, USA, April 2016.

[4] Andrew Appel, Chris Daverse, Kenneth Hines, Rafic Makki, Keith Marzullo, Celia Merzbacher, Ron Perez, Fred Schneider, Mani Soma, and Yervant Zorian. Research needs for secure, trustworthy, and reliable semiconductors. Final workshop report of the NSF/CCC/SRC workshop on *Convergence of Software Assurance Methodologies and Trustworthy Semiconductor Design and Manufacture*, https://www.src.org/calendar/e004965/sa-ts-workshop-report-final.pdf, 2013.

[5] Andrew W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Trans. on Programming Languages and Systems*, 37(2):7:1–7:31, April 2015.

[6] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge, 2014.

[7] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *Advances in CryptologyCRYPTO96*, pages 1–15. Springer, 1996.

[8] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of OpenSSL HMAC. In *24th USENIX Security Symposium*, pages 207–221. USENIX Assocation, August 2015.

[9] Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. Verified compilation for shared-memory C. In *European Symposium of Programming*, Lecture Notes in Computer Science. Springer, 2014. To appear.

[10] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward compositional verification of interruptible OS kernels and device drivers. In *Proc. 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)*, June 2016.

[11] Martin Clochard, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Formalizing semantics with an automatic program verifier. In *6th Working Conference on Verified Software: Theories, Tools and Experiments (VSTTE), Springer LNCS 8471*, pages 37–51, July 2014.

[12] JI Den Hartog and Erik P de Vink. Verifying probabilistic programs using a Hoare like logic. *International Journal of Foundations of Computer Science*, 13(03):315–340, 2002.

[13] Kevin Elphinstone, Gerwin Klein, and Rafal Kolanski. Formalising a high-performance microkernel. In *Workshop on Verified Software: Theories, Tools, and Experiments (VSTTE 06)*, pages 1–7, August 2006.

[14] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *42nd ACM Symposium on Principles of Programming Languages (POPL'15)*, pages 595–608. ACM Press, January 2015.

[15] Aquinas Hobor and Jules Villard. The ramifications of sharing in data structures. In *POPL'13: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 523–536. ACM, January 2013.

[16] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, feb 2014.

[17] Joshua A. Kroll, Gordon Stewart, and Andrew W. Appel. Portable software fault isolation. In *27th Computer Security Foundations Symposium (CSF'14)*, pages 18–32. IEEE, July 2014.

[18] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 179–191. ACM Press, 2014.

[19] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[20] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert memory model. In Appel [6], chapter 32.

[21] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, New York, NY, USA, 2013. ACM.

[22] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, July 1988.

[23] Liam O'Connor, Christine Rizkallah, Zilin Chen, Sidney Amani, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Alex Hixon, Gabriele Keller, Toby Murray, and Gerwin Klein. COGENT: certified compilation for a functional systems language. *arXiv preprint arXiv:1601.05520*, 2016.

[24] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Mass., 2002.

[25] Gordon D. Plotkin. A structural approach to operational semantics. Technical report, University of Aarhus, Denmark, 1981.

[26] Thomas Sewell, Magnus Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 471–481, Seattle, Washington, USA, jun 2013. ACM.

[27] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. A differential approach to undefined behavior detection. *Communications of the ACM*, 59(3):99–106, 2016.

[28] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the Raft consensus protocol. In *CPP 2016: The 5th ACM SIGPLAN Conference on Certified Programs and Proofs*. ACM Press, January 2016.