UNIVERSITY OF CALIFORNIA, SAN DIEGO

Reducing the Costs of Proof Assistant Based Formal Verification
or: Conviction without the Burden of Proof

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Zachary Lee Tatlock

Committee in charge:

Professor Sorin Lerner, Chair
Professor Samuel Buss
Professor Ranjit Jhala
Professor Todd Millstein
Professor Yuanyuan Zhou

2014

The Dissertation of Zachary Lee Tatlock is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2014

DEDICATION

For Ma and Pa.

EPIGRAPH

Program verification by proof is an absolute scientific ideal, like purity of materials or accuracy of measurement, pursued for its own sake in the controlled environment of the research laboratory. The practicing engineer has to be content to work around the impurities and inaccuracies of the real world. The value of purity and accuracy (and even correctness) are often not appreciated until after the scientist has shown that they are achievable.

Sir Charles Antony Richard Hoare

Calvin: You can't just turn on creativity like a faucet. You have to be in the right mood.
Hobbes: What mood is that?
Calvin: Last-minute panic.

Bill Watterson

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I owe so much to my incredible advisor Sorin Lerner for his constant support and guidance. From first meeting at Visit Day in 2007 to writing this thesis seven years later, his enthusiasm, creativity, and dedication have made every interaction a joy. Whenever I encountered hardships, technical or otherwise, Sorin was always there to patiently listen and offer sound advice. My highest aspiration is to pay it forward and emulate his leadership in my own research group.

I have also benefited tremendously from the brilliant tutelage of Ranjit Jhala. His uncompromising insistence on clarity and simplicity has radically shaped my writing and presentation styles. Ranjit not only provided the direct and sometimes rather candid feedback I needed, but also invariably offered kind, encouraging words at exactly the right moments.

Special thanks to Geoff Voelker and Stefan Savage for making UCSD such a spectacular, fun place to do a PhD and for their help tackling the challenges of the academic job market.

The UCSD Programming Systems group truly became family during my time in San Diego. You are all amazing! In particular, working with Don Jang has been an honor. His endless good humor and incredible, relentless work ethic are an inspiration. I especially must thank Ravi Chugh and Patrick Rondon for carrying me through the darkest of times. Their incredible friendship and empathy saved the day. Huge thanks to the brilliant Alexander Bakst and Valentin Robert for unending hijinks, bad jokes, and good memories  . Ross Tate will always have my gratitude for his patient explanations which magically made otherwise opaque concepts clear. One of my earliest collaborators, Sudipta Kundu, made me look forward to a day when I'd totally mastered research, and one of my most recent collaborators, Daniel "Danger" Ricketts, helped me realize that even if we never completely get there, the journey is reward enough.

I'm also incredibly grateful to many friends from across the CSE department and San Diego. In particular, Kirill Levchenko is the most adventurous person I know, fearlessly exploring all things interesting, forbidden or otherwise. My most dangerous experiences in San Diego invariably involve Kirill. In addition to providing a steady adrenaline rush, Kirill is perhaps the most empathic person I have ever met. I also want to thank Karyn Benson for many terrible puns, several good homemade salsas, and many, many excellent miles run together on the trails. Neha Chachra, Sarah Meikeljohn, Maya Rosas, and Brenna Lanton were responsible for many cherished, challenging, and supportive conversations over superb dinners.

Thanks to Stephen Checkoway for his superb UCSD thesis template!

Finally, thanks to my parents for their unconditional love and support and for being so understanding of radio silence during paper deadlines.

Now for the boring stuff:

Chapter 1, in full, is adapted from material as it appears in Programming Language Design and Implementation 2010. Tatlock, Zachary; Lerner, Sorin, ACM Press, 2010. The dissertation author was the primary investigator and author of this paper.

Chapter 2, in full, is adapted from material as it appears in Usenix Security Symposium 2012. Jang, Dongseok; Tatlock, Zachary; Lerner, Sorin, Usenix Association, 2012. The dissertation author was a primary investigator and author of this paper.

Chapter 3, in full, is adapted from material as it appears in Programming Language Design and Implementation 2014. Rickets, Daniel, Robert, Valentin, Jang, Dongseok; Tatlock, Zachary; Lerner, Sorin, ACM Press, 2014. The dissertation author was a primary investigator and author of this paper.

Chapter 4, includes and expands upon material selected from the above three works.

VITA

2007	Bachelor of Science, Purdue University
2007–2013	Research Assistant, University of California, San Diego
2010	Internship, Microsoft Research, Bangalore, India
2013–2014	Acting Assistant Professor, University of Washington
2014	Doctor of Philosophy, University of California, San Diego

ABSTRACT OF THE DISSERTATION

Reducing the Costs of Proof Assistant Based Formal Verification
***or*: Conviction without the Burden of Proof**

by

Zachary Lee Tatlock

Doctor of Philosophy in Computer Science

University of California, San Diego, 2014

Professor Sorin Lerner, Chair

This thesis considers the challenge of fully formal software verification in the demanding and foundational context of mechanical proof assistants. While this approach offers the strongest guarantees for software correctness, it has traditionally imposed tremendous costs to manually construct proofs. In this work, I explore techniques to mitigate this proof burden through careful system design. In particular, I demonstrate how formal shim verification and extensible compiler techniques can radically reduce the proof burden for realistic implementations of critical modern infrastructure.

Introduction

As our dependence on software grows, so too do the risks posed by programming errors. Broadly, my PhD research has aimed to mitigate these risks by improving software reliability and security through novel systems designs that mitigate the proof burden in the foundational context of mechanical proof assistants. I have developed several new techniques to ensure program correctness in domains ranging from embedded database query languages [80] and compiler optimizations [44, 78] to web browsers [36]. In addition to providing new verification techniques, these works provide practical tools which demonstrate the effectiveness of my techniques in real settings.

This thesis seeks to both (1) extend the frontier of tractable verification problems and (2) provide tools which make verification accessible even to non-expert programmers. To achieve these goals, I have focused on systems designs that allow verification engineers to focus their effort by identifying "verification pinch points".

0.1 Provable Security Guarantees via Formal Shim Verification

Web browsers increasingly dominate computer use, mediating access to valuable private data in applications ranging from banking and health care to social networking. Unfortunately, due to programming errors, browsers have consistently proven vulnerable to attackers who exfiltrate private data and take over the underlying system. One technique can eliminate such dangerous, exploitable programming errors: fully formal verification

in a proof assistant. This severe discipline requires the programmer to write their code in the logic of a theorem prover, formally specify its correctness, and then interactively prove that the code always satisfies the correctness properties. Unlike other techniques, the developer reasons about the code which is actually run, not a simplified model. By forcing the programmer to handle every corner case in gritty detail, this approach provides extremely strong guarantees. Unfortunately, this also makes formal verification in a proof assistant prohibitively expensive for massive applications like web browsers: it is infeasible to write and verify the millions of lines of code that make up modern HTML renderers, JavaScript interpreters, JPEG decoders, etc.

To address these challenges, I developed a technique called *formal shim verification* [36] which enabled us to formally prove important security guarantees for a realistic web browser dubbed QUARK. By applying formal shim verification, we can make strong guarantees about massive systems while only formally reasoning about a small fraction of the code. For the browser, we isolate all large, complex components (over a million lines of code) to run in sandboxes which prevent them from accessing any private data or other sensitive system resources. These sandboxed browser components must interact with one another and the underlying system via a *browser kernel* which ensures that all accesses satisfy our security policies. Relative to the other browser components, the QUARK browser kernel is tiny (only a few hundred lines), small enough that we were able to verify it in Coq, a widely used proof assistant. We formally proved that the QUARK kernel ensures several important security properties, including tab non-interference, cookie integrity and confidentiality, and address bar integrity. By sandboxing all untrusted browser components we avoid reasoning about their internal details, which leaves us free to adopt state-of-the-art implementations (*e.g.,* WebKit) for the renderer, JavaScript engine, etc. Thus QUARK is able to run popular, interactive websites like Google Maps, GMail, Facebook, and Amazon. We tested QUARK on the Alexa top 10 websites and

demonstrated that it successfully runs at interactive speeds.

0.2 Compiler Correctness

Nearly every line of code we write is first transformed by an interpreter or compiler. Unfortunately, bugs in such tools are "contagious": compiler errors could change the behavior of a program during compilation, invalidating any correctness guarantees established on the source. While such errors may seem unlikely in practice, Yang et al. [87] evaluated several mature, widely used compilers, including GCC and LLVM, and found hundreds of miscompilation bugs, many of which were caused by erroneous optimizations. Developing reliable optimizations remains a difficult and error-prone process for two primary reasons. First, simply expressing an optimization is challenging: the high-level view of a transformation is inevitably obscured by implementation details. Second, ensuring that an optimization preserves program behavior for all programs on all inputs requires tedious and thorough reasoning which must be repeated every time the optimization is modified. These challenges hinder the development of compiler technology, leading to hand-optimized code, which incurs higher maintenance costs, or, in the case of mission and safety-critical applications (*e.g.,* in health care or avionics), disabling compiler optimizations altogether. Below I describe some of my work to address these challenges.

Parameterized Equivalence Checking. To address the challenges of writing compiler optimizations, I developed a new technique called *parameterized equivalence checking* (PEC) [44] which is able to fully automatically check the correctness of many well known, sophisticated loop optimizations including software pipelining. Despite a long line of research on expressing and checking compiler optimizations, even relatively simple loop optimizations like fusion, distribution, and reversal were not previously addressed by easy-to-use, fully automated techniques. Thus, PEC extends the state

of the art by providing an expressive domain specific language (DSL) which both (1) enables even non-compiler experts to write optimizations as intuitive rewrite rules and (2) automatically checks the correctness of these optimizations. PEC proves optimizations correct by leveraging an SMT sovler (Z3) to statically prove that, for any run of the compiler, the transformation will always preserve program behavior. My work on this project was guided by earlier work I did with my co-authors on Equality Saturation [76], a new optimization technique which I helped extend to dynamically check optimization correctness, and applied to find a bug in a mature Java optimizer.

Extensibility for Formally Verified Compilers. In the study by Yang et al. [87], one compiler emerged radically more reliable than all competitors: the formally verified CompCert C compiler. CompCert's high reliability was achieved by heroic proof effort: 70% of its development (roughly 35,000 lines) are devoted to proving correctness in the Coq proof assistant. Unfortunately, like many formally verified systems, CompCert suffers from *formality inertia*: it is difficult to add new optimizations because they require new correctness proofs and once added, it is difficult to modify optimizations because such modifications require updating correctness proofs. In fact, several PLDI and POPL papers have been written on the verification of particular optimizations in CompCert, a clear indication that they are difficult to write and prove correct. With such extreme challenges, it may seem unlikely that the performance of CompCert-generated code will ever catch up to GCC or LLVM which each employ a host of sophisticated optimizations.

I extended my previous work on PEC to mitigate formality inertia in verified compilers. I built and formally verified an extension of CompCert called XCert [78] which is able to run optimizations expressed in PEC's DSL. To ensure that XCert maintains CompCert's strong correctness guarantees, I formalized my previous correctness results for PEC in Coq and formally proved that the checks PEC carries out on a rewrite rule guarantee that it preserves program behavior. This required careful reasoning about

non-termination and impure, effectful computations. The result was that an entire class of sophisticated loop optimizations (including loop unswitching, skewing, and distribution) could be added to CompCert and guaranteed to be correct without the optimization author writing a single line of Coq proof code.

0.3 Outline

The next chapter considers extensible verified compilers. After that, I focus on web browsers and detail my formal shim verification technique. Later still, I show how this design can be generalized to support the large and important class of reactive systems. After these chapters, I survey related work and then concluded.

Chapter 1

Extensible Verified Compilers

Verified compilers, such as Leroy's CompCert, are accompanied by a fully checked correctness proof. Both the compiler and proof are often constructed with an interactive proof assistant. This technique provides a strong, end-to-end correctness guarantee on top of a small trusted computing base. Unfortunately, these compilers are also challenging to extend since each additional transformation must be proven correct in full formal detail.

At the other end of the spectrum, techniques for compiler correctness based on a domain-specific language for writing optimizations, such as Lerner's Rhodium and Cobalt, make the compiler easy to extend: the correctness of additional transformations can be checked completely automatically. Unfortunately, these systems provide a weaker guarantee since their end-to-end correctness has not been proven fully formally.

We present an approach for compiler correctness that provides the best of both worlds by bridging the gap between compiler verification and compiler extensibility. In particular, we have extended Leroy's CompCert compiler with an execution engine for optimizations written in a domain specific language and proved that this execution engine preserves program semantics, using the Coq proof assistant. We present our CompCert extension, XCert, including the details of its execution engine and proof of correctness in Coq. Furthermore, we report on the important lessons learned for making the proof

development manageable.

1.1 Introduction

Optimizing compilers are a foundational part of the infrastructure developers rely on every day. Not only are compilers expected to produce high-quality optimized code, but they are also expected to be correct, in that they preserve the behavior of the compiled programs. Even though developers hit bugs only occasionally when using mature optimizing compilers, getting compilers to a level of reliability that is good enough for mainstream use is challenging and extremely time consuming. Furthermore, in the context of safety-critical applications, e.g. in medicine or avionics, compiler correctness can literally become a matter of life and death. Developers in these domains are aware of the risk presented by compiler bugs; imagine the care you would take in writing a compiler if a human life depended on its correctness. To guard against disaster, engineers often disable compiler optimizations, perform manual reviews of generated assembly, and conduct exhaustive testing, all of which are expensive precautions.

One approach to ensure compiler reliability is to implement the compiler within a proof assistant like Coq and formally prove its correctness, as done in the CompCert verified compiler [49]. Using this technique provides a strong end-to-end guarantee: each step of the compilation process is fully verified, from the first AST transformation, through register allocation, and down to assembly generation. Unfortunately, because the proofs are not fully automated, this technique requires a large amount of manual labor by developers who are both compiler experts and comfortable using an interactive theorem prover. Furthermore, extending such a compiler with new optimizations requires proving each new transformation correct in full formal detail, which is difficult and requires substantial expertise [83, 82, 81].

Another approach to compiler reliability is based on using a domain-specific

language (DSL) for expressing optimizations; examples include Rhodium [47] and PEC [44]. These systems are able to automatically check the correctness of optimizations expressed in their DSL. This technique provides superior extensibility: not only are correctness proofs produced without manual effort, but the DSL provides a convenient abstraction for implementing new optimizations. In fact, these systems are designed to make compilers extensible even for non-compiler experts. Unfortunately, the DSL based approach provides a weaker guarantee than verified compilers, since the execution engine that runs the DSL optimizations is not proved correct.

In this chapter we present a hybrid approach to compiler correctness that achieves the best of both techniques by bridging the gap between verified compilers and compiler extensibility. Our approach is based on a DSL for expressing optimizations coupled with both a fully automated correctness checker and a verified execution engine that runs optimizations expressed in the DSL. We demonstrate the feasibility of this approach by extending CompCert with a new module XCert ("Extensible CompCert"). XCert combines the DSL and automated correctness checker from PEC [44] with an execution engine implemented as a pass within CompCert and verified in Coq.

XCert achieves a strong correctness guarantee by proving the correctness of the execution engine fully formally, and also provides excellent extensibility because new optimizations can be easily expressed in the DSL and then checked for correctness fully automatically. In particular, while adding only a relatively small amount to CompCert's trusted computing base (TCB), our technique provides the following benefit: additional optimizations that are added using PEC *do not require any new manual proof effort, and do not add anything to the TCB*.

The main challenge in adding a PEC execution engine to CompCert lies in verifying its correctness in Coq. The verification is difficult for several reasons. First, it introduces new constructs into the CompCert framework including parameterized

programs, substitutions, pattern matching, and subtle CFG-manipulation operations. These constructs require careful design to make reasoning about the execution engine manageable. Second, the execution engine imports correctness guarantees provided by PEC into CompCert, which requires properly aligning the semantics of PEC and CompCert. Third, applying the PEC guarantee within the correctness proof of the engine is challenging and tedious because it requires knowing information outside the engine about tests performed deep within the engine.

We discuss three general techniques that we found extremely useful in mitigating these difficulties: (1) *Verified Validation*, a technique inspired by Tristan et al, where, for certain algorithms in the PEC engine, we reduce proof effort by implementing a verified result checker rather than directly verifying the algorithm; (2) *Semantics Alignment*, where we factor out into a separate module the issues related to aligning the semantics between PEC and CompCert, so that these difficulties do not pervade the rest of the proof; and (3) *Witness Propagation*, where we return extra information with the result of a transformation which allows us to simplify applying the PEC guarantee and reduce case analyses.

Our contributions therefore include:

- XCert, an extension to CompCert based on PEC that provides both extensibility and a strong end-to-end guarantee. We first review PEC and CompCert in Section 1.2, and then present our system and its correctness proof in Sections 1.3 and 1.4.

- Techniques to mitigate the complexity of such proofs and lessons learned while developing our proof (Sections 1.3, 1.4 and 1.5). These techniques and lessons are more broadly applicable than our current system.

- A quantitative and qualitative assessment of XCert in terms of trusted computing base, lines of code, engine complexity and proof complexity, and a comparison

```
k := 0
while (i < 100) {
   a[k] += k;
   k++;
}
```

(a)

```
k := 0
while (k < 99) {
   a[k] += k;
   k++;
}
a[k] += k;
k++;
```

(b)

Figure 1.1. Loop peeling: (a) shows original code, (b) shows transformed code.

using these metrics with CompCert and PEC (Section 1.6).

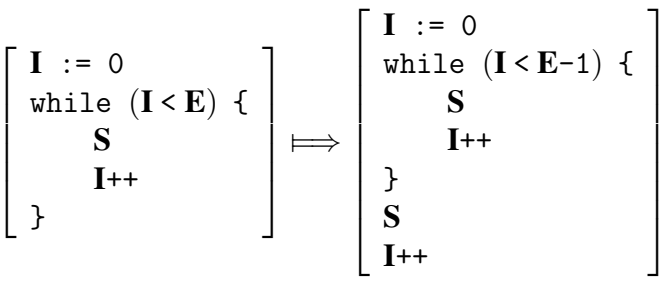
1.2 Background

In this section, we review background on PEC [44] and the CompCert verified compiler [49].

1.2.1 Parameterized Equivalence Checking (PEC)

PEC is a system for implementing optimizations and checking their correctness automatically. PEC provides the programmer with a domain-specific language for implementing optimizations. Once optimizations are written in this language, PEC takes advantage of the stylized forms of the optimizations to check their correctness automatically.

Loop peeling We show how PEC works through a simple example, loop peeling. Loop peeling is a transformation that takes one iteration of a loop, and moves it either before or after the loop. An instance of this transformation is shown in Figure 1.1. Loop peeling can be used for a variety of purposes, including modifying loop bounds to enable loop unrolling or loop merging.



where $NotMod(\mathbf{S},\mathbf{I}) \wedge NotMod(\mathbf{S},\mathbf{E}) \wedge StrictlyPos(\mathbf{E})$

Figure 1.2. Loop peeling expressed in PEC

Optimizations in PEC are expressed as guarded rewrite rules of the following form:

$$G_\ell \Longmapsto G_r \textbf{ where } S$$

where G_ℓ is a code pattern to match, G_r is the code to replace any matches with, and the side condition S is a boolean formula stating the condition under which the rewrite may safely be performed. Throughout the chapter we use subscript ℓ (which stands for "left") for the original program and subscript "r" (which stands for "right") for the transformed program. Figure 1.2 shows a simple form of loop peeling, expressed in PEC's domain-specific language. The variables **S**, **I** and **E** are PEC *pattern variables* that can match against pieces of concrete syntax: **S** matches statements, **I** variables, and **E** expressions.

The semantics of a rewrite rule $G_\ell \Longmapsto G_r$ **where** S is that, for any substitution θ mapping pattern variables to concrete syntax, if $\theta(G_\ell)$ is found somewhere in the original program (where $\theta(G_\ell)$ denotes applying the substitution θ to G_ℓ to produce concrete code), then the matched code is replaced with $\theta(G_r)$, as long as $S(\theta(G_\ell), \theta(G_r))$ holds.

The side condition S is a conjunction over a fixed set of *side condition predicates*, such as *NotMod* and *StrictlyPos*. These side condition predicates have a fixed semantic meaning – for example, the meaning of $StrictlyPos(\mathbf{I})$ is that **I** is greater than 0. PEC

trusts that the execution engine provides an implementation of these predicates that implies their semantic meaning: if the implementation of the predicate returns true, then its semantic meaning must hold.

Correctness checking PEC tries to show that a rewrite rule $G_\ell \Longmapsto G_r$ **where** S is correct by matching up execution states in G_ℓ and G_r using a simulation relation. A simulation relation $\sim$ is a relation over program states in the original and transformed programs. Intuitively, $\sim$ relates a given state η_ℓ of the original program with its corresponding state η_r in the transformed program.

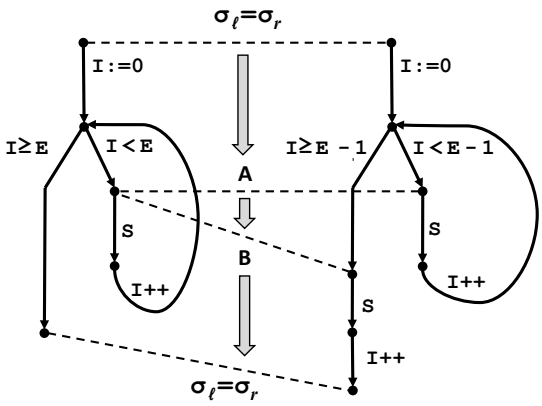
The key property to establish is that the simulation relation is preserved throughout execution. Using $\rightarrow$ to denote small-step semantics, this property can be stated as follows:

$$\eta_\ell \sim \eta_r \wedge \eta_\ell \rightarrow \eta'_\ell \Rightarrow \exists \eta'_r,\, \eta'_\ell \sim \eta'_r \wedge \eta_r \rightarrow \eta'_r \tag{1.1}$$

Essentially, if the original and transformed programs are in a pair of related states, and the original program steps, then the transformed program will also step, in such a way that the two resulting states will be related. Furthermore, if the original states of the two programs are related by $\sim$, then the above condition guarantees through an inductive argument over program traces that the two program always executes in lock step on related states.

Figure 1.3 shows G_ℓ and G_r for loop peeling, and shows the simulation relation that PEC automatically infers for this example. G_ℓ and G_r are shown in CFG form, where a node is a program point, and edges are statements. A dashed edge between G_ℓ and G_r indicates that the program points being connected are related in the simulation relation. Furthermore, each dashed edge is labeled with a formula showing how the heaps σ_ℓ and σ_r (of G_ℓ and G_r) are related at those program points.

The entry and exit points are related with state equality ($\sigma_\ell = \sigma_r$), which means



A(σℓ , σr)	σℓ=σr ∧ eval(σℓ , I < E) ∧ eval(σr , I < E-1)
B(σℓ , σr)	σℓ=σr ∧ eval(σℓ , I < E) ∧ eval(σr , I ≥ E-1)

Figure 1.3. Simulation relation for loop peeling

that the simulation relation shows that if G_ℓ and G_r start in equal states, then they will end in equal states (if the exit points are reached). Aside from the entry points, there are two other entries in the simulation relation, labeled with formulas A and B in Figure 1.3 (shown below the CFGs). The notation $\texttt{eval}(\sigma, e)$ represents the result of evaluating expression e in heap σ.

The PEC checker takes as input the rewrite rule shown in Figure 1.2, and it automatically generates the relation shown in Figure 1.3. After generating this relation, PEC checks that the relation satisfies the properties required for it to be a simulation relation, namely property (1.1). PEC does this by enumerating the paths from each simulation relation entry to other entries that are reachable. In this case, there are five such paths: entry to A, entry to B, A to A, A to B, and B to exit. While enumerating paths, PEC prunes infeasible ones. For example, PEC prunes the path "A to exit", because the simulation entry at A tells us that $\mathbf{I} < \mathbf{E} - 1$, which after executing **I++** in the original program gives $\mathbf{I} < \mathbf{E}$, which forces the original program to go back into the loop. For each feasible path that PEC enumerates, PEC shows using an automated theorem prover

(more specifically an SMT solver) that if the original and transformed programs start executing at the beginning of the path, in related heap states, then they end up in related heap states at the end of the path. One important property of the simulation relation is that all loops are cut, and so there are no loops between entries in the simulation relation. As a result, the SMT solver only has to reason about short sequences of straight line code, which SMT solvers do very well in a fully automated way.

Guarantee provided by PEC The PEC work [44] initially considered the following as its correctness guarantee: starting with any initial heap σ, if the original program executes to its exit and yields heap σ', then the transformed program will also execute to its exit and produce the same σ'. However, as we will show in Section 1.3, this fails to capture the correctness guarantee that PEC in fact provides for non-terminating computations. As a result, to integrate PEC within CompCert and prove the PEC execution engine correct, particularly for non-terminating computations, we will have to update the interface of the PEC checker so that it also returns the simulation relation it discovered.

The techniques we present in this chapter work for the "Relate" module from PEC, which accounts for about three quarters of the optimizations presented in [44]. The remaining optimizations, which include some of the more sophisticated loop optimizations like loop reversal, are handled by the PEC "Permute" module, which presents additional challenges that we leave for future work.

1.2.2 CompCert

We now give a brief overview of the CompCert [49] compiler. CompCert takes as input Clight, a large subset of C, and produces PowerPC or ARM assembly. The compiler is implemented inside the Coq proof assistant. CompCert is organized into several stages that work over a sequence of increasingly detailed intermediate representations (IRs):

from various C-like AST representations, through CFG based representations like `RTL`, and finally down to abstract syntax for PowerPC assembly.

CompCert is accompanied by a proof of correctness, also implemented in Coq. This proof provides a strong end-to-end correctness guarantee. The guarantee is *strong* because the entire proof is formalized in Coq, not leaving any parts to a paper-and-pencil proof. The guarantee is *end-to-end* because it covers all the steps of compilation, from the source language all the way to assembly code.

The proof is organized around CompCert's compilation stages. For each stage, there is a proof showing that if the input program to the stage has a certain behavior, then the program produced by the stage will have the same behavior. The particular details of how each proof is done depends on the particular stage and the semantics of the input and output IR for the stage. The individual proofs are then composed together to produce an end-to-end correctness argument.

A common strategy used in CompCert for proving optimizations correct is to use a simulation relation. For each optimization that the programmer wants to add, the programmer must carefully craft a simulation relation for the optimization, and prove that it satisfies property (1.1) in Coq. Once this is done, CompCert has several useful theorems about small-step semantics that allows the programmer to conclude that the semantics is preserved by the optimization.

In general, proving property (1.1) requires a substantial amount of manual effort, and more importantly, it requires in depth knowledge of Coq, CompCert's data-structures, and proof infrastructure provided by CompCert. In contrast, in the PEC system, once the checker has been implemented, new optimizations can be checked for correctness fully automatically, with no manual proof effort.

1.3 XCert: CompCert + PEC

We have seen in Section 1.2.1 how PEC provides extensibility, and in Section 1.2.2 how CompCert provides strong guarantees. We now give an overview of how XCert extends CompCert with PEC to get both extensibility and a strong correctness guarantee. This section gives a high-level informal description of the approach, whereas Section 1.4 will describe the formalism as implemented in Coq.

Our general approach is to implement an execution engine for PEC optimizations in CompCert, and prove that this execution engine preserves semantics, given that the optimizations being executed have successfully been checked using PEC.

1.3.1 Execution engine

To add a PEC engine to CompCert, we must decide where in CompCert's compilation pipeline the PEC engine should be added. Although there are many different points in the pipeline, each using a different IR, the decision really comes down to picking between a CFG-based IR and an AST-based IR.

We decided to apply PEC optimizations to the `RTL` intermediate representation, which is CompCert's highest level CFG-based IR. This is also the IR over which CompCert's primary optimizations work: the `RTL` stage in the compilation pipeline is perfectly suited for implementing general optimizations because all of the source language constructs have been compiled away, but none of the target specific details have yet been introduced. Although running PEC optimizations on a CFG has many benefits, it also presents several challenges.

Pattern matching First, pattern matching is more difficult on a CFG than an AST. At a high-level, given a rewrite rule $G_\ell \Longmapsto G_r$ **where** S, the PEC execution engine must find occurrences of G_ℓ in the program being optimized. An AST pattern-matcher is

quite simple to implement recursively using a simultaneous traversal over the pattern and the expression being matched. A CFG pattern matcher, on the other hand, is more complex, primarily because CFGs can have cycles, whereas ASTs are acyclic. Not only does this make the pattern matcher itself more complex, but reasoning about it formally also becomes more difficult.

Verified Validation To address the challenge of reasoning about a CFG-based pattern matcher, we make use of *Verified Validation*, a technique inspired by the work of Tristan et al. on verified translation validation [83, 82, 81]. The insight is that the result checker for an algorithm is often much simpler than the algorithm itself, and so proving the result checker correct is often much simpler than proving the algorithm correct. In our context, *Verified Validation* allows us to produce matches that are guaranteed to be correct, while only reasoning about a pattern-match result checker, rather than the pattern matcher itself.

Transforming the CFG The second challenge in executing PEC optimizations on a CFG is that a CFG is more difficult to transform than an AST, and this difficulty is reflected in the Coq proof of correctness. Because ASTs are trees with no cycles or sharing, one can easily perform transformations locally, replacing a whole subtree with another subtree. In a CFG, however, replacing one subgraph with another requires appropriately connecting incoming and outgoing edges for the region that has been replaced.

To make this task as easy as possible, we take advantage of the way that CFGs are represented in CompCert. A CFG in CompCert is a map from program points to instructions, and each instruction contains successor program points. For example, a branch instruction would contain two successor program points, whereas a simple assignment would only contain one successor program point. Consider for example the

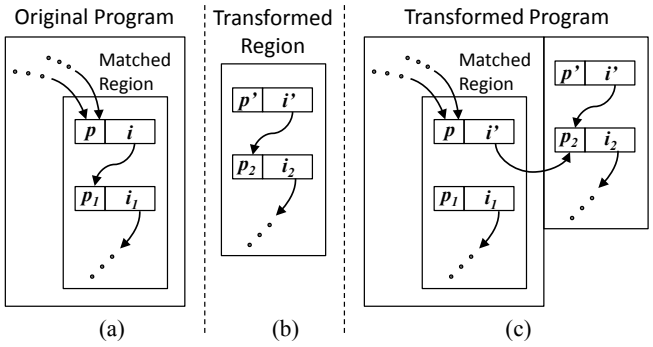


Figure 1.4. Example of CFG splicing

original CFG shown in Figure 1.4(a), with a matched region of the CFG that we want to transform. We graphically display each entry in a CompCert CFG as a box that is sub-divided into two parts: the left part of the box contains a program point p and the right part the instruction i that the program point is mapped to. We use arrows from an instruction directly to its successor program points.

Side Conditions As noted in Section 1.2.1, PEC relies on the execution engine to provide correct implementations for a fixed set of side conditions predicates, which are used to create the side conditions of the PEC rewrite rules. For achieving a strong correctness guarantee, it is crucial that the implementation of these side condition predicates be verified. To this end, we have implemented and verified a handful of side condition predicates, e.g. *NotMod* and *StrictlyPos* from Figure 1.2.

Parametrized CFGs Given a PEC rewrite rule $G_\ell \Longmapsto G_r$ **where** S, we represent G_ℓ and G_r as *parametrized* CFGs. A parametrized CFG (PCFG) is a CompCert CFG that can contain pattern variables like **S**, **E**, and **I**, which must be instantiated to get a concrete CFG. Furthermore, these PCFGs also use pattern variables wherever a program point would be expected. Thus, when the PEC engine finds a match for the loop-peeling rewrite from Figure 1.2, the resulting substitution not only states what **S**, **E**, and **I** map to, but

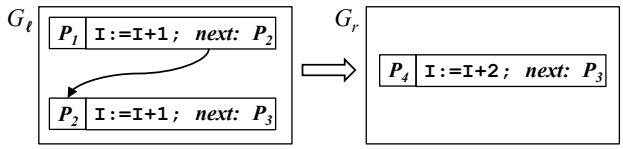


Figure 1.5. PEC rewrite rule using Parameterized CFGs

also how the program points of G_ℓ map to program points of the CFG being transformed.

For example, Figure 1.5 shows how the rewrite rule **I++;I++** $\Longmapsto$ **I+=2** would be represented using PCFGs. Note that the transformed PCFG, namely G_r, contains a program point pattern variable $\mathbf{P}_4$ that is not bound in the original PCFG, namely G_ℓ. Such unbound pattern variables (of which there can be many in the transformed PCFG) represent fresh program points that the engine will need to generate when it applies the transformation. Although in general it's perfectly legal for two pattern variables to map to the same piece of concrete syntax, these unbound program points have a special semantics, in that the engine generates a fresh (and thus distinct) program point for each unbound program point pattern variable.

For simplicity of presentation, we will assume that all parametrized program points in the domain of G_r (i.e. program points in the left parts of the boxes in the diagrams) must be free, in that they do not appear in G_ℓ. This makes the example easier to understand intuitively and slightly simplifies the formalization in Section 1.4. Our actual implementation in Coq does not make this assumption.

Connecting outgoing edges To see how we connect edges leaving the transformed region, let's take a look at Figure 1.5 again. Note that the transformed PCFG uses the pattern variable $\mathbf{P}_3$, which *is* bound in the original PCFG. Thus, when the PEC engine finds a match for G_ℓ in Figure 1.5, the resulting substitution will have an entry for $\mathbf{P}_3$, which essentially captures the fall-through of the matched region of code. When the

engine applies this substitution to G_r, to produce the transformed region of code, $\mathbf{P}_3$ will be replaced with the fall-through of the original region. In this way, the regular match-and-transform process in the PEC engine naturally connects outgoing edges in the transformed region, without requiring a special case.

Connecting incoming edges For connecting edges entering the transformed region, let's go back to Figure 1.4(a), and suppose the pattern matcher has found a sub-CFG g_ℓ in the original CFG that matches G_ℓ, and let's assume that the resulting substitution is θ. Furthermore, suppose that applying θ to G_r produces the replacement CFG shown in Figure 1.4(b). As mentioned previously, the engine generates new fresh program points in the transformed CFG, which means that we can simply union the CFG from Figures 1.4(a) and (b) without any name clashes in the program points. Furthermore, after this union is performed, outgoing edges of the replacement CFG are already connected, as mentioned previously. As a result, we are only left with connecting the incoming edges.

Our approach to doing this is simple yet effective. In particular, we take the entry program point in the matched region from Figure 1.4(a) and *update* the instruction at that point with the first instruction of the replacement region from Figure 1.4(b). Figure 1.4(c) shows the result of this process. In essence, instruction i' has been copied to the entry of the matched region, and since i' contains inside of it all its successor program points, the instruction at p now has successor links pointing directly into the transformed region. The remainder of the original matched region is left unchanged, although disconnected (except if there are other entry points into the matched region). Any unreachable code will be removed by a subsequent dead code elimination phase. Note that in our example, the program point p' is also left disconnected, but this does not have to be the case in general, since instructions from the transformed region may point to it (for example, in

the case of a loop).

Witness Propagation In general, applying the PEC guarantee within the Coq correctness proof of the execution engine is challenging and tedious because it requires knowing information outside the engine about tests performed deep within the engine. To facilitate the task of applying the PEC guarantee, we use *Witness Propagation*, a technique in which functions are made to return additional information that is used only for reasoning purposes. For example, we make the PEC execution engine in CompCert return not only the final transformed CFG, but also the substitution that was used to generate this transformed CFG. When executing the compiler, the substitution is not used outside the engine; however, in the proof it makes applying the PEC guarantee much easier, and it simplifies case analysis for code that calls the execution engine.

1.3.2 Correctness

Recall that optimizations at the `RTL` level are proved correct in CompCert using a simulation relation, and this amounts to showing property (1.1) in Coq, where η_ℓ and η'_ℓ are states in the original program, and η_r and η'_r are states in the program produced by the PEC execution engine. When performing this proof in Coq, we assume that all the rules executed by the engine have been checked successfully by PEC, and therefore, we know that the correctness condition provided by PEC holds for those rules (outlined in Section 1.2.1).

One of the challenges that comes up in performing this proof is that the original program and the transformed program don't execute in perfect synchrony anymore with respect to the small-step semantics $\rightarrow$: given a piece of code that has been transformed, it may take, say, 5 steps to go through it in the original program, and only 2 steps in the transformed one. This misalignment in the semantics means that, strictly speaking,

property (1.1) does not hold. Although CompCert has stuttering variations of (1.1) that can be used in this case, using these variations makes the proof more complex, but more importantly it also conflates issues: the proof would have to deal at the same time with the misalignment of semantics, and with the complexities of reasoning about PEC rewrites.

Semantics alignment To separate these concerns, and to modularize the proof, we introduce two new semantics for the purposes of *Semantics Alignment*, $\rightarrow_\ell$ and $\rightarrow_r$, which are meant to align exactly: each step taken by $\rightarrow_\ell$ should correspond to precisely one step of $\rightarrow_r$, making it easier to show the equivalence of $\rightarrow_\ell$ and $\rightarrow_r$. In a separate *Semantics Alignment* module, we can then show the equivalence between $\rightarrow$ and $\rightarrow_\ell$ for the original program, and between $\rightarrow_r$ and $\rightarrow$ for the transformed program.

Our first attempt at defining $\rightarrow_\ell$ and $\rightarrow_r$ unfortunately was not strong enough. In particular, we stated that $\rightarrow_\ell$ and $\rightarrow_r$ act like $\rightarrow$, but step "over" any regions of code transformed by PEC in the original or optimized programs, respectively. Although this approach works well for terminating computations, non-terminating computations introduce additional challenges. When CompCert proves that an optimization preserves behavior, the definition of behavior includes the possibility of running forever (with a infinite trace of externally visible events, such as calls to `printf`). Thus, we need to prove that the PEC engine preserves non-terminating behaviors (including the details of the infinite trace). In general formally reasoning about the preservation of non-termination has proven challenging in the context of formally verified compilers. Indeed, many verified compilers, for example the recent work of Chlipala [18], still don't have a proof that non-termination is preserved.

The big-step problem The problem with our original definition of $\rightarrow_\ell$ and $\rightarrow_r$ in regards to non-termination is that they take a big step over regions that PEC has trans-

formed, and such a big step does not provide a guarantee when the program gets into an infinite loop inside these "stepped over" regions. The checks that PEC performs does however guarantee that non-termination is preserved inside of the regions it transforms. Thus, one way to address this problem is to strengthen the original guarantee provided by the PEC work (stated in Section 1.2.1), using a similar approach to what CompCert does at the optimization level: define the behavior of a region of code as either "terminates" or "runs forever". The guarantee that PEC provides would then state that the behavior of a region transformed by PEC is preserved, which would include the "runs forever" case.

While pursuing this approach, we realized that the proof was getting unwieldy. Applying the new PEC correctness guarantee was difficult because in the non-terminating case, CompCert requires the proof to produce the infinite trace in the transformed program, which in turn requires a lot of accounting to properly "glue" traces together. The complexity is in part due to the fact that different kinds of traces must be glued together: finite (inductively defined) traces with infinite (co-inductively defined) traces.

By carefully observing the challenges in the proof, we realized that, in the end, all the problems stemmed from a single mismatch in the semantics: big-step vs. small step. The CompCert `RTL` theory works using a small-step semantics, and our "step-over" approach essentially introduces a big step over potentially non-terminating computations.

Changing the PEC interface Our solution to this problem is another instance of the *Semantic Alignment* technique, where we essentially change the PEC interface so that it aligns with CompCert's small-step proofs. The key to achieving this alignment stems from the realization that PEC actually performs its checking using small steps. In particular, the simulation relation that PEC generates has the property that there are no loops between entries. If there is a loop, PEC will generate an entry in the simulation relation that cuts the loop into acyclic paths, in much the same way that a loop invariant

cuts loops in program verification. Entry *A* in Figure 1.3 is such a loop-cutting entry in the simulation relation. Therefore, there is no possibility that a program will not terminate *between* simulation relation entries. Furthermore, PEC uses a simulation relation in its checking, which is precisely the technique used in CompCert too. It would therefore make sense to change the interface between the two systems to take advantage of their similarities.

To this end, we modify the interface between PEC and CompCert so that the PEC checker *returns* the simulation relation that it used to prove a particular optimization correct, and we import this simulation relation into CompCert. When we prove that running this optimization in CompCert using the PEC execution engine preserves behavior, we can make use of CompCert's simulation relation approach, by creating a simulation relation for the entire program as follows: if we're *not* in a region that has been transformed, use state equality; if we *are* in a region that has been transformed, use the simulation relation returned by the PEC checker for that optimization.

Furthermore, along with the PEC simulation relation, we assume that the PEC checker returns a Coq proof that the simulation relation satisfies the simulation property, namely property (1.1). This proof is nothing more than a Coq reification of the proofs that PEC's SMT solver performed. If PEC used an SMT solver that returned proofs, it could perform a translation from the SMT proofs into Coq's proof language. The proof returned by PEC is used in our proof to show that the simulation relation we created for the entire program is preserved while inside transformed code.

Function calls are handled in CompCert using small steps, so that a call instruction transfers execution to the CFG of the callee. If a call instruction occurs inside the transformed region, we consider the call to essentially leave the transformed region. As a result, inside the callee, the simulation relation we construct will simply use state equality, not the PEC simulation relation. Once the call returns, execution comes back into the

transformed region, and the simulation relation we construct goes back to using the PEC simulation relation.

Left and right semantics, revisited Now that PEC returns a simulation relation, we can give the definitions of $\rightarrow_\ell$ and $\rightarrow_r$ that we use in our proof: if we're *not* in a region that has been transformed, $\rightarrow_\ell$ and $\rightarrow_r$ work the same as $\rightarrow$; if we *are* in a region that has been transformed, $\rightarrow_\ell$ and $\rightarrow_r$ simply step from one entry to another in the simulation relation returned by PEC.

To illustrate how $\rightarrow$, $\rightarrow_\ell$ and $\rightarrow_r$ work, Figure 1.6 shows part of an execution trace $trace_\ell$ for the original program (with round circles for program states), and part of a trace $trace_r$ for the transformed program (with crosses for program states), along with the simulation relation as it unfolds throughout execution (shown as dotted edges between the original and transformed traces). The simulation relation inside the transformed region is the one that PEC returns. Figure 1.6 also shows how the three step semantics operate on the original and transformed programs: $\rightarrow$ and $\rightarrow_\ell$ on the original program and $\rightarrow_r$ and $\rightarrow$ on the transformed program.

1.3.3 Proof architecture

To summarize, our proof is therefore organized into three steps, which we show separately: (1) if a program π has behavior b under $\rightarrow$, then π has behavior b under $\rightarrow_\ell$; (2) if a program π has behavior b under $\rightarrow_\ell$, then the program produced by our execution engine on π has behavior b under $\rightarrow_r$; (3) if a program π has behavior b under $\rightarrow_r$, then π has behavior b under $\rightarrow$. Steps (1) and (3) are where semantics alignment issues are resolved, and step (2) is where we build a simulation relation for the original and transformed programs using the simulation relation returned by the PEC checker.

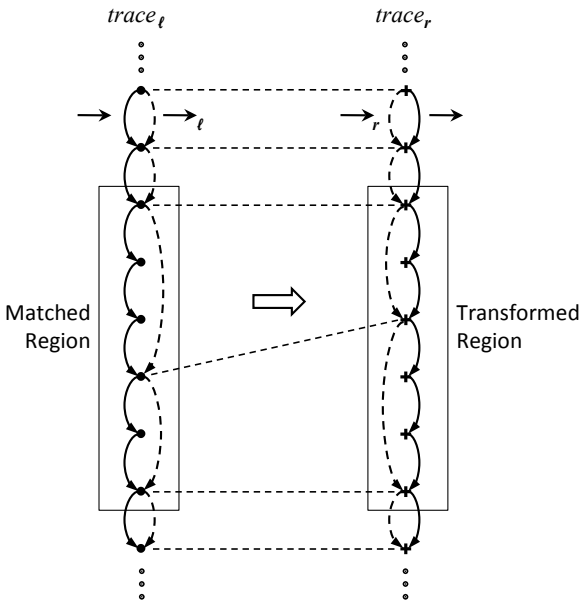


Figure 1.6. Traces showing how $\rightarrow$, $\rightarrow_\ell$ and $\rightarrow_r$ work

1.4 Formalization

In this section, we make the ideas from Section 1.3 more precise, by presenting a formalization of the PEC engine and its proof. The development presented here closely mirrors our implementation in Coq. Later, in Section 1.5, we describe some of the additional challenges that arose when translating these high level ideas into Coq code.

1.4.1 Basic definitions

We start with some basic definitions, shown in Figure 1.7. An instruction i may be any one of a number of basic `RTL` instructions already defined in CompCert. A CFG g is a map from program points to instructions, and a program is map from function names (strings) to CFGs. A program heap σ contains the state of dynamically allocated memory blocks. For simplicity of presentation, we assume the heap also contains the state of the registers and stack, even though in the implementation they are kept separate. A program state η consists of a CFG (representing the current code being executed), a

$$
\begin{array}{rcll}
\text{Instruction} & i & \in & \mathit{Instr} \\
\text{Program point} & p & \in & \mathit{PP} \\
\text{CFG} & g & \in & \mathit{CFG} \quad = \quad \mathit{PP} \rightharpoonup \mathit{Instr} \\
\text{Program} & \pi & \in & \mathit{Prog} \quad = \quad \mathit{String} \rightharpoonup \mathit{CFG} \\
 & & & \\
\text{Program heap} & \sigma & \in & \mathit{Heap} \\
\text{Program state} & \eta & \in & \mathit{State} \quad = \quad \mathit{CFG} \times \mathit{PP} \times \mathit{Heap} \\
\text{PEC Sim Rel} & \psi & \in & \mathit{Sim} \quad = \quad \mathscr{P}(\mathit{State} \times \mathit{State}) \\
 & & & \\
\text{Substitution} & \theta & \in & \mathit{Subst} \\
\text{Param. } \mathit{Sim} & \Psi & \in & \mathit{PSim} \\
\text{Param. } \mathit{CFG} & G & \in & \mathit{PCFG} \\
\text{Side condition} & S & \in & \mathit{SC} \quad = \quad \mathit{CFG} \times \mathit{CFG} \\
\text{Rewrite rule} & r & \in & \mathit{Rule} \quad = \quad \mathit{PCFG} \times \mathit{PCFG} \times \mathit{SC}
\end{array}
$$

Figure 1.7. Common types used in our formalism

program point in that CFG (representing where execution has reached), and the heap (which includes the stack). We project these fields of a program state η as follows: $\mathsf{g}(\eta)$ denotes the CFG, $\mathsf{p}(\eta)$ denotes the program point, and $\mathsf{s}(\eta)$ denotes the heap.

A PEC simulation relation ψ is a relation over program states that is returned by the PEC checker. Because they are generated by PEC, these simulation relations have entries for related program points, and each entry is a predicate over program heaps (recall Figure 1.3). Therefore, such relations have the form:

$$\psi((g_\ell, p_\ell, \sigma_\ell), (g_r, p_r, \sigma_r)) \triangleq \psi_P(p_\ell, p_r)(\sigma_\ell, \sigma_r)$$

where $\psi_P \in (\mathit{PP} \times \mathit{PP}) \rightharpoonup \mathscr{P}(\mathit{Heap} \times \mathit{Heap})$. We use the notation $p \in \psi$ to denote that p is in the domain of ψ_P (either as a first parameter or second parameter).

A substitution θ is a map from pattern variables to concrete pieces of syntax. A parametrized simulation relation Ψ is a version of a simulation relation that contains pattern variables which must be instantiated to yield a concrete simulation relation. For

example, the simulation relation shown in Figure 1.3 is parametrized because syntactic values for **S**, **E**, and **I** must be provided before the simulation relation can apply to concrete program states. Given a parametrized simulation relation Ψ, and a substitution θ that maps every free pattern variable in Ψ to concrete syntax, the result of applying θ to Ψ, denoted $\theta(\Psi)$, is a concrete simulation relation ψ. Similarly, a parameterized CFG G is a parametrized version of a CFG. A side condition is a boolean function from two concrete CFGs (here expressed as a relation). A PEC rewrite rule r contains two parametrized CFGs (representing the pattern to match, and the replacement to perform), and a side condition.

1.4.2 PEC checker and guarantee

PEC takes a rewrite rule and attempts to construct a parameterized simulation relation. If PEC is able to check that the rewrite rule is correct, it also returns a proof that the simulation relation satisfies the simulation property. Specifically, PEC has the type:

$$\mathsf{PEC}(r : \mathit{Rule}) : (\Psi : \mathit{PSim} \times \mathsf{Proof}[\mathsf{IsSimRel}(r, \Psi)]) \cup \{\mathsf{Fail}\}$$

The proof returned by PEC plays a central role in our Coq proof of the correctness of the execution engine. To describe the proof returned by PEC we'll make use of a modified step relation, $\eta \xrightarrow{t}_{\psi} \eta'$, which essentially steps over any program points not in the PEC simulation relation ψ. That is, $\xrightarrow{t}_{\psi}$ combines the sequence of regular $\xrightarrow{t}$ steps from one entry in ψ to the next into a single "medium" step.

Using this definition, we now define $\mathsf{IsSimRel}(r, \Psi)$, the guarantee provided by the proof term returned by PEC:

Definition 1. *We say* $\mathsf{IsSimRel}((G_\ell, G_r, S), \Psi)$ *holds iff:*

$$S(\theta(G_\ell), \theta(G_r)) \Rightarrow \mathsf{IsConSimRel}(\theta(\Psi), \theta(G_\ell), \theta(G_r))$$

```
TrProg(π, r) :
  return λs.
    fst(TrCFG(π(s), r))

TrCFG(g, r) :
  C ← ∅
  for p ∈ ProgPoints(g) do
    x ← TrPoint(g, r, p)
    C ← C ∪ {x}
  return Pick(C)

TrPoint(g_ℓ, (G_ℓ, G_r, S), p) :
  θ ← Match(G_ℓ, g_ℓ, p)
  if ¬ θ(G_ℓ) ≝^p g_ℓ
    return (g_ℓ, ⊥)
  θ ← Fresh(θ, G_r)
  if ¬ S(θ(G_ℓ), θ(G_r))
    return (g_ℓ, ⊥)
  g_r ← g_ℓ ∪ θ(G_r)
  i ← g_r(θ(G_r.entry))
  g_r ← g_r[p ↦ i]
  return (g_r, θ)
```

Figure 1.8. PEC execution engine

where $\mathsf{IsConSimRel}(\psi, g_\ell, g_r)$ *holds iff:*

$$\psi_P(\mathsf{Entry}(g_\ell), \mathsf{Entry}(g_r)) = \mathsf{HeapEq} \wedge$$

$$\psi_P(\mathsf{Exit}(g_\ell), \mathsf{Exit}(g_r)) = \mathsf{HeapEq} \wedge$$

$$\left[\begin{array}{l} g_\ell = \mathsf{g}(\eta_\ell) \wedge g_r = \mathsf{g}(\eta_r) \wedge \\ \psi(\eta_\ell, \eta_r) \wedge \eta_\ell \xrightarrow{t}_\psi \eta'_\ell \end{array}\right] \Rightarrow \left[\exists \eta'_r \,.\, \psi(\eta'_\ell, \eta'_r) \wedge \eta_r \xrightarrow{t}_\psi \eta'_r\right]$$

Intuitively, the above definition guarantees that the simulation relation returned by PEC: (a) relates states on entry and exit to G_ℓ and G_r by heap equality – HeapEq is defined by $\forall\sigma.\mathsf{HeapEq}(\sigma,\sigma)$; and (b) satisfies the simulation property (1.1).

1.4.3 Execution engine

Figure 1.8 shows pseudo code for the PEC execution engine in XCert. Given a program π and a PEC rewrite r, TrProg applies r to each CFG in π using TrCFG. It projects the first element of the result of TrCFG because it contains both the transformed CFG and the substitution used to produce this CFG. TrCFG iterates over all the program

points in the given CFG g, and for each program point it attempts to apply the rewrite starting at that point by calling TrPoint. It gathers the resulting CFGs and chooses one as the transformed version of g.

TrPoint first tries to match the left parameterized CFG G_ℓ of the rewrite rule to the given concrete CFG g_ℓ. It then checks that any generated substitution θ applied to G_ℓ is identical to the CFG fragment of g_ℓ rooted at p; we denote this as $\theta(G_\ell) \stackrel{p}{=} g_\ell$. If this check or the pattern match fails, TrPoint simply returns the original CFG and $\perp$ which indicates an invalid substitution.This instance of *Verified Validation* allows us to avoid reasoning about Match directly and instead simply show that our comparison $\stackrel{p}{=}$ is correct, which is a much smaller proof burden. Next TrPoint creates fresh program points for any parameterized program points that are free in G_r. Now, TrPoint checks that the rewrite rule's side condition holds on the CFGs generated by applying θ to the left and right parameterized programs, G_ℓ and G_r. Once again, if the check fails, TrPoint simply returns the original CFG and $\perp$. Next TrPoint generates the transformed version of the code g_r by applying the substitution θ to the right parameterized CFG G_r. TrPoint then changes g_r so that program location p points to the first instruction of the transformed part of the CFG. Finally, TrPoint returns the transformed CFG g_r and the substitution θ.

1.4.4 Correctness condition

We define the set of behaviors of a program as follows:

$$Beh = \{\mathsf{term}(t) \mid t \in Trace\} \cup \{\mathsf{forever}(t) \mid t \in Trace\}$$

where t represents a potentially infinite trace of observable events, and $\mathsf{term}(t)$ and $\mathsf{forever}(t)$ respectively denote executions terminating or diverging with a trace t. We use $\pi \Downarrow b$ to indicate that π has behavior b, as defined below.

Definition 2. *The relation $\pi \Downarrow b$ is defined as follows:*

- if $\eta_i(\pi) \xrightarrow{t}{}^{*} \eta_f$ and $\eta_f \in \mathit{Final}$ then $\pi \Downarrow \mathsf{term}(t)$

- if $\eta_i(\pi) \xrightarrow{t}{}^{\infty}$ then $\pi \Downarrow \mathsf{forever}(t)$

where: $\eta_i(\pi)$ is the initial state of program π; $\xrightarrow{t}{}^{*}$ is the reflexive transitive closure of $\xrightarrow{t}$; Final is the set of final program states (indicating program termination); and $\eta \xrightarrow{t}{}^{\infty}$ indicates that execution runs forever producing trace t under $\rightarrow$ when started at η.

To show the correctness of our execution engine, we prove the following theorem in Coq:

Theorem 1. *If* $\mathsf{PEC}(r) \neq \mathsf{Fail}$ *and* $\pi \Downarrow b$ *then* $\mathsf{TrProg}(\pi, r) \Downarrow b$.

In the following, we describe a Coq proof of Theorem 1. To do this, we fix a particular rule r and assume $\mathsf{PEC}(r) = (\Psi, \rho)$, where Ψ is the parametrized simulation relation found by PEC for r and ρ is a proof of $\mathsf{IsSimRel}(r, \Psi)$ (which essentially guarantees that $\mathsf{IsSimRel}(r, \Psi)$ holds).

Step left and step right To simplify applying the proof ρ of $\mathsf{IsSimRel}(r, \Psi)$, we construct two new, closely related semantics that are specialized to a concrete simulation relation:

Definition 3. *We define* $\eta_\ell \xrightarrow{t}_\ell \eta'_\ell$ *as the smallest relation satisfying:*

$$\left[\begin{array}{rcl} \mathsf{TrCFG}(\mathsf{g}(\eta_\ell), r) = & (\mathsf{g}(\eta_r), \theta) \\ \psi = & \theta(\Psi) \end{array}\right] \Longrightarrow$$

$$\left[\begin{array}{lcl} \mathsf{p}(\eta_\ell) \notin \theta \wedge \mathsf{p}(\eta'_\ell) \notin \theta \wedge \eta_\ell \xrightarrow{t} \eta'_\ell & \Rightarrow & \eta_\ell \xrightarrow{t}_\ell \eta'_\ell \\ \mathsf{p}(\eta_\ell) \notin \theta \wedge \mathsf{p}(\eta'_\ell) \in \psi \wedge \eta_\ell \xrightarrow{t} \eta'_\ell & \Rightarrow & \eta_\ell \xrightarrow{t}_\ell \eta'_\ell \\ \mathsf{p}(\eta_\ell) \in \psi \wedge \mathsf{p}(\eta'_\ell) \in \psi \wedge \eta_\ell \xrightarrow{t}_\psi \eta'_\ell & \Rightarrow & \eta_\ell \xrightarrow{t}_\ell \eta'_\ell \\ \mathsf{p}(\eta_\ell) \in \psi \wedge \mathsf{p}(\eta'_\ell) \notin \theta \wedge \eta_\ell \xrightarrow{t} \eta'_\ell & \Rightarrow & \eta_\ell \xrightarrow{t}_\ell \eta'_\ell \end{array}\right]$$

Note that formulas in square brackets are implicit conjunctions of formulas, one formula per line. The relation $\eta_r \xrightarrow{t}_r \eta'_r$ is defined analogously to $\xrightarrow{t}_\ell$ by substituting η_r for η_ℓ in the right-hand side of the main implication above.

The notation $\mathsf{p}(\eta_\ell) \notin \theta$ indicates that the program point of state η is not in a region of CFG transformed by TrCFG. This is implemented by searching θ to determine if a parameterized program point maps to $\mathsf{p}(\eta_\ell)$ such that the parameterized program point is not one of the *exit* points from the transformed code back to unmodified code. For briefness, we may speak of a state η not being in the transformed region; this simply means $\mathsf{p}(\eta) \notin \theta$.

Intuitively, $\xrightarrow{t}_\ell$ captures distinct ways in which the original code can step from state η_ℓ to η'_ℓ. In Definition 3, the first line of the main implication's right-hand side handles situations where neither η_ℓ nor η'_ℓ are in the transformed region. In this case $\eta_\ell \xrightarrow{t}_\ell \eta'_\ell$ holds whenever $\eta_\ell \xrightarrow{t} \eta'_\ell$ holds, that is whenever η_ℓ could take a normal `RTL` step to η'_ℓ. The second and fourth lines capture entering and exiting the transformed region, which again requires $\eta_\ell \xrightarrow{t} \eta'_\ell$. Note that we only allow entering and exiting transformed code through program locations that are in ψ. The third line captures the

situation where the original program executes from entry to entry of ψ using $\rightarrow_\psi$.

Similar to the definition of $\Downarrow$ (Definition 2), we also define $\Downarrow_\ell$ and $\Downarrow_r$, which respectively use $\rightarrow_\ell$ and $\rightarrow_r$ rather than $\rightarrow$.

1.4.5 Proof architecture

To establish Theorem 1 for program π and rewrite rule $r = (G_\ell, G_r, S)$ where $\mathsf{PEC}(r) \neq \mathsf{Fail}$, our Coq proof shows following three lemmas, which we describe in more detail below:

Lemma 1. *If $\pi \Downarrow b$ then $\pi \Downarrow_\ell b$.*

Lemma 2. *If $\pi \Downarrow_\ell b$ then $\mathsf{TrProg}(\pi, r) \Downarrow_r b$.*

Lemma 3. *If $\pi \Downarrow_r b$ then $\pi \Downarrow b$.*

Lemma 1 CompCert's library for small-step semantics allows us to demonstrate Lemma 1 if we can exhibit a simulation relation $\sim_1$ and a well-founded order $<$ on program states such that:

$$\eta \sim_1 \eta_\ell \wedge \eta \xrightarrow{t} \eta' \Rightarrow \exists \eta'_\ell, \eta' \sim_1 \eta'_\ell \wedge (\eta_\ell \xrightarrow{t}_\ell \eta'_\ell \vee \eta' < \eta) \tag{1.2}$$

Intuitively, this is the same as the standard simulation property (1.1), except that we allow for the possibility that η_ℓ does not step as long as the order is decreasing from η to η'.

We define $\eta \sim_1 \eta_\ell$ to hold when either: (a) $\eta = \eta_\ell$ and either η and η_ℓ are outside transformed code or both are at an entry in ψ or (b) η is in a transformed region, but not at an entry in ψ, η_ℓ is at an entry in ψ, and $\eta \xrightarrow{t}_\psi \eta_\ell$. Furthermore, we define the $<$ order as follows: $\eta' < \eta$ iff $m(\eta') < m(\eta)$ where $m(\eta)$ and $m(\eta')$ are the number of steps that η and η' have, respectively, until reaching the next entry in ψ.

We now have to show condition (1.2). The first and simpler case corresponds to (a) in the definition of $\eta \sim_1 \eta_\ell$. Here we show that the executions are in lockstep and that the successor states η' and η'_l are equal. The second and more difficult case, corresponding to (b) in the definition of $\eta \sim_1 \eta_\ell$, involves accounting for the steps of π's execution *between* entries in ψ. In this case: η_ℓ is at an entry in ψ (because we are in case (b) of the definition of $\eta \sim_1 \eta_\ell$) and it does not step; η is not at an entry in ψ and steps to η'; and from the definition of $\sim_1$ (the second case) we know $\eta \xrightarrow{t}_\psi \eta_\ell$. Thus η' is closer than η to the next entry in ψ (namely the program point of η_ℓ), which allows us to show that $\eta' < \eta$.

Lemma 2 Lemma 2 is the most difficult aspect of our Coq proof. CompCert's library for small-step semantics provides a theorem which allows us to demonstrate Lemma 2 if we can exhibit a simulation relation $\sim_2$ between the states of π and $\mathsf{TrProg}(\pi, r)$ that satisfies the following (which is essentially property (1.1)):

$$\eta_\ell \sim_2 \eta_r \wedge \eta_\ell \xrightarrow{t}_\ell \eta'_\ell \Rightarrow \exists \eta'_r, \eta'_\ell \sim_2 \eta'_r \wedge \eta_r \xrightarrow{t}_r \eta'_r \tag{1.3}$$

Definition 4. *We define $\eta_\ell \sim_2 \eta_r$ as the smallest relation satisfying:*

$$\left[\begin{array}{rl} \mathsf{TrCFG}(\mathsf{g}(\eta_\ell), r) = & (\mathsf{g}(\eta_r), \theta) \\ \psi = & \theta(\Psi) \end{array}\right] \Longrightarrow$$

$$\left[\begin{array}{rl} \mathsf{p}(\eta_\ell) \notin \theta \wedge \mathsf{p}(\eta_\ell) = \mathsf{p}(\eta_r) \wedge \mathsf{s}(\eta_\ell) = \mathsf{s}(\eta_r) \Rightarrow & \eta_\ell \sim_2 \eta_r \\ \psi(\eta_\ell, \eta_r) \Rightarrow & \eta_\ell \sim_2 \eta_r \end{array}\right]$$

Intuitively $\sim_2$ relates states using heap equality when the program points are outside of a transformed region, and using the simulation relation returned by PEC when

the program points are inside of a transformed region.

Proving condition (1.3) has four main cases, which correspond to the four conjuncts in the definitions of $\rightarrow_\ell$ and $\rightarrow_r$.

Case 1: η_ℓ and η_r are both outside of transformed regions and so are their successor states. This case is straightforward. Because $\eta_\ell \sim_2 \eta_r$ we know their heaps and program points are equal and because they are outside of transformed code, we know they are executing the same instruction. Thus η_r will step to η_r' where $\mathsf{p}(\eta_r') = \mathsf{p}(\eta_\ell')$ and $\mathsf{s}(\eta_r') = \mathsf{s}(\eta_\ell')$, which implies $\eta_\ell' \sim_2 \eta_r'$ (using the first case of $\sim_2$).

Case 2: η_ℓ and η_r are both stepping from outside the transformed region into the transformed region. Because both states start outside the transformed region, we know their heaps are equal and that they're executing the same instruction. Thus η_r will step to η_r' such that $\mathsf{s}(\eta_\ell') = \mathsf{s}(\eta_r')$. Furthermore, because PEC guarantees that the entries of matched code will be related in ψ with heap equality (see the part of definition 1 that uses HeapEq), $\mathsf{s}(\eta_\ell') = \mathsf{s}(\eta_r')$ implies $\psi(\eta_\ell', \eta_r')$. Thus $\eta_\ell' \sim_2 \eta_r'$ (using the second case of $\sim_2$).

Case 3: η_ℓ and η_r are both stepping from one entry of ψ to the next. We use the fact that $\mathsf{TrCFG}(\mathsf{g}(\eta_\ell), r) = (\mathsf{g}(\eta_r), \theta)$ to invoke the guarantee provided by PEC's proof of $\mathsf{IsSimRel}(r, \Psi)$. Specifically, $\mathsf{TrCFG}(\mathsf{g}(\eta_\ell), r) = (\mathsf{g}(\eta_r), \theta)$ implies that $S(\theta(G_\ell), \theta(G_r))$ which ensures $\mathsf{IsConSimRel}(\theta(\Psi), \theta(G_\ell), \theta(G_r))$ (see Definition 1 and TrCFG in Figure 1.8) . This fact ensures that the η_r will execute to η_r' and $\psi(\eta_\ell', \eta_r')$. Thus $\eta_\ell' \sim_2 \eta_r'$ (using the second case of $\sim_2$).

Case 4: η_ℓ and η_r are both stepping from inside the transformed region to outside the transformed region. Similar to Case 2 above, PEC guarantees that exits of matched code will be related in ψ with heap equality (see the part of definition 1 that uses HeapEq), meaning that $\psi(\eta_\ell, \eta_r)$ at the exit implies $\mathsf{s}(\eta_\ell) = \mathsf{s}(\eta_r)$. Furthermore, the way our pattern matching works ensures that $\mathsf{p}(\eta_\ell) = \mathsf{p}(\eta_r)$ and that the instruction at these

program points are equal. Thus η_r will step to η'_r where $\mathsf{p}(\eta'_r) = \mathsf{p}(\eta'_\ell)$ and $\mathsf{s}(\eta'_r) = \mathsf{s}(\eta'_\ell)$. From this it follows that $\eta'_\ell \sim_2 \eta'_r$ (using first case of $\sim_2$).

Lemma 3 CompCert's library for small-step semantics provides a theorem which allows us to demonstrate Lemma 3 if we can show:

$$\eta_r \xrightarrow{t}_r \eta'_r \Rightarrow \eta_r \xrightarrow{t}^+ \eta'_r$$

The above follows immediately from the definition of $\rightarrow_r$.

1.5 Coping with challenges

Throughout Sections 1.3 and 1.4, we have already shown how three techniques are very useful in managing the complexity of extending CompCert to support PEC rewrite rules: *Verified Validation*, *Semantics Alignment* and *Witness Propagation*. In this section we present several additional important challenges that we faced in our development and their solutions.

1.5.1 Termination of Coq code

Functions expressed in Coq's Calculus of Inductive Constructions must be shown to terminate. In most cases, Coq can prove termination automatically by finding an appropriate measure on a function's arguments that decreases with recursive calls. However, analyses that attempt to reach a fixed point or traverse cyclic structures like CFGs often pose problems for Coq's automated termination-proving strategy. One solution to this problem is to develop a termination proof for such functions in Coq. In general this can be hard, and it also makes the functions more difficult to update, since the termination proof also needs updating.

Another solution to is the introduction of a timeout parameter that is decremented for each recursive call. If it ever reaches zero the function immediately returns with a special $\perp$ value. Using this approach, Coq can now show termination automatically. The downside of this simplistic approach is that the algorithm is now incomplete, since in some cases it can return $\perp$, and the proof of correctness needs to take this into account. However, this is not a problem in domains where there is a safe fallback return value that makes the proof go through. This is indeed the case in the compiler domain: the safe return value is the one that leads to no transformations – for example a pattern matcher can always return Fail. Although a constant timeout may appear to be crude solution at first, we have found that it presents a very good engineering trade-off, since a large timeout often suffices in practice.

1.5.2 Case explosion

Conceptually, our intermediate semantics $\rightarrow_\ell$ and $\rightarrow_r$ have only four cases, as show in Section 1.4. However, such definitions on paper often lead to formal Coq definitions with many cases. For example, expressing $\rightarrow_\ell$ and $\rightarrow_r$ in terms of CompCert's small-step $\rightarrow$ leads to a total of 9 cases. Most of these 9 cases use $\rightarrow$ which itself has 12 cases, leading to an explosion in the number of cases. In the end, however, only a handful of these case are actually feasible at any one point in the proof, and a paper-and-pencil proof could easily say "the only feasible cases are ...". However, the formal proof needs to handle every case, leading to complex accounting.

One approach that we have found very helpful with eliminating the many infeasible cases is to thread additional information in the return values of functions. This additional information is not used by the computation itself, but rather in the proof, to provide the right context in the callers to know how to prune appropriate cases. One example of this approach is the PEC execution engine from Figure 1.8, which threads

the substitution found in TrPoint all the way back up to TrProg, even though for the purposes of applying PEC rules, this substitution is not needed outside of TrPoint. In other cases, we have also found that implementing specialized tactics in Coq's tactic languages allows us to easily handle many similar cases using few lines of proof.

1.5.3 Law of the excluded middle

The law of excluded middle occurs very naturally when working out high level proof sketches. Unfortunately, the constructive logic underlying Coq does not provide this luxury. As an example, one could be tempted in a proof sketch to split on termination: either execution returns from a given function call or it does not. However, this intuitive fact cannot be shown in Coq, because it would require deciding algorithmically if the function terminates. Instead one must create an inductive construct with two constructors corresponding to the intuitive case split. This is precisely how termination vs. non-termination is handled in CompCert, as shown in the definition of $\Downarrow$ (Definition 2). Alternatively, in situations where it is possible, one can implement a decision procedure that correctly distinguishes between the various cases of interest. Then, within a proof, one can perform case analysis on the result produced by this decision procedure.

1.6 Evaluation

XCert extends the CompCert verified compiler with an execution engine that applies parameterized rewrite rules checked by PEC. Below we characterize our implementation of XCert by comparing it to both an untrusted prototype execution engine and to some of the manual optimizations found within CompCert (Sections 1.6.1 and 1.6.2). Next, we evaluate XCert in terms of its trusted computing base (Section 1.6.3), extensibility (Section 1.6.4) and correctness guarantee (Section 1.6.5). We conclude by considering the limitations of our current execution engine (Section 1.6.6).

1.6.1 Engine Complexity

The PEC execution engine that we added to CompCert comprises approximately 1,000 lines of Coq code. Its main components are the pattern matching and the substitution application which allow us to easily implement the transformations specified by PEC rewrite rules.

The PEC untrusted prototype execution engine mentioned in [44] was roughly 400 lines of OCaml code. Although both execution engines apply PEC rewrite rules to perform optimizations, they work in very different settings. The CompCert execution engine targets the CFG-based `RTL` representation in CompCert, while the prototype in [44] targets an AST-based representation of a C-like IR.

We also compare the PEC execution engine against CompCert's two main `RTL` optimizations, common subexpression elimination (CSE) and constant propagation (CP). CSE is 440 lines of Coq code, and CP is 1,000 lines. Both of these optimizations make use of a general purpose dataflow solver, which is about 1,200 lines. Structurally, the PEC execution engine is very different from the optimizations in CompCert. Most of the code in the PEC engine performs pattern matching and tricky CFG splicing to achieve the task of replacing an entire region of the CFG with another. Instead, CSE and CP in CompCert perform simple CFG rewrites (one statement to another), and instead focus their efforts on computing dataflow information.

1.6.2 Proof Complexity

The proof of correctness for our execution engine is approximately 3,000 lines of Coq proof code. This code defines (1) the intermediate semantics $\rightarrow_\ell$ and $\rightarrow_r$ that facilitate applying the PEC guarantee, (2) Coq proof scripts demonstrating the semantic preservation of transformations performed by the execution engine and (3) tactics that make developing these proofs easier and more concise.

CompCert's correctness proofs for CSE and CP each span nearly 1,000 lines of proof code. Structurally, the correctness proofs for these CompCert optimizations are quite different from the execution engine's correctness proof, because they deal with different challenges. The CSE and CP proofs are mainly devoted to extracting useful facts from the result of the dataflow analysis performed by the transformation. These facts are then used to establish sufficient conditions for semantic preservation. In contrast, the proof of the execution engine focuses on showing that the many-to-many CFG rewrites that the PEC engine performs are correct. This typically involves splitting into two cases: cases where execution is not in the transformed code, which are typically straightforward; and cases where execution is in some region that has been transformed, in which case the proof effort involves either showing the case cannot arise or the simulation relation from PEC applies.

Note that the correctness proof for the PEC execution engine is three times larger than the PEC execution engine itself. However, the engineering effort for developing the proof was at least an order of magnitude greater than the effort for developing the execution engine. This is because we re-engineered the proof several times to make it simpler, cleaner, and more manageable using tactics.

1.6.3 Trusted Computing Base

The trusted computing base (TCB) consists of those components that are trusted to be correct. A bug in these components could invalidate any of the correctness guarantees that are being provided. The TCB for the regular CompCert compiler (without the PEC engine) includes CompCert's implementation of the C semantics, Coq's underlying theory (the Calculus of Inductive Constructions), and Coq's internal proof checker.

When CompCert is extended with the PEC execution engine, the TCB grows because, even though the engine is proved correct in Coq, we trust that the PEC checker

correctly checks any simulation relation it returns. Within the PEC implementation this checker is implemented in about 100 lines of OCaml code and makes calls to an SMT solver like Simplify [25] or Z3 [24]. Thus, the PEC engine adds the following to CompCert's TCB: 100 lines of OCaml for the PEC checker, an SMT solver like Simplify or Z3, and the encoding of CompCert's `RTL` semantics to be used by the SMT solver.

1.6.4 Extensibility

With this relatively small increase in TCB comes the following benefit: additional optimizations that are added using PEC *do not require any new manual proof effort, and do not add anything to the TCB*. In contrast, for each new optimization added to CompCert, unless a verified validator has already been specifically designed for it, the new optimization would either have to be proved correct, or if not, it would be trusted, thus increasing the TCB. Thus, the provably correct PEC execution engine brings all of the expressiveness and extensibility shown previously in [44] to CompCert while adding only a small amount to the TCB.

To test the extensibility of our system, we implemented and ran all the optimizations checked by PEC's "Relate" module in [44]. We ran the optimizations on an array of CompCert C benchmarks totaling about 10,000 lines of code. The benchmarks included cryptographic code like AES and SHA1, numeric computations such as FFT and Mandelbrot, and a raytracer. We manually checked that the transformations were carried out as expected.

1.6.5 Correctness Guarantee

While the size of the TCB tells us how much needs to be trusted, it is also important to evaluate the correctness guarantee provided in exchange for this trust. Essentially, the CompCert compiler extended with our PEC execution engine provides

the same guarantee as the original CompCert compiler: *if* the compiler produces an output program, then the output program will be semantically equivalent to the corresponding input program.

There are two ways in which this guarantee is not as strong as one may hope for. First, CompCert extended with our PEC execution engine is not guaranteed to produce an output program, even on a valid input program, because some passes from CompCert may abort compilation. For example, during the stack layout phase of CompCert, if a program spills too many variables and exceeds the available stack for a given function, then CompCert is forced to abort without producing an assembly output program. However, the PEC engine itself always produces an output program, and therefore is not a source of incompleteness.

The other weakness in the PEC engine's correctness guarantee is shared by all systems that use verified validation. In particular, those parts of the system that are checked using verified validation may still contain bugs in them. For example, the initial version of our PEC execution engine did not always correctly instantiate fresh nodes for the RHS of a PCFG. However, when this bug was exercised, our verified validator detected that the generated nodes did not have the required freshness property, and prevented the incorrect transformation from being performed. Such bugs therefore manifest themselves not as violations of the input/output equivalence guarantee, but as missed optimization opportunities. The existence of such quality-of-optimization bugs emphasizes the value of having run our PEC engine on real code, as described in Section 1.6.4, and ensuring that the optimizations operate as expected.

1.6.6 Limitations

The PEC checker is currently not implemented in Coq. Thus, for each PEC rewrite rule r, we must translate by hand the simulation relation produced by the PEC

checker for r into a Coq term and axiomatize its correctness proof. We intend to develop a version of PEC that directly outputs these simulation relations as Coq terms. Eventually, we plan to also implement all of PEC in Coq and thus eliminate the disconnect between the two systems.

Our current version of parameterized statements like **S** in Figure 1.2 are only able to match fixed length sequences of arbitrary instructions. Although this allows us to simulate parameterized statements of a fixed size, we must properly implement parameterized statements to achieve the full expressiveness of PEC.

1.7 Future Work

There are several directions for future work that we intend to explore. First, we plan to systematically and thoroughly compare the quality of existing CompCert optimizations with their corresponding PEC versions. Our evaluation will consider the runtime performance of generated code and the number of missed optimization opportunities. This comparison will enable us to fine tune our PEC optimizations and execution engine which, eventually, we hope will match the optimization capabilities currently found in CompCert. More broadly, we will also evaluate the relative effort of adding optimizations using XCert versus coding them directly in Coq or within other optimization frameworks.

We also plan to explore further reductions to the TCB. When our PEC execution engine is added to CompCert, the TCB grows because the PEC checker becomes trusted. However, if we reimplement the PEC checker in Coq and formally prove its correctness, then our PEC engine would not at all increase the size of the TCB. The core of the PEC checker consists of only 100 lines of stateless OCaml code, which we anticipate will be easy to implement and reason about in Coq. However, this core checker makes queries to an SMT solver (like Z3) which could be challenging to integrate into Coq. Fortunately,

there are several reasons to be optimistic. First, some SMT solvers have recently been re-engineered to produce proof terms, which we should be able to automatically translate to Coq terms and thus integrate into a Coq proof (possibly using the Coq Classical extension to accommodate for the refutation based proof strategies common in SMT solvers). Second, the PEC checker's SMT queries tend to be simple and highly stylized. Thus, it may instead be possible to implement a sophisticated tactic in Coq's tactic language to discharge these obligations directly. We plan to investigate both of these approaches, with the ultimate goal of implementing a verified PEC checker in Coq.

Finally, we would also like to investigate extending XCert to support the "Permute" module from PEC [44]. This would allow additional loop optimizations to be easily added to CompCert, such as loop reversal and loop distribution. Adding such support to XCert would require formally developing the general theory of loop reordering transformations found in [92], upon which the PEC checker's "Permute" module is based. Doing this will be challenging because it's not clear how to express the above theory of loop transformations in a way that meshes well with CompCert's existing support for correctness proofs using simulation relations. Nonetheless, formalizing such a theory in Coq is worthwhile, as it would not only enable support for "Permute" optimizations in XCert, but could also be broadly useful within CompCert.

1.8 Acknowledgements

We thank Xavier Leroy, Jean-Baptiste Tristan, and the rest of the CompCert team for developing and releasing a well-documented and well-engineered tool. We also thank the anonymous reviewers for their careful reading and helpful comments. Finally, we thank the UCSD Programming Systems group for many useful conversations and insightful feedback during the development of XCert.

This chapter in full, is adapted from material as it appears in Programming

Language Design and Implementation 2010. Tatlock, Zachary; Lerner, Sorin, ACM Press, 2010. The dissertation author was the primary investigator and author of this paper.

Chapter 2

Formal Shim Verification

Web browsers mediate access to valuable private data in domains ranging from health care to banking. Despite this critical role, attackers routinely exploit browser vulnerabilities to exfiltrate private data and take over the underlying system. We present QUARK, a browser whose kernel has been implemented and verified in Coq. We give a specification of our kernel, show that the implementation satisfies the specification, and finally show that the specification implies several security properties, including tab non-interference, cookie integrity and confidentiality, and address bar integrity.

The QUARK project was a joint project with another PhD student, Dongseok Jang. The disseration author's primary contribution in QUARK was the architecture of QUARK's kernel, as well as the implementation and formalization of the kernel in Coq. Dongseok Jang's primary contribution was the design and implementation of all the non-kernel components that interact with the kernel.

2.1 Introduction

Web browsers increasingly dominate computer use as people turn to Web applications for everything from business productivity suites and educational software to social networking and personal banking. Consequently, browsers mediate access to highly valuable, private data. Given the browser's sensitive, essential role, it should be highly

secure and robust in the face of adversarial attack.

Unfortunately, security experts consistently discover vulnerabilities in all popular browsers, leading to data loss and remote exploitation. In the annual Pwn2Own competition, part of the CanSecWest security conference [4], security experts demonstrate new attacks on *up-to-date* browsers, allowing them to subvert a user's machine through the click of a single link. These vulnerabilities represent realistic, zero-day exploits and thus are quickly patched by browser vendors. Exploits are also regularly found in the wild; Google maintains a Vulnerability Reward Program, publishing its most notorious bugs and rewarding the cash to their reporters [2].

Researchers have responded to the problems of browser security with a diverse range of techniques, from novel browser architectures [10, 84, 27, 75, 57] and defenses against specific attacks [40, 32, 34, 8, 67] to alternative security policies [39, 74, 33, 8, 73, 5] and improved JavaScript safety [21, 35, 72, 6, 90]. While all these techniques improve browser security, the intricate subtleties of Web security make it very difficult to know with full certainty whether a given technique works as intended. Often, a solution only "works" until an attacker finds a bug in the technique or its implementation. Even in work that attempts to provide strong guarantees (for example [27, 16, 75, 15]) the guarantees come from analyzing a model of the browser, not the actual implementation. Reasoning about such a simplified model eases the verification burden by omitting the gritty details and corner cases present in real systems. Unfortunately, attackers exploit precisely such corner cases. Thus, these approaches still leave a *formality gap* between the theory and implementation of a technique.

There is one promising technique that could minimize this formality gap: *fully formal verification of the browser implementation*, carried out in the demanding and foundational context of a mechanical proof assistant. This severe discipline forces the programmer to specify precisely how their code should behave and then provides the

tools to formally guarantee that it does, all in fully formal logic, building from basic axioms up. For their trouble, the programmer is rewarded with a *machine checkable proof* that the implementation satisfies the specification. With this proof in hand, we can avoid further reasoning about the large, complex implementation, and instead consider only the substantially smaller, simpler specification. In order to believe that such a browser truly satisfies its specification, one needs only trust a very small, extensively tested proof checker. By reasoning about the actual implementation directly, we can guarantee that any security properties implied by the specification will hold in every case, on every run of the actual browser.

Unfortunately, formal verification in a proof assistant is tremendously difficult. Often, those systems which we can formally verify are severely restricted, "toy" versions of the programs we actually have in mind. Thus, many researchers still consider full formal verification of realistic, browser-scale systems an unrealistic fantasy. Fortunately, recent advances in fully formal verification allow us to begin challenging this pessimistic outlook.

In this chapter we demonstrate how *formal shim verification* radically reduces the verification burden for large systems to the degree that we were able to formally verify the implementation of a modern Web browser, QUARK, within the demanding and foundational context of the mechanical proof assistant Coq.

At its core, formal shim verification addresses the challenge of formally verifying a large system by cleverly reducing the amount of code that must be considered; instead of formalizing and reasoning about gigantic system components, all components communicate through a small, lightweight shim which ensures the components are restricted to only exhibit allowed behaviors. Formal shim verification only requires one to reason about the shim, thus eliminating the tremendously expensive or infeasible task of verifying large, complex components in a proof assistant.

Our Web browser, QUARK, exploits formal shim verification and enables us to verify security properties for a *million* lines of code while reasoning about only a *few hundred*. To achieve this goal, QUARK is structured similarly to Google Chrome [10] or OP [27]. It consists of a small browser kernel which mediates access to system resources for all other browser components. These other components run in sandboxes which only allow the component to communicate with the kernel. In this way, QUARK is able to make strong guarantees about a million lines of code (*e.g.,* the renderer, JavaScript implementation, JPEG decoders, etc.) while only using a proof assistant to reason about a few hundred lines of code (the kernel). Because the underlying system is protected from QUARK's untrusted components (*i.e.,* everything other than the kernel) we were free to adopt state-of-the-art implementations and thus QUARK is able to run popular, complex Web sites like Facebook and GMail.

By applying formal shim verification to only reason about a small core of the browser, we formally establish the following security properties in QUARK, all within a proof assistant:

1. **Tab Non-Interference**: no tab can ever affect how the kernel interacts with another tab

2. **Cookie Confidentiality and Integrity**: cookies for a domain can only be accessed/modified by tabs of that domain

3. **Address Bar Integrity and Correctness**: the address bar cannot be modified by a tab without the user being involved, and always displays the correct address bar.

To summarize, our contributions are as follows:

- We demonstrate how formal shim verification enabled us to formally verify the implementation of a modern Web browser. We discuss the techniques, tools, and

design decisions required to formally verify QUARK in detail.

- We identify and formally prove key security properties for a realistic Web browser.
- We provide a framework that can be used to further investigate and prove more complex policies within a working, formally verified browser.

The rest of the chapter is organized as follows. Section 4.2 provides background on browser security techniques and formal verification. Section 2.2 presents an overview of the QUARK browser. Section 2.3 details the design of the QUARK kernel and its implementation. Section 2.4 explains the tools and techniques we used to formally verify the implementation of the QUARK kernel. Section 2.5 evaluates QUARK along several dimensions while Section 2.6 discusses lessons learned from our endeavor.

2.2 QUARK Architecture and Design

Figure 2.1 diagrams QUARK's architecture. Similar to Chrome [10] and OP [27], QUARK isolates complex and vulnerability-ridden components in sandboxes, forcing them to access all sensitive resources through a small, simple browser kernel. Our kernel, written in Coq, runs in its own process and mediates access to resources including the keyboard, disk, and network. Each tab runs a modified version of WebKit in its own process. WebKit is the open source browser engine used in Chrome and Safari. It provides various callbacks for clients as Python bindings which we use to implement tabs. Since tab processes cannot directly access any system resources, we hook into these callbacks to re-route WebKit's network, screen, and cookie access through our kernel written in Coq. QUARK also uses separate processes for displaying to the screen, storing and accessing cookies, as well reading input from the user.

Throughout the chapter, we assume that an attacker can compromise any QUARK component which is exposed to content from the Internet, except for the kernel which we

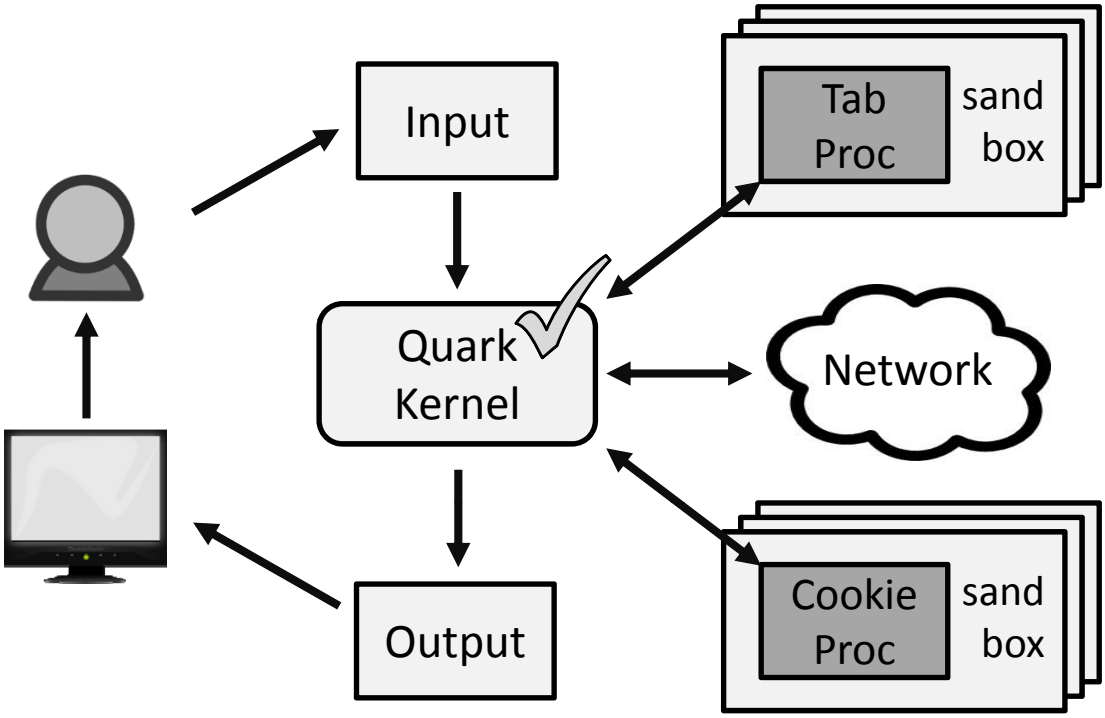


Figure 2.1. QUARK *Architecture.* How QUARK factors a modern browser into distinct components.

formally verified. This includes all tab processes, cookie processes, and the graphical output process. Thus, we provide strong formal guarantees about tab and cookie isolation, even when some processes have been completely taken over (*e.g.,* by a buffer overflow attack in the rendering or JavaScript engine of WebKit).

2.2.1 Graphical User Interface

The traditional GUI for Web browsers manages several key responsibilities: reading mouse and keyboard input, showing rendered graphical output, and displaying the current URL. Unfortunately, such a monolithic component cannot be made to satisfy our security goals. If compromised, such a GUI component could spoof the current URL or send arbitrary user inputs to the kernel, which, if coordinated with a compromised tab, would violate tab isolation. Thus QUARK must carefully separate GUI responsibilities to preserve our security guarantees while still providing a realistic browser.

QUARK divides GUI responsibilities into several components which the kernel orchestrates to provide a traditional GUI for the user. The most complex component displays rendered bitmaps on the screen. QUARK puts this component in a separate process to which the kernel directs rendered bitmaps from the currently selected tab. Because the kernel never reads input from this graphical output process, any vulnerabilities it may have cannot subvert the kernel or impact any other component in QUARK. Furthermore, treating the graphical output component as a separate process simplifies the kernel and proofs because it allows the kernel to employ a uniform mechanism for interacting with the outside world: messages over channels.

To formally reason about the address bar, we designed our kernel so that the current URL is written directly to the kernel's `stdout`. This gives rise to a hybrid graphical/text output as shown in Figure 2.2 where the kernel has complete control over the address bar. With this design, the graphical output process is never able to spoof the

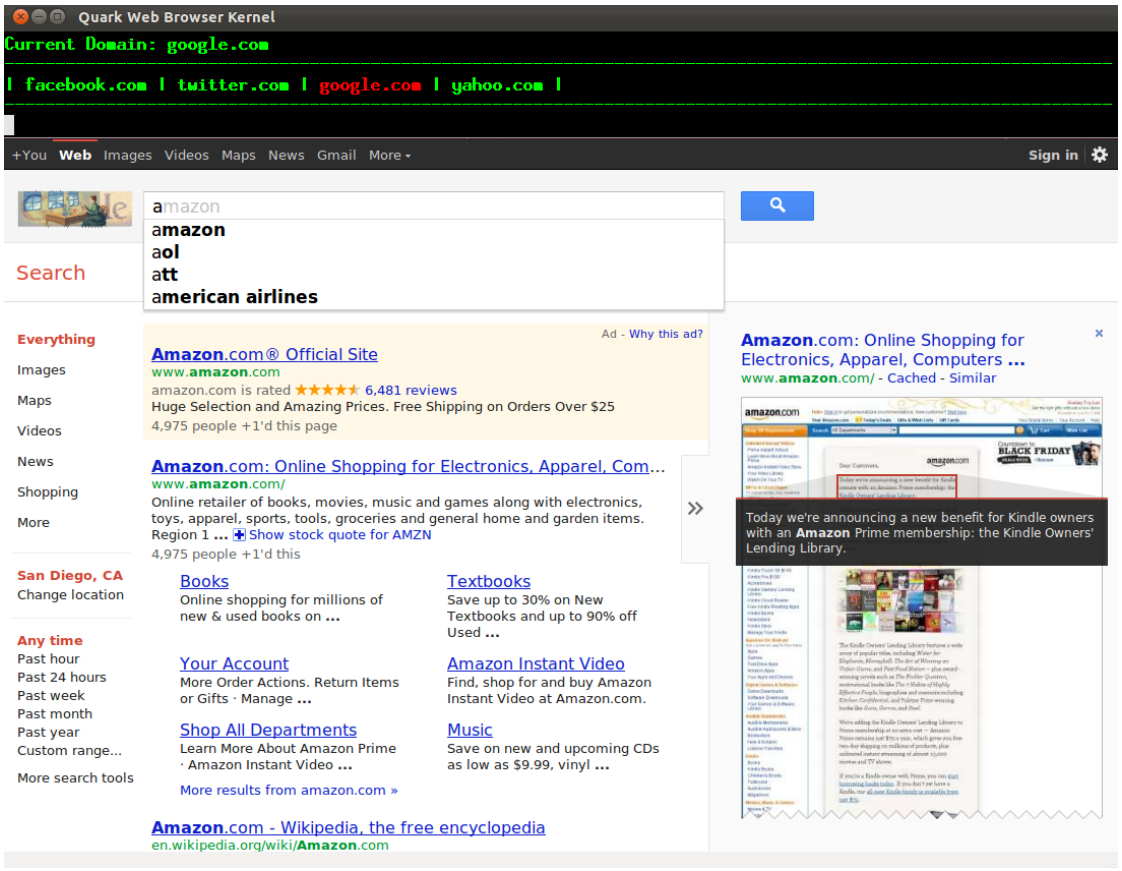


Figure 2.2. QUARK *Screenshot.* QUARK running a Google search.

address bar.

QUARK also uses a separate input process to support richer inputs, *e.g.,* the mouse. The input process is a simple Python script which grabs keyboard and mouse events from the user, encodes them as user input messages, and forwards them on to the kernel's `stdin`. For keystrokes, the input process simply writes characters in ASCII format to the kernel's `stdin`. We use several "un-printable" ASCII values (all smaller than 60 and all untypeable from the keyboard) to pass special information from the input process to the kernel. For example, the input process maps keys F1-F12 to such un-printable characters, which allows the kernel to use F11 for "new tab", and F1-F10 for selecting tabs 1-10. Mouse clicks are also sent to the kernel through un-printable ASCII values. Because the input process only reads from the keyboard and mouse, and never from the kernel or any other QUARK components, it cannot be exposed any attacks originating from the network.

2.2.2 Example of Message Exchanges

To illustrate how the kernel orchestrates all the components in QUARK, we detail the steps from startup to a tab loading http://www.google.com. The user opens QUARK by starting the kernel which in turn starts three processes: the input process, the graphical output process, and a tab process. The kernel establishes a two-way communication channel with each process it starts. Next, the kernel then sends a `(Go "http://www.google.com")` message to the tab indicating it should load the given URL (for now, assume this is normal behavior for all new tabs).

The tab process comprises our modified version of WebKit wrapped by a thin layer of Python to handle messaging with the kernel. After recieving the `Go` message, the Python wrapper tells WebKit to start processing http://www.google.com. Since the tab process is running in a sandbox, WebKit cannot directly access the network. When it

attempts to, our Python wrapper intervenes and sends a `GetURL` request to the kernel. As long as the request is valid, the kernel responds with a `ResDoc` message containing the HTML document the tab requested.

Once the tab process has received the necessary resources from the kernel and rendered the Web pages, it sends a `Display` message to the kernel which contains a bitmap to display. When the kernel receives a `Display` message from the current tab, it forwards the message on to the graphical output process, which in turn displays the bitmap on the screen.

When the kernel reads a printable character `c` from standard input, it sends a `(KeyPress c)` message to the currently selected tab. Upon receiving such a message, the tab calls the appropriate input handler in WebKit. For example, if a user types "a" on Google, the "a" character is read by the kernel, passed to the tab, and then passed to WebKit, at which point WebKit adds the "a" character to Google's search box. This in turn causes WebKit's JavaScript engine to run an event handler that Google has installed on their search box. The event handler performs an "instant search", which initiates further communication with the QUARK kernel to access additional network resources, followed by another `Display` message to repaint the screen. Note that to ease verification, QUARK currently handles all requests synchronously.

2.2.3 Efficiency

With a few simple optimizations, we achieve performance comparable to WebKit on average (see Section 2.5 for measurements). Following Chrome, we adopt two optimizations critical for good graphics performance. First, QUARK uses shared memory to pass bitmaps from the tab process through the kernel to the output process, so that the `Display` message only passes a shared memory ID instead of a bitmap. This drastically reduces the communication cost of sending bitmaps. To prevent a malicious tab from

accessing another tab's shared memory, we run each tab as a different user, and set access controls so that a tab's shared memory can only be accessed by the output process. Second, QUARK uses *rectangle-based* rendering: instead of sending a large bitmap of the entire screen each time the display changes, the tab process determines which part of the display has changed, and sends bitmaps only for the rectangular regions that need to be updated. This drastically reduces the size of the bitmaps being transferred, and the amount of redrawing on the screen.

For I/O performance, the original Ynot library used single-character read/write routines, imposing significant overhead. We defined a new I/O library which uses size `n` reads/writes. This reduced reading an `n` byte message from `n` I/O calls to just three: reading a 1 byte tag, followed by a 4 byte payload size, and then a single read for the entire payload.

We also optimized socket connections in QUARK. Our original prototype opened a new `TCP` connection for each HTTP `GET` request, imposing significant overhead. Modern Web servers and browsers use *persistent connections* to improve the efficiency of page loading and the responsiveness of Web 2.0 applications. These connections are maintained anywhere from a few seconds to several minutes, allowing the client and server can exchange multiple request/responses on a single connection. Services like Google Chat make use of very long-lived HTTP connections to support responsive interaction with the user.

We support such persistent HTTP connections via Unix domain sockets which allow processes to send open file descriptors over channels using the `sendmsg` and `recvmsg` system calls. When a tab needs to open a socket, it sends a `GetSoc` message to the kernel with the host and port. If the request is valid, the kernel opens and connects the socket, and then sends an *open* socket file descriptor to the tab. Once the tab gets the socket file descriptor, it can read/write on the socket, but it cannot re-connect the socket

to another host/port. In this way, the kernel controls all socket connections.

Even though we formally verify our browser kernel in a proof assistant, we were still able to implement and reason about these low-level optimizations.

2.2.4 Socket Security Policy

The `GetSoc` message brings up an interesting security issue. If the kernel satisfied all `GetSoc` requests, then a compromised tab could open sockets to any server and exchange arbitrary amounts of information. The kernel must prevent this scenario by restricting socket connections.

To implement this restriction, we introduce the idea of a *domain suffix* for a tab which the user enters when the tab starts. A tab's domain suffix controls several security features in QUARK, including which socket connections are allowed and how cookies are handled (see Section 2.2.5). In fact, our address bar, located at the very top of the browser (see Figure 2.2), displays the domain suffix, not just the tab's URL. We therefore refer to it as the "domain bar".

For simplicity, our current domain suffixes build on the notion of a *public suffix*, which is a top-level domain under which Internet users can directly register names, for example .com, .co.uk, or .edu – Mozilla maintains an exhaustive list of such suffixes [3]. In particular, we require the domain suffix for a tab to be exactly one level down from a public suffix, *e.g.,* google.com, amazon.com, etc. In the current QUARK prototype the user provides a tab's domain suffix separately from its initial URL, but one could easily compute the former from the later. Note that, once set, a tab's domain suffix never changes. In particular, any frames a tab loads do not affect its domain suffix.

We considered using the tab's origin (which includes the URL, scheme, and port) to restrict socket creation, but such a policy is too restrictive for many useful sites. For example, a single GMail tab uses frames from domains such as static.google.com and

mail.google.com. However, our actual domain suffix checks are modularized within QUARK, which will allow us to experiment with finer grained policies in future work.

To enforce our current socket creation policy, we first define a subdomain relation $\leq$ as follows: given domain d_1 and domain suffix d_2, we use $d_1 \leq d_2$ to denote that d_1 is a subdomain of d_2. For example www.google.com $\leq$ google.com. If a tab with domain suffix t requests to open a connection to a host h, then the kernel allows the connection if $h \leq t$. To load URLs that are not a subdomain of the tab suffix, the tab must send a `GetURL` message to the kernel – in response, the kernel does not open a socket but, if the request is valid, may provide the content of the URL. Since the kernel does not attach any cookies to the HTTP request for a `GetURL` message, a tab can only access publicly available data using `GetURL`. In addition, `GetURL` requests only provide the response body, not HTTP headers.

Note that an exploited tab could leak cookies by encoding information within the URL parameter of `GetURL` requests, but only cookies for that tab's domain could be leaked. Because we do not provide any access to HTTP headers with GetURL, we consider this use of GetURL to leak cookies analogous to leaking cookie data over timing channels.

Although we elide details in the current work, we also slightly enhanced our socket policy to improve performance. Sites with large data sets often use content distribution networks whose domains will not satisfy our domain suffix checks. For example facebook.com uses fbcdn.net to load much of its data. Unfortunately, the simple socket policy described above will force all this data to be loaded using slow `GetURL` requests through the kernel. To address this issue, we associate *whitelists* with the most popular sites so that tabs for those domains can open sockets to the associated content distribution network. The tab domain suffix remains a single string, e.g. facebook.com, but behind the scenes, it gets expanded into a list depending on the

domain, *e.g.,* [facebook.com, fbcdn.net]. When deciding whether to satisfy a given socket request, QUARK considers this list as a disjunction of allowed domain suffixes. Currently, we provide these whitelists manually.

2.2.5 Cookies and Cookie Policy

QUARK maintains a set of cookie processes to handle cookie accesses from tabs. This set of cookie processes will contain a cookie process for domain suffix S if S is the domain suffix of a running tab. By restricting messages to and from cookie processes, the QUARK kernel guarantees that browser components will only be able to access cookies appropriate for their domain.

The kernel receives cookie store/retrieve requests from tabs and directs the requests to the appropriate cookie process. If a tab with domain suffix t asks to store a cookie with domain c, then our kernel allows the operation if $c \leq t$, in which case it sends the store request to the cookie process for domain t. Similarly, if a tab with domain suffix t wants to retrieve a cookie for domain c, then our kernel allows the operation if $c \leq t$, in which case it sends the request to the cookie process for domain t and forwards any response to the requesting tab.

The above policy prevents cross-domain cookie reads from a compromised tab, and it prevents a compromised cookie process from leaking information about its cookies to another domain; yet it also allows different tabs with the same domain suffix (but different URLs) to communicate through cookies (for example, mail.google.com and calendar.google.com).

2.2.6 Security Properties of QUARK

We provide descriptions of the security properties we proved for QUARK's kernel; formal definitions appear later in Section 2.3. A tab in the kernel is a pair, containing the

tab's domain suffix as a string and the tab's communication channel as a file descriptor. A cookie process is also a pair, containing the domain suffix that this cookie process manages and its communication channel. We define the state of the kernel as the currently selected tab, the list of tabs, and the list of cookie processes. Note that the kernel state only contains strings and file descriptors.

We prove the following main theorems in Coq:

1. **Response Integrity**: The way the kernel responds to any request only depends on past user "control keys" (namely keys F1-F12). This ensures that one browser component (*e.g.,* a tab or cookie process) can never influence how the kernel responds to another component, and that the kernel never allows untrusted input (*e.g.,* data from the web) to influence how the kernel responds to a request.

2. **Tab Non-Interference**: The kernel's response to a tab's request is the same no matter how other tabs interact with the kernel. This ensures that the kernel never provides a direct way for one tab to attack another tab or steal private information from another tab.

3. **No Cross-domain Socket Creation**: The kernel disallows any cross-domain socket creation (as described in Section 2.2.4).

4. **Cookie Integrity/Confidentiality**: The kernel disallows any cross-domain cookie stores or retrieves (as described in Section 2.2.5).

5. **Domain Bar Integrity and Correctness**: The domain bar cannot be compromised by a tab, and is always equal to the domain suffix of the currently selected tab.

2.3 Kernel Implementation in Coq

QUARK's most distinguishing feature is its kernel, which is implemented and

proved correct in Coq. In this section we present the implementation of the main kernel loop. In the next section we explain how we formally verified the kernel.

Coq enables users to write programs in a small, simple functional language and then reason formally about them using a powerful logic, the Calculus of Constructions. This language is essentially an effect-free (pure) subset of popular functional languages like ML or Haskell with the additional restriction that programs must always terminate. Unfortunately, these limitations make Coq's default implementation language ill-suited for writing system programs like servers or browsers which must be effectful to perform I/O and by design may not terminate.

To address the limitations of Coq's implementation language, we use Ynot [61]. Ynot is a Coq library which provides monadic types that allow us to write effectful, non-terminating programs in Coq while retaining the strong guarantees and reasoning capabilities Coq normally provides. Equipped with Ynot, we can write our browser kernel in a fairly straightforward style whose essence is shown in Figure 2.3.

Single Step of Kernel. QUARK's kernel is essentially a loop that continuously responds to requests from the user or tabs. In each iteration, the kernel calls `kstep` which takes the current kernel state, handles a single request, and returns the new kernel state as shown in Figure 2.3. The kernel state is a tuple of the current tab (`ctab`), the list of tabs (`tabs`), and a few other components which we omit here (*e.g.,* the list of cookie processes). For details regarding the loop and kernel initialization code please see [37].

`kstep` starts by calling `iselect` (the "i" stands for input) which performs a Unix-style select over `stdin` and all tab input channels, returning `Stdin` if `stdin` is ready for reading or `Tab t` if the input channel of tab `t` is ready. `iselect` is implemented in Coq using a `select` primitive which is ultimately just a thin wrapper over the Unix select system call. The Coq extraction process, which converts Coq into OCaml for execution, can be customized to link our Coq code with OCaml implementations of primitives like

```
Definition kstep(ctab, ctabs) :=
  chan <- iselect(stdin, tabs);
  match chan with
  | Stdin =>
      c <- read(stdin);
      match c with
      | "+" =>
          t <- mktab();
          write_msg(t, Render);
          return (t, t::tabs)
      | ...
      end
  | Tab t =>
      msg <- read_msg(t);
      match msg with
      | GetSoc(host, port) =>
          if(safe_soc(host, domain_suffix(t)) then
            send_soc(t, host, port);
            return (ctab, tabs)
          else
            write_msg(t, Error);
            return (ctab, tabs)
      | ...
      end
  end
```

Figure 2.3. *Body for Main Kernel Loop.* How the QUARK kernel receives and responds to requests from other browser components.

`select`. Thus `select` is exposed to Coq essentially as a primitive of the appropriate monadic type. We have similar primitives for reading/writing on channels, and opening sockets.

Request from User. If `stdin` is ready for reading, the kernel reads one character `c` using the `read` primitive, and then responds based on the value of `c`. If `c` is "+", the kernel adds a new tab to the browser. To achieve this, it first calls `mktab` to start a tab process (another primitive implemented in OCaml). `mktab` returns a tab object, which contains an input and output channels to communicate with the tab process. Once the tab `t` is created, the kernel sends it a `Render` message using the `write_msg` function – this tells `t` to render itself, which will later cause the tab to send a `Display` message to the

kernel. Finally, we return an updated kernel state `(t, t::tabs)`, which sets the newly created tab `t` as the current tab, and adds `t` to the list of tabs.

In addition to "+" the kernel handles several other cases for user input, which we omit in Figure 2.3. For example, when the kernel reads keys F1 through F10, it switches to tabs 1 through 10, respectively, if the tab exists. To switch tabs, the kernel updates the currently selected tab and sends it a `Render` message. The kernel also processes mouse events delivered by the input process to the kernel's `stdin`. For now, we only handle mouse clicks, which are delivered by the input process using a single un-printable ASCII character (adding richer mouse events would not fundamentally change our kernel or proofs). The kernel in this case calls a primitive implemented in OCaml which gets the location of the mouse, and it sends a `MouseClick` message using the returned coordinates to the currently selected tab. We use this two-step approach for mouse clicks (un-printable character from the input process, followed by primitive in OCaml), so that the kernel only needs to processes a single character at a time from `stdin`, which simplifies the kernel and proofs.

Request from Tab. If a tab `t` is ready for reading, the kernel reads a message `m` from the tab using `read_msg`, and then sends a response which depends on the message. If the message is `GetSoc(host, port)`, then the tab is requesting that a socket be opened to the given host/port. We apply the socket policy described in Section 2.2.4, where `domain_suffix t` returns the domain suffix of a tab `t`, and `safe_soc(host, domsuf)` applies the policy (which basically checks that `host` is a sub-domain of `domsuf`). If the policy allows the socket to be opened, the kernel uses the `send_socket` to open a socket to the host, and send the socket over the channel to the tab (recall that we use Unix domain sockets to send open file descriptors from one process to another). Otherwise, it returns an `Error` message.

In addition to `GetSoc` the kernel handles several other cases for tab requests,

which we omit in Figure 2.3. For example, the kernel responds to `GetURL` by retrieving a URL and returning the result. It responds to cookie store and retrieve messages by checking the security policy from Section 2.2.5 and forwarding the message to the appropriate cookie process (note that for simplicity, we did not show the cookie processes in Figure 2.3). The kernel also responds to cookie processes that are sending cookie results back to a tab, by forwarding the cookie results to the appropriate tab. The kernel responds to `Display` messages by forwarding them to the output process.

Monads in Ynot. The code in Figure 2.3 shows how Ynot supports an imperative programming style in Coq. This is achieved via *monads* which allow one to encode effectful, non-terminating computations in pure languages like Haskell or Coq. Here we briefly show how monads enable this encoding. In the next section we extend our discussion to show how Ynot's monads also enable reasoning about the kernel using pre- and post-conditions as in Hoare logic.

We use Ynot's `ST` monad which is a parameterized type where `ST T` denotes computations which may perform some I/O and then return a value of type `T`. To use `ST`, Ynot provides a `bind` primitive which has the following dependent type:

```
bind : forall T1 T2,
  ST T1 -> (T1 -> ST T2) -> ST T2
```

This type indicates that, for any types `T1` and `T2`, `bind` will take two parameters: (1) a monad of type `ST T1` and (2) a function that takes a value of type `T1` and returns a monad of type `ST T2`; then `bind` will produce a value in the `ST T2` monad. The type parameters `T1` and `T2` are inferred automatically by Coq. Thus, the expression `bind X Y` returns a monad which represents the computation: run `X` to get a value `v`; run `(Y v)` to get a value `v'`; return `v'`.

To make using `bind` more convenient, Ynot also defines Haskell-style "do" syntactic sugar using Coq's Notation mechanism, so that `x <- a;b` is translated to

`bind a (fun x => b)`, and `a;b` is translated to `bind a (fun _ => b)`. Finally, the Ynot library provides a `return` primitive of type `forall T (v: T), ST T` (where again `T` is inferred by Coq). Given a value `v`, the monad `return v` represents the computation that does no I/O and simply returns `v`.

2.4 Kernel Verification

In this section we explain how we verified QUARK's kernel. First, we specify correct behavior of the kernel in terms of *traces*. Second, we prove the kernel satisfies this specification using the full power of Ynot's monads. Finally, we prove that our kernel specification implies our target security properties.

2.4.1 Actions and Traces

We verify our kernel by reasoning about the sequences of calls to primitives (*i.e.,* system calls) it can make. We call such a sequence a *trace*; our kernel specification (henceforth "spec") defines which traces are allowed for a correct implementation as in [54].

We use a list of *actions* to represent the trace the kernel produces by calling primitives. Each action in a trace corresponds to the kernel invoking a particular primitive. Figure 2.4 shows a partial definition of the `Action` datatype. For example: `ReadN f n l` is an `Action` indicating that the `n` bytes in list `l` were read from input channel `f`; `MkTab t` indicates that tab `t` was created; `SentSoc t host port` indicates a socket was connected to `host`/`port` and passed to tab `t`.

We can manipulate traces and Actions like any other values in Coq. For example, we can define a function `Read c b` to encode the special case that a single byte `b` was read on input channel `c`. Though not shown here, we also define similar helper functions to build up trace fragments which correspond to having read or written a particular message

```
Definition Trace := list Action.

Inductive Action :=
  | ReadN   : chan -> positive -> list ascii -> Action
  | WriteN  : chan -> positive -> list ascii -> Action
  | MkTab   : tab -> Action
  | SentSoc : tab -> list ascii -> list ascii -> Action
  | ...

Definition Read c b :=
  ReadN c 1 [c]
```

Figure 2.4. *Traces and Actions.* Coq code defining the type of externally visible actions.

to a given component. For example, `ReadMsg t (GetSoc host port)` corresponds to the trace fragment that results from reading a `GetSoc` request from tab `t`.

2.4.2 Kernel Specification

Figure 2.5 shows a simplified snippet of our kernel spec. The spec is a predicate `tcorrect` over traces with two constructors, stating the two ways in which `tcorrect` can be established: (1) `tcorrect_nil` states that the empty trace satisfies `tcorrect` (2) `tcorrect_step` states that if `tr` satisfies `tcorrect` and the kernel takes a single step, meaning that after `tr` it gets a request `req`, and responds with `rsp`, then the trace `rsp ++ req ++ tr` (where `++` is list concatenation) also satisfies `tcorrect`. By convention the first action in a trace is the most recent.

The predicate `step_correct` defines correctness for a single iteration of the kernel's main loop: `step_correct tr req rsp` holds if given the past trace `tr` and a request `req`, the response of the kernel should be `rsp`. The predicate has several constructors (not all shown) enumerating the ways `step_correct` can be established. For example, `step_correct_add_tab` states that typing "+" on `stdin` leads to the creation of a tab and sending the `Render` message. The `step_correct_socket_true` case captures the successful socket creation case, whereas `step_correct_socket_false` captures the error case.

```
Inductive tcorrect : Trace -> Prop :=
  | tcorrect_nil:
      tcorrect nil
  | tcorrect_step: forall tr req rsp,
      tcorrect tr ->
      step_correct tr req rsp ->
      tcorrect (rsp ++ req ++ tr).

Inductive step_correct :
  Trace -> Trace -> Trace -> Prop :=
  | step_correct_add_tab: forall tr t,
      step_correct tr
        (MkTab t :: Read stdin "+" :: nil)
        (WroteMsg t Render)
  | step_correct_socket_true: forall tr t host port,
      is_safe_soc host (domain_suffix t) = true ->
      step_correct tr
        (ReadMsg t (GetSoc host port))
        (SentSoc t host port)
  | step_correct_socket_false: forall tr t host port,
      is_safe_soc host (domain_suffix t) <> true ->
      step_correct tr
        (ReadMsg t (GetSoc host port) ++ tr)
        (WroteMsg t Error ++ tr)
  | ...
```

Figure 2.5. *Kernel Specification.* `step_correct` is a predicate over triples containing a past trace, a request trace, and a response trace.

```
Axiom readn:
 forall (f: chan) (n: positive) {tr: Trace},
 ST (list ascii)
    {traced tr * open f}
    {fun l =>
      traced (ReadN f n l :: tr) *
      [len l = n] * open f }.

Definition read_msg:
 forall (t: tab) {tr: Trace},
 ST msg
    {traced tr * open (tchan t)}
    {fun m =>
      traced (ReadMsg t m ++ tr) * open (tchan t)} :=
  ...
```

Figure 2.6. *Example Monadic Types.* Code showing the monadic types for the `readn` primitive and for the `read_msg` function.

2.4.3 Monads in Ynot Revisited

In the previous section, we explained Ynot's `ST` monad as being parameterized over a single type `T`. In reality, `ST` takes two additional parameters representing pre- and post-conditions for the computation encoded by the monad. Thus, `ST T P Q` represents a computation which, if started in a state where `P` holds, may perform some I/O and then return a value of type `T` in a state where `Q` holds. For technical reasons, these pre- and post-conditions are expressed using separation logic, but we defer details to a tech report [37].

Following the approach of Malecha et al. [54], we define an *opaque* predicate `(traced tr)` to represent the fact that at a given point during execution, `tr` captures all the past activities; and `(open f)` to represent the fact that channel `f` is currently open. An opaque predicate cannot be proven directly. This property allows us to ensure that no part of the kernel can forge a proof of `(traced tr)` for any trace it independently constructs. Thus `(traced tr)` can only be true for the current trace `tr`.

Figure 2.6 shows the full monadic type for the `readn` primitive, which reads `n` bytes of data and returns it. The $*$ connective represents the separating conjunction from

separation logic. For our purposes, consider it as a regular conjunction. The precondition of `(readn f n tr)` states that `tr` is the current trace and that `f` is open. The post-condition states that the trace after `readn` will be the same as the original, but with an additional `(ReadN f n l)` action at the beginning, where the length of `l` is equal to `n` (`len l = n` is a regular predicate, which is lifted using square brackets into a separation logic predicate). After the call, the channel `f` is still open.

The full type of the Ynot `bind` operation makes sure that when two monads are sequenced, the post-condition of the first monad implies the pre-condition of the second. This is achieved by having `bind` take an additional third argument, which is a proof of this implication. The syntactic sugar for `x <- a; b` is updated to pass the wildcard "`_`" for the additional argument. When processing the definition of our kernel, Coq will enter into an interactive mode that allows the user to construct proofs to fill in these wildcards. This allows us to prove that the post-condition of each monad implies the pre-condition of the immediately following monad in Coq's interactive proof environment.

2.4.4 Back to the Kernel

We now return to our kernel from Figure 2.3 and show how we prove that it satisfies the spec from Figure 2.5. We augment the kernel state to additionally include the trace of the kernel so far, and we update our kernel code to maintain this `tr` field. By using a special encoding in Ynot for this trace, the `tr` field is not realized at run-time, it is only used for proof purposes.

We define the `kcorrect` predicate as follows (`s.tr` projects the current trace out of kernel state `s`):

```
Definition kcorrect (s: kstate) :=
  traced s.tr * [tcorrect s.tr]
```

Now we want to show that `kcorrect` is an invariant that holds throughout execution of

the kernel. Essentially we must show that `(kcorrect s)` is a loop invariant on the kernel state `s` for the main kernel loop, which boils down to showing that `(kcorrect s)` is valid as both the pre- and post-condition for the loop body, `kstep` as shown in Figure 2.3.

As mentioned previously, Coq will ask us to prove implications between the post-condition of one monad and the pre-condition of the next. While these proofs are ultimately spelled out in full formal detail, Coq provides facilities to automate a substantial portion of the proof process. Ynot further provides a handful of sophisticated tactics which helped automatically dispatch tedious, repeatedly occurring proof obligations. We had to manually prove the cases which were not handled automatically. While we have only shown the key kernel invariant here, in the full implementation there are many additional Hoare predicates for the intermediate goals between program points. We defer details of these predicates and the manual proof process to [37], but discuss proof effort in Section 2.5.

2.4.5 Security Properties

Our security properties are phrased as theorems about the spec. We now prove that our spec implies these key security properties, which we intend to hold in QUARK. Figure 2.7 shows these key theorems, which correspond precisely to the security properties outlined in Section 2.2.6.

State Integrity. The first security property, `kstate_dep_user`, ensures that the kernel state only changes in response to the user pressing a "control key" (e.g. switching to the third tab by pressing F3). The theorem establishes this property by showing its contrapositive: if the kernel steps by responding with `rsp` to request `req` after trace `tr` *and* no "control keys" were read from the user, then the kernel state remains unchanged by this step. The function `proj_user_control`, not shown here, simply projects from the trace all actions of the form `(Read c stdin)` where `c` is a control key. The function

`kernel_state`, not shown here, just computes the kernel state from a trace. We also prove that at the beginning of any invocation to `kloop` in Figure 2.3, all fields of `s` aside from `tr` are equal to the corresponding field in `(kernel_state s.tr)`.

Response Integrity. The second security property, `kresponse_dep_kstate`, ensures that every kernel response depends solely on the request and the kernel state. This delineates which parts of a trace can affect the kernel's behavior: for a given request `req`, the kernel will produce the same response `rsp`, for any two traces that *induce the same kernel state*, even if the two traces have completely different sets of requests/responses (recall that the kernel state only includes the current tab and the set of tabs, and most request responses don't change these). Since the kernel state depends only the user's control key inputs, this theorem immediately establishes the fact that *our browser will never allow one component to influence how the kernel treats another component unless the user intervenes*.

Note that `kresponse_dep_kstate` shows that the kernel will produce the same response given the same request after any two traces that induce the same kernel state. This may seem surprising since many of the kernel's operations produce nondeterministic results, *e.g.,* there is no way to guarantee that two web fetches of the same URL will produce the same document. However, such nondeterminism is captured in the request, which is consistent with our notion of requests as inputs and responses as outputs.

Tab Non-Interference. The second security property, `tab_NI`, states that the kernel's response to a tab is not affected by any other tab. In particular, `tab_NI` shows that if in the context of a valid trace, `tr1`, the kernel responds to a request `req` from tab `t` with `rsp1`, then the kernel will respond to the same request `req` with an equivalent response in the context of any other valid trace `tr2` which also contains tab `t`, irrespective of what other tabs are present in `tr2` or what actions they take. Note that this property holds in particular for the case where trace `tr2` contains *only* tab `t`, which leads to the

following corollary: the kernel's response to a tab will be the same even if all other tabs did not exist

The formal statement of the theorem in Figure 2.7 is made slightly more complicated because of two issues. First, we must assume that the focused tab at the end of `tr1` (denoted by `cur_tab tr1`) is `t` if and only if the focused tab at the end of `tr2` is also `t`. This additional assumption is needed because the kernel responds differently based on whether a tab is focused or not. For example, when the kernel receives a `Display` message from a tab (indicating that the tab wants to display its rendered page to the user), the kernel only forwards the message to the output process if the tab is currently focused.

The second complication is that the communication channel underlying the cookie process for `t`'s domain may not be the same between `tr1` and `tr2`. Thus, in the case that kernel responds by forwarding a valid request from `t` to its cookie process, we guarantee that the kernel sends the same payload to the cookie process corresponding to `t`'s domain.

Note that, unlike `kresponse_dep_kstate`, `tab_NI` does not require `tr1` and `tr2` to induce the same kernel state. Instead, it merely requires the request `req` to be from a tab `t`, and `tr1` and `tr2` to be valid traces that both contain `t` (indeed, `t` must be on both traces otherwise the `step_correct` assumptions would not hold). Other than these restrictions, `tr1` and `tr2` may be arbitrarily different. They could contain different tabs from different domains, have different tabs focused, different cookie processes, etc.

Response Integrity and Tab Non-Interference provide different, complimentary guarantees. Response Integrity ensures the response to *any* request `req` is only affected by control keys and `req`, while Tab Non-Interference guarantees that the response to a tab request does not leak information to another tab. Note that Response Integrity could still hold for a kernel which mistakenly sends responses to the wrong tab, but Tab Non-Interference prevents this. Similarly, Tab Non-Interference could hold for a kernel which allows a tab to affect how the kernel responds to a cookie process, but Response

Integrity precludes such behavior.

It is also important to understand that `tab_NI` proves the absence of interference as caused by the *kernel*, not by other components, such as the network or cookie processes. In particular, it is still possible for two websites to communicate with each other through the network, causing one tab to affect another tab's view of the web. Similarly, it is possible for one tab to set a cookie which is read by another tab, which again causes a tab to affect another one. For the cookie case, however, we have a separate theorem about cookie integrity and confidentiality which states that cookie access control is done correctly.

Note that this property is an adaptation of the traditional non-interference property. In traditional non-interference, the program has "high" and "low" inputs and outputs; a program is non-interfering if high inputs never affect low outputs. Intuitively, this constrains the program to never reveal secret information to untrusted principles.

We found that this traditional approach to non-interference fits poorly with our trace-based verification approach. In particular, because the browser is a non-terminating, reactive program, the "inputs" and "outputs" are infinite streams of data.

Previous research [13] has adapted the notion of non-interference to the setting of reactive programs like browsers. They provide a formal definition of non-interference in terms of possibly infinite input and output streams. A program at a particular state is non-interfering if it produces *similar* outputs from *similar* inputs. The notion of similarity is parameterized in their definition; they explore several options and examine the consequences of each definition for similarity.

Our tab non-interference theorem can be viewed in terms of the definition from [13], where requests are "inputs" and responses are "outputs"; essentially, our theorem shows the inductive case for potentially infinite streams. Adapting our definition to fit directly in the framework from [13] is complicated by the fact that we deal with

a unified trace of input and output events in the sequence they occur instead of having one trace of input events and a separate trace of output events. In future work, we hope to refine our notion of non-interference to be between domains instead of tabs, and we believe that applying the formalism from [13] will be useful in achieving this goal. Unlike [13], we prove a version of non-interference for a particular program, the QUARK browser kernel, directly in Coq.

No Cross-domain Socket Creation. The third security property ensures that the kernel never delivers a socket bound to domain d to a tab whose domain does not match d. This involves checking URL suffixes in a style very similar to the cookie policy as discussed earlier. This property forces a tab to use `GetURL` when accessing websites that do not match its domain suffix, thus restricting the tab to only access publicly available data from other domains.

Cookie Integrity/Confidentiality. The fourth security property states cookie integrity and confidentiality. As an example of how cookies are processed, consider the following trace when a cookie is set:

```
SetCookie key value cproc ::
SetCookieRequest tab key value :: ...
```

First, the `SetCookieRequest` action occurs, stating that a given `tab` just requested a cookie (in fact, `SetCookieRequest` is just defined in terms of a `ReadMsg` action of the appropriate message). The kernel responds with a `SetCookie` action (defined in terms of `WroteMsg`), which represents the fact that the kernel sent the cookie to the cookie process `cproc`. The kernel implementation is meant to find a `cproc` whose domain suffix corresponds to the `tab`. This requirement is given in the theorem `no_xdom_cookie_set`, which encodes cookie integrity. It requires that, within a correct trace, if a cookie process is ever asked to set a cookie, then it is in immediate

response to a cookie set request for the same exact cookie from a tab whose domain matches that of the cookie process. There is a similar theorem `no_xdom_cookie_get`, not shown here, which encodes cookie confidentiality.

Domain Bar Integrity and Correctness. The fifth property states that the domain bar is equal to the domain suffix of the currently selected tab, which encodes the correctness of the address bar.

2.5 Evaluation

In this section we evaluate QUARK in terms of proof effort, trusted computing base, performance, and security.

Proof Effort and Component Sizes. QUARK comprises several components written in various languages; we summarize their sizes in Figure 2.8. All Python components share the "Python Message Lib" for messaging with the kernel. Implementing QUARK took about 6 person months, which includes several iterations redesigning the kernel, proofs, and interfaces between components. Formal shim verification saved substantial effort: we guaranteed our security properties for a million lines of code by reasoning just 859.

Trusted Computing Base. The trusted computing base (TCB) consists of all system components we assume to be correct. A bug in the TCB could invalidate our security guarantees. QUARK's TCB includes:

- Coq's core calculus and type checker
- Our formal statement of the security properties
- Several primitives used in Ynot
- Several primitives unique to QUARK

- The Ocaml compiler and runtime
- The underlying Operating System kernel
- Our chroot sandbox

Because Coq exploits the Curry-Howard Isomorphism, its type checker is actually the "proof checker" we have mentioned throughout the chapter. We assume that our formal statement of the security properties correctly reflects how we understand them intuitively. We also assume that the primitives from Ynot and those we added in QUARK correctly implement the monadic type they are axiomatically assigned. We trust the OCaml compiler and runtime since our kernel is extracted from Coq and run as an OCaml program. We also trust the operating system kernel and our traditional chroot sandbox to provide process isolation, specifically, our design assumes the sandboxing mechanism restricts tabs to only access resources provided by the kernel, thus preventing compromised tabs from commuting over covert channels.

Our TCB does *not* include WebKit's large code base or the Python implementation. This is because a compromised tab or cookie process can not affect the security guarantees provided by kernel. Furthermore, the TCB does not include the browser kernel code, since it has been proved correct.

Ideally, QUARK will take advantage of previous formally verified infrastructure to minimize its TCB. For example, by running QUARK in seL4 [41], compiling QUARK's ML-like browser kernel with the MLCompCert compiler [1], and sandboxing other QUARK components with RockSalt [58], we could drastically reduce our TCB by eliminating its largest components. In this light, our work shows how to build yet another piece of the puzzle (namely a verified browser) needed to for a fully verified software stack. However, these other verified building blocks are themselves research prototypes

which, for now, makes them very difficult to stitch together as a foundation for a realistic browser.

Performance. We evaluate our approach's performance impact by comparing QUARK's load times to stock WebKit. Figure 2.9 shows QUARK load times for the top 10 Alexa Web sites, normalized to stock WebKit. QUARK's overhead is due to factoring the browser into distinct components which run in separate processes and explicitly communicate through a formally verified browser kernel.

By performing a few simple optimizations, the final version of QUARK loads large, sophisticated websites with only 24% overhead. This is a substantial improvement over a naïve implementation of our architecture, shown by the left-most "not-optimized" bars in Figure 2.9. Passing bound sockets to tabs, whitelisting content distribution networks for major websites, and caching cookie accesses, improves performance by 62% on average.

The WebKit baseline in Figure 2.9 is a full-featured browser based on the Python bindings to WebKit. These bindings are simply a thin layer around WebKit's C/C++ implementation which provide easy access to key callbacks. We measure 10 loads of each page and take the average. Over all 10 sites, the average slowdown in load-time is 24% (with a minimum of 5% for `blogger` and a maximum of 42% for `yahoo`).

We also measured load-time for the previous version of QUARK, just before rectangle-based rendering was added. In this previous version, the average load-time was only 12% versus 24% for the current version. The increase in overhead is due to additional communication with the kernel during incremental rendering. Despite this additional overhead in load time, incremental rendering is preferable because it allows QUARK to display content to the user as it becomes available instead of waiting until an entire page is loaded.

Security Analysis. QUARK provides strong, formal guarantees for security

policies which are not fully compatible with traditional web security policies, but still provide some of the assurances popular web browsers seek to provide.

For the policies we have not formally verified, QUARK offers exactly the same level of traditional, unverified enforcement WebKit provides. Thus, QUARK actually provides security far beyond the handful policies we formally verified. Below we discuss the gap between the subset of policies we verified and the full set of common browser security policies.

The *same origin policy* [70] (SOP) dictates which resources a tab may access. For example, a tab is allowed to load cross-domain images using an `img` tag, but not using an `XMLHttpRequest`.

Unfortunately, we cannot easily verify this policy since restricting how a resource may be used after it has been loaded (*e.g.,* in an `img` tag vs. as a JavaScript value) requires reasoning across abstraction boundaries, *i.e.,* analyzing the large, complex tab implementation instead of treating it as a black box.

The SOP also restricts how JavaScript running in different frames on the same page may access the DOM. We could formally reason about this aspect of the SOP by making frames the basic protection domains in QUARK instead of tabs. To support this refined architecture, frames would own a rectangle of screen real estate which they could subdivide and delegate to sub-frames. Communication between frames would be coordinated by the kernel, which would allow us to formally guarantee that all frame access to the DOM conforms with the SOP.

We only formally prove inter-domain cookie isolation. Even this coarse guarantee prohibits a broad class of attacks, *e.g.,* it protects all Google cookies from any non-Google tab. QUARK does enforce restrictions on cookie access between subdomains; it just does so using WebKit as unverified cookie handling code within our cookie processes. Formally proving finer-grained cookie policies in Coq would be possible and would not

require significant changes to the kernel or proofs.

Unfortunately, Quark does not prevent all cookie exfiltration attacks. If a subframe is able to exploit the entire tab, then it could steal the cookies of its top-level parent tab, and leak the stolen cookies by encoding the information within the URL parameter of `GetURL` requests. This limitation arises because tabs are principles in Quark instead of frames. This problem can be prevented by refining Quark so that frames themselves are the principles.

Our socket security policy prevents an important subset of cross-site request forgery attacks [9]. Quark guarantees that a tab uses a `GetURL` message when requesting a resource from sites whose domain suffix doesn't match with the tab's one. Because our implementation of `GetURL` does not send cookies, the resources requested by a `GetURL` message are guaranteed to be publicly available ones which do not trigger any privileged actions on the server side. This guarantee prohibits a large class of attacks, *e.g.,* cross-site request forgery attacks against Amazon domains from non-Amazon domains. However, this policy cannot prevent cross-site request forgery attacks against sites sharing the same domain suffix with the tab, *e.g.,* attacks from a tab on www.ucsd.edu against cse.ucsd.edu since the tab on www.ucsd.edu can directly connect to cse.ucsd.edu using a socket and cookies on cse.ucsd.edu are also available to the tab.

Compatibility Issues. QUARK enforces non-standard security policies which break compatibility with some web applications. For example, Mashups do not work properly because a tab can only access cookies for its domain and subdomains, *e.g.,* a subframe in a tab cannot properly access any page that needs user credentials identified by cookies if the subframe's domain suffix does not match with the tab's one. This limitation arises because tabs are the principles of Quark as opposed to subframes inside tabs. Unfortunately, tabs are too coarse grained to properly support mashups and retain our strong guarantees.

For the same reason as above, Quark cannot currently support third-party cookies. It is worth noting that third-party cookies have been considered a privacy-violating feature of the web, and there are even popular browser extensions to suppress them. However, many websites depend on third party cookies for full functionality, and our current Quark browser does not allow such cookies since they would violate our non-interference guarantees.

Finally, Quark does not support communications like "postMessage" between tabs; again, this would violate our tab non-interference guarantees.

Despite these incompatibilities, Quark works well on a variety of important sites such as Google Maps, Amazon, and Facebook since they mostly comply with Quarks' security policies. More importantly, our hope is that in the future Quark will provide a foundation to explore all of the above features within a formally verified setting.

In particular, adding the above features will require future work in two broad directions. First, frames need to become the principles in Quark instead of tabs. This change will require the kernel to support parent frames delegating resources like screen region to child frames. Second, finer grained policies will be required to retain appropriate non-interference results in the face of these new features, e.g. to support interaction between tabs via "postMessage". Together, these changes would provide a form of "controlled" interference, where frames are allowed to communicate directly, but only in a sanctioned manner. Even more aggressively, one may attempt to re-implement other research prototypes like MashupOS [31] within Quark, going beyond the web standards of today, and prove properties of its implementation.

There are also several other features that Quark does not currently support, and would be useful to add, including local storage, file upload, browser cache, browser history, etc. However, we believe that these are not fundamental limitations of our approach or Quark's current design. Indeed, most of these features don't involve inter-tab

communication. For the cases where they do (for example history information is passed between tabs if visited links are to be correctly rendered), one would again have to refine the non-interference definition and theorems to allow for controlled flow of information.

2.6 Discussion

In this section we discuss lessons learned while developing QUARK and verifying its kernel in Coq. In hindsight, these guidelines could have substantially eased our efforts. We hope they prove useful for future endeavors.

Formal Shim Verification. Our most essential technique was *formal shim verification*. For us, it reduced the verification burden to proving a small browser kernel. Previous browsers like Chrome, OP, and Gazelle clearly demonstrate the value of kernel-based architectures. OP further shows how this approach enables reasoning about a model of the browser. We take the next step and formally prove the actual browser implementation correct.

Modularity through Trace-based Specification. We ultimately specified correct browser behavior in terms of traces and proved both that (1) the implementation satisfies the spec and (2) the spec implies our security properties. Splitting our verification into these two phases improved modularity by separating concerns. The first proof phase reasons using monads in Ynot to show that the trace-based specification correctly abstracts the implementation. The second proof phase is no longer bound to reasoning in terms of monads – it only needs to reason about traces, substantially simplifying proofs.

This modularity aided us late in development when we proved address bar correctness (Theorem `dom_bar_correct` in Figure 2.7). To prove this theorem, we only had to reason about the trace-based specification, not the implementation. As a result, the proof of `dom_bar_correct` was only about 300 lines of code, tiny in comparison to the total proof effort. Thus, proving additional properties can be done with relatively little

effort over the trace-based specification, without having to reason about monads or other implementation details.

Implement Non-verified Prototype First. Another approach we found effective was to write a non-verified version of the kernel code before verifying it. This allowed us to carefully design and debug the interfaces between components and to enable the right browsing functionality before starting the verification task.

Iterative Development. After failing to build and verify the browser in a single shot, we found that an iterative approach was much more effective. We started with a text-based browser, where the tab used `lynx` to generate a text-based version of QUARK. We then evolved this browser into a GUI-based version based on WebKit, but with no sockets or cookies. Then we added sockets and finally cookies. When combined with our philosophy of "write the non-verified version first", this meant that we kept a working version of the kernel written in Python throughout the various iterations. Just for comparison, the Python kernel which is equivalent to the Coq version listed in Figure 2.8 is 305 lines of code.

Favor Ease of Reasoning. When forced to choose between adding complexity to the browser kernel or to the untrusted tab implementation, it was *always* better keep the kernel as simple as possible. This helped manage the verification burden which was the ultimate bottleneck in developing QUARK. Similarly, when faced with a choice between flexibility/extensibility of code and ease of reasoning, we found it best to aim for ease of reasoning.

2.7 Future work

One important direction for future work is to refine the principles of QUARK to resolve the current compatibility issues and provide stronger security guarantees. For future work, frames must become the principles in QUARK instead of tabs. This

change will support Mashups with multiple cross-domain subframes in one tab. The kernel will be required to support parent frames delegating resources like screen region to child frames. Finer grained policies will also be required to retain appropriate non-interference results in the face of these new features, e.g. to support interaction between tabs via ”postMessage”. Together, these changes would provide a form of ”controlled” interference, where frames are allowed to communicate directly, but only in a sanctioned manner.

Even more aggressively, one may attempt to provide formal guarantees for the same origin policy. Enforcing SOP will require the QUARK kernel to control how cross-domain resources (*e.g.,* image resources) are used in an HTML renderer, and this seems quite challenging in terms of engineering efforts.

Another direction for future work is to support various features of the modern browsers for QUARK. These might include local storage, file upload, browser cache, browser history, etc. However, we believe that these are not fundamental limitations of our approach or QUARK's current design. Indeed, most of these features don't involve communications between tabs. For the cases where they do (*e.g.,* history information is passed between tabs if visited links are to be correctly rendered), one would again have to refine the non-interference definition and theorems to allow for controlled flow of information.

2.8 Conclusions

In this chapter, we demonstrated how *formal shim verification* can be used to achieve strong security guarantees for a modern Web browser using a mechanical proof assistant. We formally proved that our browser provides tab noninterference, cookie integrity and confidentiality, and address bar integrity and correctness. We detailed our design and verification techniques and showed that the resulting browser, QUARK,

provides a modern browsing experience with performance comparable to the default WebKit browser. For future research, QUARK furnishes a framework to easily experiment with additional web policies without re-engineering an entire browser or formalizing all the details of its behavior from scratch.

2.9 Acknowledgements

This chapter, in full, is adapted from material as it appears in Usenix Security Symposium 2012. Jang, Dongseok; Tatlock, Zachary; Lerner, Sorin, Usenix Association, 2012. The dissertation author was a primary investigator and author of this paper.

```
Theorem kstate_dep_user:
  forall tr req rsp,
  step_correct tr req rsp ->
  proj_user_control tr
    = proj_user_control (rsp ++ req ++ tr) ->
  kernel_state tr = kernel_state (rsp ++ req ++ tr).

Theorem kresponse_dep_kstate:
  forall tr1 tr2 req rsp,
  kernel_state tr1 = kernel_state tr2 ->
  step_correct tr1 req rsp ->
  step_correct tr2 req rsp.

Theorem tab_NI:
  forall tr1 tr2 t req rsp1 rsp2,
  tcorrect tr1 -> tcorrect tr2 ->
  from_tab t req ->
  (cur_tab tr1 = Some t <-> cur_tab tr2 = Some t) ->
  step_correct tr1 req rsp1 ->
  step_correct tr2 req rsp2 ->
  rsp1 = rsp2 \/
  (exists m, rsp1 = WroteCMsg (cproc_for t tr1) m /\
             rsp2 = WroteCMsg (cproc_for t tr2) m).

Theorem no_xdom_sockets: forall tr t,
  tcorrect tr ->
  In (SendSocket t host s) tr ->
  is_safe_soc host (domain_suffic t).

Theorem no_xdom_cookie_set: forall tr1 tr2 cproc,
  tcorrect (tr1 ++ SetCookie key value cproc :: tr2) ->
     exists tr t,
       (tr2 = (SetCookieRequest t key value :: tr) /\
        is_safe_cook (domain cproc) (domain_suffix t))

Theorem dom_bar_correct: forall tr,
  tcorrect tr -> dom_bar tr = domain_suffix (cur_tab tr).
```

Figure 2.7. *Kernel Security Properties.* This Coq code shows how traces allow us to formalize QUARK's security properties.

Component	Language	Lines of code
Kernel Code	Coq	859
Kernel Security Properties	Coq	142
Kernel Proofs	Coq	4,383
Kernel Primitive Specification	Coq	143
Kernel Primitives	Ocaml/C	538
Tab Process	Python	229
Input Process	Python	60
Output Process	Python	83
Cookie Process	Python	135
Python Message Lib	Python	334
WebKit Modifications	C	250
WebKit	C/C++	969,109

Figure 2.8. QUARK *Components by Language and Size.*

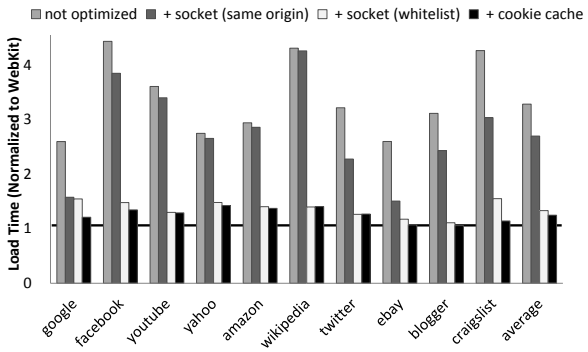


Figure 2.9. QUARK *Performance.* QUARK load times for the Alexa Top 10 Web sites.

Chapter 3

Language and Automation Co-design

Implementing systems in proof assistants like Coq and proving their correctness in full formal detail has consistently demonstrated promise for making extremely strong guarantees about critical software, ranging from compilers and operating systems to databases and web browsers. Unfortunately, these verifications demand such heroic manual proof effort, even for a *single* system, that the approach has not been widely adopted.

We demonstrate a technique to eliminate the manual proof burden for verifying many properties within an entire *class* of applications, in our case reactive systems, while only expending effort comparable to the manual verification of a single system. A crucial insight of our approach is simultaneously designing both (1) a domain-specific language (DSL) for expressing reactive systems and their correctness properties and (2) proof automation which exploits the constrained language of both programs and properties to enable fully automatic, pushbutton verification. We apply this insight in a deeply embedded Coq DSL, dubbed REFLEX, and illustrate REFLEX's expressiveness by implementing and automatically verifying realistic systems including a modern web browser, an SSH server, and a web server. Using REFLEX radically reduced the proof burden: in previous, similar versions of our benchmarks written in Coq by experts, proofs accounted for over 80% of the code base; our versions require no manual proofs.

The REFLEX project was a joint project with three other PhD students, Daniel Ricketts, Valentin Robert, and Dongseok Jang. The disseration author's primary contribution in QUARK was the architecture of REFLEX's kernel framework, as well as the implementation and formalization of this general kernel mechanism in Coq.

3.1 Introduction

Software systems like the OS and web browser are responsible for manipulating private, sensitive data in domains ranging from banking and medical record management to email and social networking. Bugs in these systems lead not only to reliability failures, but also security failures, allowing attackers to gain access to secret information or tamper with trusted outputs. Unfortunately, testing alone has proven insufficient for preventing such dangerous errors.

Over the past decade, one approach has consistently demonstrated the potential to establish extremely strong guarantees for safety- and security-critical software: implement the system in a proof assistant like Coq and interactively prove its correctness fully formally. This approach has been successfully applied in a variety of systems, including compilers [50], operating systems [42], web servers [55], database systems [52], and web browsers [38]. These systems provide extremely strong guarantees because the programmer is forced to correctly handle every corner case in full formal detail. While severe, this discipline enables its practitioners to develop software which is substantially more reliable and secure than traditionally developed code. In fact, Yang et al. [88] demonstrate *experimentally* that the fully formally verified CompCert C compiler [50] is significantly more robust and reliable than its non-verified competitors like GCC and LLVM.

While attractive, the strong guarantees of fully formal verification often only come at an exorbitant price: heroic manual proof effort carried out by highly trained experts.

For example, the initial CompCert development comprised 4,400 lines of compiler code and 28,000 lines devoted to verification. In terms of effort, the ratio of programming to proving is typically far worse than line counts imply. Implementing the kernel for the seL4 verified OS took only 2 person *months*, while verifying those roughly 9,000 lines of C code required over 20 person *years*. In addition to their substantial size, these proofs demand deep expertise in the specialized logic and tactic language of the proof assistant. Such tactics are notoriously difficult to debug and maintain, which also makes *modifying* verified software extremely expensive. In the end, all this effort results in the verification of a *single* program. Unfortunately, these costs have been prohibitive for all but the most critical applications.

In this chapter, we demonstrate an approach to eliminate the manual proof burden (and need for Coq expertise) in verifying many properties within an entire *class* of applications, while only expending effort comparable to the manual verification of a single system. We focus on the domain of *reactive systems*: programs which listen for input requests, perform computations necessary to service requests, reply with output responses, and then return to listening for additional requests. To support automatic, fully formal verification of important properties for reactive systems, we simultaneously design two mutually dependent entities: (1) a domain-specific language (DSL) for implementing reactive systems and specifying their properties and (2) proof automation tactics which exploit the structure of programs and properties in the DSL to eliminate all manual proof obligations. We dub this design methodology *Language and Automation Co-design* (LAC).

General purpose verification frameworks like Ynot [62, 20] and Bedrock [19] provide unbounded expressiveness of programs and properties, which makes complete automation of formal Coq proofs intractable, LAC restricts both the structure of programs and properties to gain much better traction on proof automation. By automatically

constructing foundational Coq proofs for applications written in the DSL, we aim to make fully formal verification accessible even for programmers with no previous experience using proof assistants, and to significantly reduce the costs of fully formal verification for those with previous experience. Furthermore, modifying such applications does not create any additional proof burden since the verification is carried out fully automatically.

We applied LAC to design REFLEX, a DSL for expressing reactive systems as event-processing loops, and specifying properties these programs should satisfy. Properties can characterize which observable run-time behaviors the system may exhibit and specify which classes of system components should not interfere. We implemented REFLEX as a deeply embedded DSL in Coq and built upon the Ynot, trace-based verification approach of Malecha et al. [55].

As a case study, we applied REFLEX to build and automatically prove properties about *privilege separated systems* [65], an important type of reactive system. In such systems, sensitive resources are protected by running most of an application's code in separate, strictly sandboxed processes and routing interactions through a small kernel which ensures that all communications and resource accesses are allowed by the security policy. We used REFLEX to implement and formally verify the kernel of three realistic, privilege separated systems: (1) a modern web browser capable of running popular websites like Facebook, GMail, and Amazon which mediates resource access and interaction between browser tabs in the style of Chrome [68], (2) an SSH daemon which provides remote, secure terminal access for unmodified SSH clients in the style of [65] and (3) a web server providing file access subject to user authentication and an access control policy.

To summarize, this chapter makes the following contributions:

- We describe REFLEX, a DSL deeply embedded in Coq, which completely eliminates the manual Coq proof burden for verifying many important properties of

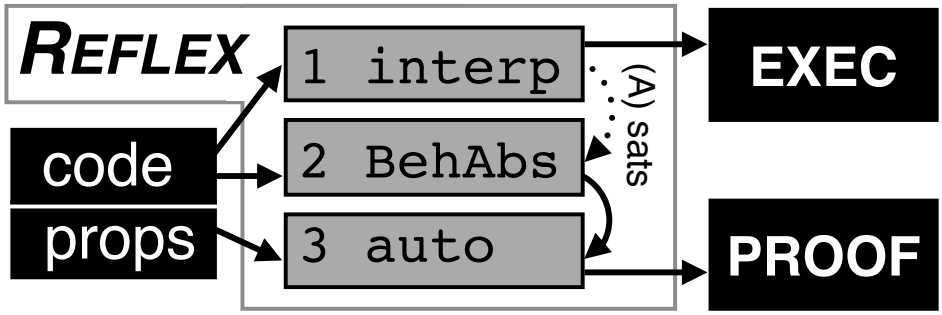


Figure 3.1. REFLEX Overview. REFLEX programs implement reactive systems and specify their properties.

realistic *reactive systems* (Sections 3.2, 3.3, and 3.4). REFLEX programs express both reactive systems and their properties, and REFLEX provides tactics to automatically construct fully formal Coq proofs that guarantee a given REFLEX program satisfies its user-provided correctness properties (Section 3.5).

- We demonstrate the expressiveness, utility, and practicality of REFLEX by implementing and verifying several security properties for three large privilege separated systems: a modern web browser, an SSH server, and a web server (Section 3.6).

- We describe a set of general design principles that arose from applying *Language and Automation Co-design* (LAC) to develop REFLEX. LAC's key insight lies in the simultaneous design of both (1) the language of programs and properties and (2) proof automation tactics to guarantee programs satisfy their user-provided properties. We discuss how LAC enabled us to eliminate the formal proof burden in REFLEX, along with additional lessons learned (Section 3.7).

3.2 Overview

Figure 3.1 illustrates the REFLEX system which consists of (1) an interpreter to run REFLEX programs, (2) a function BehAbs which, given a program *P*, computes

a behavioral abstraction characterizing the sequences of observable actions, or *traces*, P may produce, and (3) tactics which automatically search for a proof that any trace satisfying $\mathsf{BehAbs}(P)$ also satisfies user-provided safety and security properties. We also proved once and for all, manually in Coq, that the traces produced by running the interpreter on program P satisfy $\mathsf{BehAbs}(P)$. Thus, for a given correctness property C, if REFLEX tactics are able to construct a proof that $\mathsf{BehAbs}(P)$ satisfies C, then we have an *end-to-end* guarantee that all runs of P through the interpreter satisfy C.

Example: SSH Server. We illustrate REFLEX through a running example: the kernel of a privilege separated SSH server. Figure 3.2 shows the architecture of our SSH server, which closely follows the privilege separation approach of Provos et al. [65]. In particular, the server is separated into several communicating components, each running in its own sandboxed process, with a kernel that mediates communication between the components. From a security viewpoint, this approach provides the advantage that sensitive resources can be protected from complex and vulnerability-prone components, while the kernel remains simple and reliable. For example, consider the Connection component which manages communication with the outside network. A complete compromise of this process from a buffer overflow in the packet processing library does not immediately cause the server to grant access to unauthenticated users (as would be the case in a regular SSH server), since the kernel and Password modules, which are separated from the Connect module, control user authentication.

REFLEX Code. Figure 3.3 shows a simplified version of the SSH kernel we developed and verified using REFLEX. The kernel (written using REFLEX) is a reactive system that orchestrates communication between all components (written in C and Python) in the SSH server. The components and the kernel communicate using messages over channels implemented as Unix domain sockets. The kernel continuously services request messages from components.

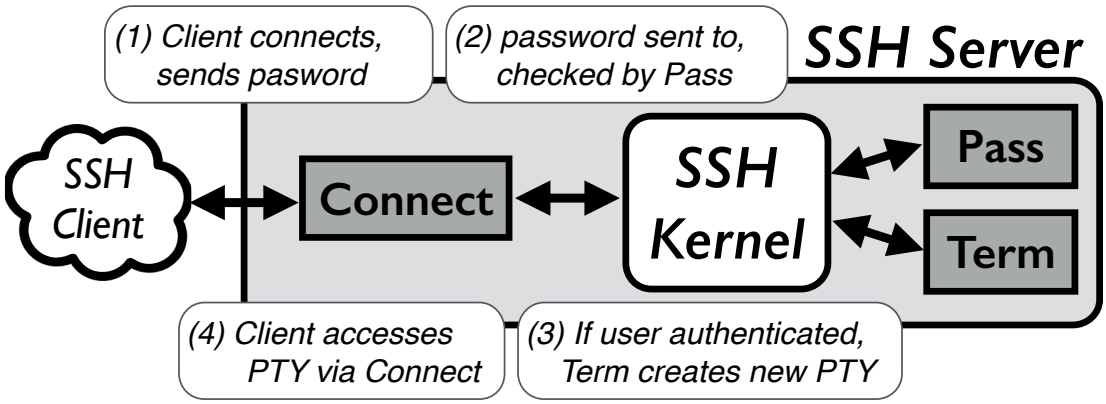


Figure 3.2. Simplified SSH Architecture. REFLEX enabled us to implement and formally verify an SSH server.

REFLEX programs comprise five sections: The `Components` section declares the types of components the kernel communicates with, along with the executable on disk for each component type. The kernel creates a new component of type τ by spawning a new process running τ's executable and setting up a channel to the new component instance. The `Messages` section declares the different message types exchanged between components and the kernel. The `Init` section contains code to declare global variables and initialize the system. In this case, the `authorized` variable is a pair of a user name and boolean, which indicates the authentication state of the kernel: if `authorized == (user, true)`, then the kernel believes that the given `user` has authenticated properly. The three other declarations in the `Init` section declare and initialize a component of each type for the kernel to communicate with.

The kernel functionality is expressed in the `Handlers` section, which is a list of request/response rules to continuously apply. For our SSH example, the first rule states that, if the kernel receives a message from a `Connection` component, and this message is of the form `ReqAuth(user, pass)`, indicating that `user` wants to authenticate using password `pass`, then the kernel responds by forwarding the message to `P`, which is initial-

ized in the `Init` as the `Password` component. If the `Password` component determines the password is valid, it responds with an `Auth(user)` message back to the kernel. The second rule in the `Handlers` section specifies that when an `Auth(user)` is received from the `Password` component, the kernel should set its authentication state to indicate that the given `user` has authenticated. The third rule specifies how the kernel responds when the `Connection` component requests a terminal for a given `user`: if the authentication state indicates that `user` has authenticated, then the request for a terminal is forwarded to the `Terminal` component. The `Terminal` component will then create the terminal for the given `user`, and respond back to the kernel with a `Term(user, fd)` response (where `fd` is a file descriptor to the terminal PTY). Finally, the fourth rule states that when the kernel receives a `Term(user, fd)` message from the `Terminal` component, it forwards the information to the client (through the `Connection` component), but *only* if the user has authenticated. In any cases where the user did not specify a message handler, the kernel simply sends no response and returns to its event processing loop, waiting for the next input message.

Note that each handler specifies how to respond when a *type* of component sends a message, not a particular component. In our SSH example, there is exactly one component of each type, but in other programs, the kernel may spawn and service an arbitrary number of components of any particular type. For example, our web browser, discussed in Section 3.6, may spawn many `Tab` s.

Properties. The last part of Figure 3.3 shows an example property that should hold on all traces of the kernel. A *trace* records all observable interactions between the kernel and the outside world. Receiving a message from a component and spawning a component are examples of such interactions; we call each such interaction an *action*. The property `AuthBeforeTerm` states that in any execution, before the kernel requests a terminal for any user `u` from the `Terminal` component, `u` must have been authenticated

by the `Password` component. This property is expressed using predicates over actions called *action patterns*, a primitive of the policy language for REFLEX called Enables, and universally quantified variables. For any two action patterns `A` and `B`, `A` Enables `B` on a trace if any action matching `B` in the trace is preceded at some point in the trace by an action matching `A`. The universally quantified variable `u` in `AuthBeforeTerm` ensures that for any string `u`, a request for a terminal for `u` is preceded by a successful authentication of `u`, as opposed to a successful authentication of a different user. Section 3.4 further describes our property language.

Once the programmer has implemented and specified this SSH kernel in REFLEX, our system automatically generates an end-to-end proof in Coq that any run of the kernel will satisfy the user-provided security property. The programmer only needs to write the code in Figure 3.3, and does not need any proof assistant expertise. This level of fully automatic, pushbutton automation is our main contribution, which we achieve through the co-design of the REFLEX DSL along with proof-automation techniques. The remainder of this chapter further details the REFLEX system and how we achieve both expressiveness and proof automation.

```
Components :
  Connection "client.py"
  Password   "user-auth.c"
  Terminal   "pty-alloc.c"

Init :
  authorized = ("", false)
  C <= spawn(Connection)
  P <= spawn(Password)
  T <= spawn(Terminal)

Properties :
  AuthBeforeTerm: forall u,
    [Recv(Password, Auth(u))]
    Enables
    [Send(Terminal, ReqTerm(u))]

Messages :
  ReqAuth(string, string)
  Auth(string)
  ReqTerm(string)
  Term(string, fdesc)

Handlers :
Connection=>ReqAuth(user, pass):
  send(P, ReqAuth(user, pass))
Password=>Auth(user):
  authorized = (user, true)
Connection=>ReqTerm(user):
  if (user, true) == authorized
    send(T, ReqTerm(user))
Terminal=>Term(user, t):
  if (user, true) == authorized
    send(C, Term(user, t))
```

Figure 3.3. Simplified SSH Kernel in REFLEX DSL.

3.3 The REFLEX DSL for Reactive Systems

In this section, we describe the REFLEX interpreter and the BehAbs function, which, given a program P, computes a behavioral abstraction characterizing the traces P can produce. We emphasize design decisions following LAC that allow us to prove once and for all, manually in Coq, that the traces produced by running the interpreter on program P satisfy $\mathsf{BehAbs}(P)$. We illustrate these REFLEX features at a high level, but the full, commented REFLEX implementation is available online.

3.3.1 The REFLEX Language

As we saw in the previous section, REFLEX programs implement reactive systems primarily as a sequence of *handlers* which are registered to run when the kernel receives an appropriate message from a component. The programmer implements these handlers using mostly standard imperative programming features (assignment to global variables, sequencing, branching).

REFLEX handlers can also include commands to send a message to a component, spawn a new component, invoke a custom OCaml function returning a string, and look up an existing component based on its type and what we call its *configuration*. A component configuration is a *read-only* record whose fields are set when the component is spawned. Configurations help distinguish between components of a given type and play a crucial role in expressing safety and security properties. For example, in our web browser kernel implemented in REFLEX, a tab's domain is stored in its configuration. Making configurations read-only aided proof automation. Looping constructs are notably *absent* from handler code; this plays a crucial role in both the definition of BehAbs and proof automation.

We use a python frontend to translate the concrete REFLEX syntax to the abstract

```
Definition wf_state (s: state) : hprop :=
  fds_open s.comps * traced s.tr * ...

Definition step :
 (handle: comp -> msg -> cmd) -> (s: state) -> (s': state)
   PRE : wf_state s * BehAbs s
   POST : wf_state s' * BehAbs s' :=
  c  <- select s.comps s.tr <@> ...;
  m  <- recv_msg c (Select(c) :: s.tr) <@> ...;
  let tr := Recv(c, m) :: Select(c) :: s.tr in
  s' <- run_cmd (s.comps, tr, s.env) (handle c m) <@> ...;
  Return s'

Definition run_cmd :
  (c: cmd) -> (s: state) -> (s': state)
    PRE : wf_state s
    POST : wf_state s' :=
  match c with
  | Assign x e =>
      Return (s.comps, s.tr, s.env[x := eval e s])
  | Spawn t cfg =>
      f <- spawn t cfg s.tr <@> ...;
      let c := (t, cfg, f) in
      Return (c::s.comps, Spawn(c)::s.tr, s.env)
  | ...
  end

  (* Write a string to open channel c *)
  send : (c: chan) -> (s: str) -> {tr: Trace } -> unit
  PRE : { traced tr /\ open c }
  POST : { traced (SendS c s :: tr) /\ open c }
```

Figure 3.4. REFLEX Interpreter.

syntax tree of the program. In addition to providing user convenience, this also allows us to insulate the programmer from complex dependent types used in the REFLEX implementation. This is crucial since we make heavy use of dependent types in Coq to ensure that REFLEX programmers never "go wrong" by attempting to access undefined variables or execute an effectful primitive without satisfying its preconditions.

3.3.2 REFLEX Interpreter

REFLEX programs implement reactive systems, which are fundamentally impure and non-terminating. To support these features in Coq's otherwise pure, strongly normalizing core calculus, we wrote our interpreter using the Ynot [62] library for encoding monadic stateful computations in Coq. The REFLEX interpreter interacts with the outside

world using Ynot to invoke effectful operations; each operation is guarded by low-level pre-conditions to ensure their proper use, *e.g.,* sends may only be performed on open file descriptors. For example, at the bottom of Figure 3.4, we see the Ynot axiomatization of the send primitive which takes as input a channel *c* and string *s* to write to *c*. Additionally send takes the current *trace* as an input.

As mentioned in the previous section, traces record the sequence of observable actions a program performs, *i.e.,* the sequence of calls to Ynot primitives. The trace is represented as a list recording which primitives were called, with what arguments, and what results they produced. The list is stored in *reverse chronological order*, so that the most recent calls to primitives are at the head of the list. Traces are threaded through the interpreter, but they are ghost variables only used for verification – they are not actually generated in the executable code. The axiomatization of send also includes pre- and post-conditions to support the Ynot trace-based verification approach. In addition to requiring that *c* be an open channel, send requires that its trace argument satisfy the `traced` predicate which follows a linear typing discipline to ensure there is at most one, unforgeable "current trace" [55]. The send post-conditions show how the current trace is updated to record this call and also that the channel *c* is still open.

The top part of Figure 3.4 shows an overview of the REFLEX interpreter. The central `step` function operates over program states which include the current list of components, the trace of actions performed so far, and an environment mapping variables to values. The `step` function repeatedly selects a component that is ready, reads a message from the component, determines which command should be run based on the handler rules, and then interprets the appropriate command using `run_cmd`.

Figure 3.4 shows two illustrative cases for `run_cmd`: (1) assignment, which is standard, and (2) spawn which updates the state by adding the newly spawned component to the list of components and adds a `Spawn` action to the head of the trace. Several

additional cases are not shown, including sequencing, conditionals, message sends, and component look-up. All of these are standard, except for component look-up, which works as follows: `lookup` takes an expression *e*, which is a predicate over a component's configuration, and two commands c_1 and c_2 and searches the set of current components for a component of the appropriate type for which *e* evaluates to true. If such a component can be found, it is bound in the environment, and c_1 is run, otherwise c_2 is run.

We illustrate how execution proceeds: suppose the interpreter is running the SSH kernel from Figure 3.3 and, after initialization, has produced some state `s` with trace `s.tr`. Further, suppose the kernel selects the next ready component, gets the Connection component (`C`), and reads a `ReqAuth(u,p)` message from it. The kernel will then call the `run_cmd` function in order to execute the command from the appropriate handler, in this case, `send(P, ReqAuth(u, p))`. This will have two results: (A) the appropriate system calls will be made to send the message `ReqAuth(u, p)` to the password component `(P)` and (B) `step` will return a new state identical to `s`, except that the new trace will be updated to:

```
Send P ReqAuth(u,p) :: Recv C ReqAuth(u,p)
                                    :: Select C :: s.tr
```

Note again that the trace is stored in *reverse chronological order*, so that *the most recent calls to primitives are recorded at the head of the list*. Finally, to correctly track all the traces a program can produce: (1) each REFLEX primitive in Ynot takes the current trace as an extra argument, and returns an updated trace which reflects the call to that primitive (2) each primitive also has pre and post conditions which state how the resulting trace is related to the incoming trace.

3.3.3 Behavioral Abstractions

To separate low level requirements from higher level, user-specified properties, verification in REFLEX is broken into two phases. First, we carried out *program independent* verification manually once and for all in Coq. This proof shows that for any

program P, running P with the REFLEX interpreter always satisfies the low-level, Ynot pre-conditions required to execute primitives, that the trace of the state returned by `step` captures exactly the observable behaviors from running P, and that this trace will be included in $\mathsf{BehAbs}(P)$. Second, *program dependent* verification is performed automatically by REFLEX tactics (described in Section 3.5), and proves that a given program P's user-specified properties hold on all possible execution traces included in $\mathsf{BehAbs}(P)$. Following LAC guided our design of REFLEX to exhibit this clean separation (discussed in Section 3.7).

We define the function to compute behavioral abstractions, BehAbs, inductively: BehAbs holds on the state after the **init** code is run, and inductively on any state resulting from an *exchange* with a component, captured by the Exchange relation. The Exchange relation $s \overset{c,m}{\rightsquigarrow} s'$ holds when starting at state s, the interpreter receives message m from component c, and responds by symbolically evaluating the appropriate handler to produce state s'. This ability to symbolically evaluate handlers using a pure, total Coq function is crucial in defining the critical BehAbs definition, made possible by our LAC-inspired decision to omit looping constructs from handlers.

3.4 REFLEX Properties

The previous section showed how we implement reactive systems in REFLEX and formalize their behavior in terms of their *traces* of observable actions. REFLEX also enables programmers to specify correctness properties for these systems by indicating when sensitive actions are required, permitted, or forbidden and how components may interact. These properties can be used to capture common safety and security policies that arise in practice, *e.g.*,, in our SSH server example from Section 3.2, where system access is only granted once a user has correctly authenticated.

In this section we detail REFLEX property primitives which programmers can

compose to encode high level, intuitive safety and security policies. In the next section we show how REFLEX's proof automation exploits the structure of REFLEX programs and properties to eliminate the manual proof burden. Then in Section 3.6, we demonstrate how REFLEX property primitives can be used to capture important correctness properties for several realistic systems including an SSH server and modern web browser.

REFLEX properties come in two flavors: (1) *trace properties* which specify which traces a correct program may produce, and (2) *non-interference properties* which specify that one set of components may not affect another.

3.4.1 Trace Properties

Programmers specify which traces a REFLEX program is allowed to produce via *trace patterns*, which only match traces satisfying constraints on order of actions. For example, the sample SSH kernel in Section 3.2, uses the Enables trace pattern to require that a user correctly authenticates before accessing the underlying system.

REFLEX provides five primitive trace patterns: ImmBefore, ImmAfter, Enables, Ensures, and Disables. These are inspired by temporal logic, but kept simple following our LAC design decision. Each of these primitives are in turn parameterized by *action patterns* that range over trace elements. Action patterns are simply actions whose fields can contain literals, variables, or wildcards. Thus, the action pattern `Send(C(), M(3,_,s))` matches any `Send` action whose recipient is a component of type `C` with an empty configuration, and whose message is of type `M` with a payload containing first a 3, then any value, and finally a value matching the variable `s`. All variables are universally quantified at the outermost level. Below, we detail each primitive REFLEX trace pattern.

Immediately Before. This primitive specifies traces that must contain adjacent patterns: ImmBefore `A B` holds if for each action *b* that matches `B`, the action that happened immediately before *b* matches `A` (note that, since *traces are recorded in reverse*

chronological order, the action that happened immediately before *b* actually occurs right after *b* in the list representing the trace). We formally define ImmBefore as follows:

```
Definition immbefore A B tr := forall b pre suf,
  AMatch B b -> tr = suf ++ b :: pre ->
  exists a pre', AMatch A a /\ pre = a :: pre'.
```

Enables. The Enables primitive is a relaxation of ImmBefore: Enables `A B` holds if for each action *b* matching `B`, there is an action matching `A` that occurs temporally before *b*. Enables is defined as follows:

```
Definition enables A B tr : forall b pre suf,
  AMatch B b -> tr = suf ++ b :: pre ->
  exists b pre' suf',
    AMatch A a /\ pre = suf' ++ a :: pre'.
```

Disables. We also provide Disables, an analog of Enables: Disables `A B` holds if for each action matching `B`, there is no previous action matching `A`.

```
Definition disables A B tr : forall b pre suf,
  AMatch A a -> tr = suf ++ a :: pre ->
  AMatch B b -> ~ In b suf.
```

Immediately After and Ensures. Two additional primitives arise as the temporal duals of ImmBefore and Enables, respectively ImmAfter and Ensures (Disables is self-dual, and thus does not give rise to another primitive). ImmAfter specifies that a pattern must appear immediately after any occurrence of another. Ensures specifies that the occurrence of an action matching one pattern ensures the presence of an action matching the second pattern later.

```
Definition immafter A B tr := immbefore B A (rev tr).
Definition ensures  A B tr := enables B A (rev tr).
```

```
Components :                  Messages :
  Engine "engine.c"             Crash()
  Doors  "doors.c"              Accelerating()
  Radio  "radio.c"              DoorsM(string)
                                Volume(string)
Init :
  E <= spawn(Engine)          Handlers :
  D <= spawn(Doors)             Engine=>Crash():
  R <= spawn(Radio)               send(D, DoorsM("unlock"))
                                Engine=>Accelerating():
Properties :                      send(R, Volume("crank it up"))
  NoInterfere                   Doors=>DoorsM(s):
    [Engine] [E]                  if s == "open":
                                    send(R, Volume("mute"))
```

Figure 3.5. Simplified REFLEX Kernel for Car Controller.

3.4.2 Non-interference

Many systems comprise modules of mixed criticality. Ensuring security or robustness in these systems often involves guaranteeing that high-criticality components are isolated from low-criticality components. For example, consider the REFLEX program shown in Figure 3.5 for controlling and coordinating different components of an automobile. In this example, the user wants to ensure that messages from less critical components, Doors and Radio, do not interfere with the critical Engine component. As another example, in a web browser we want to ensure that a tab for domain `A` does not affect the kernel's behavior towards tabs for domain `B`. To support such non-interference policies, REFLEX provides a primitive expressing that, for a given partition of components into low and high, low components cannot interfere with high components.

Traditionally, for non-reactive, deterministic programs, a user specifies non-interference by partitioning program inputs and outputs into low and high sets. Non-interference holds with respect to this partitioning iff for any two executions with the same high inputs, the program *terminates* with the same high outputs, that is, the high outputs are deterministic with respect to the high inputs, and therefore isolated from any influence from low inputs. However, a reactive system continually receives inputs from, and sends outputs to, components; by design it does not terminate.

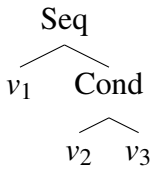
To address this issue, we adapt ideas on Reactive Non-Interference from [14]. As in the traditional setting, we specify non-interference by partitioning components into high and low sets. We would now like to define non-interference to hold iff, for any two executions where the kernel receives the same, possibly infinite, *sequence* of inputs from high components, it sends the same, possibly infinite, *sequence* of outputs to the high components. Unfortunately, simply adapting ideas from [14] is not sufficient for REFLEX programs, which pose yet another challenge, one that, to our knowledge, has not been previously explored in the context of reactive systems. In particular, REFLEX programs are not only reactive, but also *non-deterministic*, something that was not considered in [14]. Non-determinism in REFLEX programs arises from calls to Ynot primitives, which are implemented as OCaml functions that interact with the outside world. These primitives can be used to perform program specific tasks such as retrieving the contents of a webpage. From the perspective of a REFLEX program, primitives produce results non-deterministically. Thus, even in a system where one would intuitively expect non-interference to hold with respect to some partitioning, there may be two executions with precisely the same high inputs but different high outputs, simply because the high outputs depend on some values non-deterministically generated by Ynot primitives.

A common definition of non-interference in the presence of non-determinism is called possibilistic non-interference [56]. Possibilistic non-interference holds iff, for any reachable state, the kernel's behavior from the perspective of the high components would be *possible*, even if the kernel received no messages from low components. Unfortunately, it is well known that possibilistic non-interference is not preserved under refinement [91]; that is, possibilistic non-interference can hold on a specification of a system but not hold on the implementation of that system. In REFLEX, it is not possible to precisely characterize the behaviors allowed by the implementation because this would require precisely specifying the behavior of every outside system with which the kernel interacts

(e.g. the OS, web servers, etc.). Thus, possibilistic non-interference would not provide any guarantees about the implementations of REFLEX programs.

Instead of using possibilistic non-interference, our approach to non-determinism consists of tweaking the conditions under which two executions must produce the same sequence of high outputs. Intuitively, we would like to say that non-interference holds iff, for any two executions where the kernel receives the same sequence of inputs from high components *and* the non-deterministic outside world behaves the same way during high handlers, the kernel sends the same sequence of outputs to high components. We do so by factoring out non-deterministic values as additional ghost inputs to the kernel. In particular, we modify the definition of an input to the kernel to include both a message from a component *and* the non-deterministic context under which the handler for that message runs. A non-deterministic context is a tree containing the values that could be produced by the outside world (i.e. resulting from invoking a Ynot primitive) in the course of running the handler. The tree, which is now an additional input to the kernel for a given execution of a handler, follows the structure of the handler's code. Each leaf in the tree corresponds to an atomic command in the code whose execution would result in the introduction of some non-deterministic value from the outside world. For example, in the following hypothetical handler, the non-deterministic context contains three leaves corresponding to the three strings that could be produced by the invocations of Ynot primitives:

```
C=>M():
  s1 <- wget(...)
  if ...
  then s2 <- wget(...)
  else s3 <- wget(...)
```



The hypothetical context tree to the right of the handler contains the three strings v_1, v_2, and v_3. If the handler were executed under this context, then v_1 would be assigned

to `s1`, and either v_2 would be assigned to `s2` or v_3 would be assigned to `s3`, depending on which branch was taken.

Unlike messages from components, these context trees are not actually produced at runtime. Instead they are only used to specify which pairs of executions of a non-interfering REFLEX program must produce the same high outputs.

With this new definition of input, we can now capture our intuitive notion of non-interference for reactive systems with non-determinism. Non-interference holds iff for any two executions where the kernel receives the same inputs from high components (where each input includes both a message from a high component and the non-deterministic context for the corresponding handler), then the kernel sends the same outputs to high components. Thus, our definition of non-interference guarantees that any interference occur indirectly by influencing the non-deterministic context of high handlers. In other words, interference can only occur if low handlers influence non-deterministic values originating outside of the REFLEX kernel. This amounts to interference through channels that occur outside of the kernel (for example, through outside web servers), which REFLEX has no way of preventing. Note that unlike possibilistic non-interference, our definition of non-interference is preserved under refinement.

There is another, final, subtlety in expressing non-interference for programs written in REFLEX. Previous work on non-interference has assumed the set of components (in the case of reactive systems) or variables is fixed. While REFLEX programs have a fixed, finite set of component *types*, the set of actual components may change during execution. Partitioning components by type alone is too coarse, as a user would be unable to specify many important properties, *e.g.,* that tabs for different domains do not interfere. Instead, we allow the user to provide a *function* θ_c that takes a component's type *and configuration* and returns the label low or high. Thus, the user provides a partitioning of all possible sets of components that can arise rather than for some fixed set.

To formally define non-interference, we use two auxiliary projection functions π_i and π_o which take as input a user provided labeling function θ_c and state s. The function $\pi_i(\theta_c, s)$ returns the list of pairs of recv actions and corresponding non-deterministic context trees for messages received from high components in s (according to the labeling θ_c). The function $\pi_o(\theta_c, s)$ returns the list of send and spawn actions in the trace of s for messages sent to, and spawns of, high components (according to the labeling θ_c). With these projections, we can now formally define non-interference:

Definition 1 (Non-interference)**.** *Non-interference holds with respect to a component labeling function θ_c iff:*

$$\forall\, s_1\ s_2, \mathsf{BehAbs}\ s_1 \Rightarrow \mathsf{BehAbs}\ s_2 \Rightarrow$$
$$\pi_i(\theta_c, s_1) = \pi_i(\theta_c, s_2) \Rightarrow \pi_o(\theta_c, s_1) = \pi_o(\theta_c, s_2)$$

The above definition states that, for any two reachable states s_1 and s_2, if s_1 and s_2 contain the same high inputs (including both received messages and nondeterministic contexts), then they contain the same messages sent to high components. This essentially shows that any interference from low components must occur through influence of nondeterministic values originating outside the REFLEX kernel.

This section has described the various properties provided by REFLEX for specifying desirable properties REFLEX programs should satisfy. The next section discusses how we automatically prove these properties fully formally in Coq.

3.5 Proof Automation in REFLEX

Currently, the tremendous cost of manually constructed formal proofs prohibits broad adoption of highly reliable, proof assistant based, verification techniques. Our goal for REFLEX is to eliminate this prohibitive manual proof burden. To automate the proof that a user's code satisfies their policies, REFLEX follows our *Language and Automation*

Co-design (LAC) technique: by focusing on a particular domain and carefully designing the languages for programming and specification hand in hand with proof automation, REFLEX is able to automatically verify common safety and security properties for reactive systems. Below we discuss how LAC enabled us to construct powerful tactics that achieve this goal.

3.5.1 Automatically Proving Trace Properties

The tactics we implemented for showing that REFLEX programs satisfy their trace properties follow a general pattern: perform induction over BehAbs for the given program, and (1) for the base case, check that the program state after running **init** satisfies the policy, (2) for the inductive case, show that, assuming the policy has held on any execution up to this point, for any message sent by any component, the property will continue to hold after running the appropriate handler. This last step breaks into a number of subcases, each corresponding to a particular message type and component type. Each of these cases is typically solved in one of two ways:

1. Symbolically evaluate the handler on the abstract state being considered and evaluate the policy on the resulting abstract state. This is sufficient to solve the goal when the handler maintains the validity of the property regardless of which handlers were run in the past.

2. Otherwise, prove that relevant branch conditions cannot be satisfied without also satisfying obligations required by the given property (*e.g.,* for `AuthBeforeTerm`, the trace must contain a valid authentication). In this way the tactics are able to discover non-trivial program invariants and adapt to common idioms in user code, for instance that a variable's value guards a potentially dangerous send of a sensitive resource to a component.

We illustrate this process by diving into a detailed example, stepping through Coq proof goals to show how our tactics handle the Enables property for the SSH kernel from Section 3.2. In this case, our tactics must establish the authentication policy which requires that the kernel passes a logged-in terminal for user *u* to a connection *only after* that connection has provided a valid password for *u*.

We will only describe the inductive cases, as the base case is strictly easier for the automation. To prove the goal, our tactics consider each possible path through the handler, *which is straightforward because handlers were designed to be loop free*, and shows for each path that the property will be satisfied. The subgoal in all of these cases follows the following pattern:

```
E := forall u, [Recv(Password, Auth(u))]
       Enables [Send(Terminal, ReqTerm(u))]
s  : state             (* start at state s     *)
HR : BehAbs s          (*   which is reachable *)
IH : rprop_sats E s.tr (*   and satisfies prop *)
c  : component         (* receive m from c     *)
m  : message
...
s' : state :=          (* run handler, get state s'  *)
   { authorized := ... (* state updated by handler   *)
   ; ...   (* T' contains the last actions performed *)
   ; tr := T' ++ Recv c m :: Select s.comps c :: s.tr }
============================================================
rprop_sats E s'.tr
```

The hypotheses (above the line) characterize a valid exchange: in some reachable state *s*, the kernel receives message *m* from component *c*, and runs the user-specified handler to end up in state s'. Additionally the hypotheses provide proofs that *s* is a

reachable state and it satisfies our property E. Now the tactic's obligation is to prove from these hypotheses that our property E still holds on the new s'.

For example, in the first handler of the SSH server (when the Connection component sends a ReqAuth message), our proof obligation looks like this pattern, where s' is the state obtained by running this particular handler. In particular, it generates the following subtrace:

```
T' := Send s.P ReqAuth(m.user, m.pass) :: nil
```

Our tactics easily solve this goal: since the extended trace sends no messages to Terminal and *s.tr* satisfies the property, our goal is immediately satisfied. The case for when a Connection component sends a ReqTerm message is more complex. If the branch condition `(user, true) == authorized` evaluates to *false*, then the handler takes no actions, and again the property is trivially established. However, if the branch is taken, the subgoal context becomes:

```
H  : (m.user, true) == s.authorized
T' := Send s.T ReqTerm(m.user) :: nil
```

Note how this case additionally includes the fact that the branch condition evaluated to true. Adding these branch conditions to the context is crucial to verification, as we will see shortly.

In this case, the handler *does* send a message to Terminal, and so our property does not immediately hold. In particular, we must show that somewhere in the history of *s.tr*, the kernel received a message from the Password component authenticating user *m.user*. Since such an action does not occur earlier in the same handler, our tactics must perform a second induction over BehAbs. In each resulting subgoal, our tactics show that the corresponding handler either (A) contains a message received from the Password component authenticating user *m.user*, (B) does not affect the value of the `authorized`

state variable, or (C) invalidates the branch condition `(user, true) == authorized`, resulting in a contradiction. Adding branch conditions to the context is essential here, as it prunes unfeasible paths.

3.5.2 Automatically Proving Non-interference

Unfortunately, the general definition of non-interference given in the previous section does not lend itself immediately to proof automation, because the associated tactic would essentially need to track information flow through REFLEX program variables, tantamount to implementing a static taint tracking engine. Instead, in addition to a labeling θ_c of components as high or low, we also require a simple labeling function θ_v of global variables in the REFLEX program as high or low. This is actually an advantage of LAC: when proof automation is difficult, it is often possible to ask simple, specialized questions of the user in order to guide the search for inductive invariants. Given such a variable labeling, we define $\textit{h-vals}(\theta_v, s)$ to be the values of all variables in s that are labeled high by θ_v.

We now strengthen Definition 1 to ease automatically proving non-interference. First, define $\text{NI}_{inv}(\theta_c, \theta_v, s, s')$ as:

$$\pi_o(\theta_c, s) = \pi_o(\theta_c, s') \ \wedge \textit{h-vals}(\theta_v, s) = \textit{h-vals}(\theta_v, s'))$$

Definition 2 (Non-interference (strengthened))**.** *Non-interference holds with respect to labellers for components θ_c and global variables θ_v iff*

$$\forall\, s_1\ s_2, \mathsf{BehAbs}\ s_1 \Rightarrow \mathsf{BehAbs}\ s_2 \Rightarrow$$
$$\pi_i(\theta_c, s_1) = \pi_i(\theta_c, s_2) \Rightarrow \text{NI}_{inv}(\theta_c, \theta_v, s_1, s_2)$$

This strengthened definition allows us to prove formally in Coq that two simple conditions are sufficient for establishing non-interference. In order to state these

sufficient conditions, we need a variant of the Exchange relation from Section 3.3 with one additional parameter, the non-deterministic context. We now say that the Exchange relation $s \overset{c,m,t}{\rightsquigarrow} s'$ holds when starting at state s, the interpreter receives message m from component c and responds by symbolically evaluating the appropriate handler with non-deterministic context t to produce state s'. We have proven, in Coq, that BehAbs defined using this variant of Exchange also produces sound behavioral abstractions for any REFLEX program.

Using this variant of Exchange, we can give the two sufficient conditions for establishing non-interference:

Theorem 1. *Non-interference holds with respect to labellers for components θ_c and global variables θ_v if the following two conditions hold:*

$$\text{NI}_{lo} : \forall\ c\ m\ t\ s\ s', [\theta_c(c) = low \wedge s \overset{c,m,t}{\rightsquigarrow} s'] \Rightarrow [\text{NI}_{inv}(\theta_c, \theta_v, s, s')]$$

$$\begin{aligned}\text{NI}_{hi} : \forall\ & c\ m\ t\ s_1\ s_1'\ s_2\ s_2', \\ & [\theta_c(c) = high \wedge s_1 \overset{c,m,t}{\rightsquigarrow} s_1' \wedge s_2 \overset{c,m,t}{\rightsquigarrow} s_2'] \Rightarrow \\ & [\text{NI}_{inv}(\theta_c, \theta_v, s_1, s_2)] \Rightarrow [\text{NI}_{inv}(\theta_c, \theta_v, s_1', s_2')]\end{aligned}$$

Intuitively, NI_{lo} requires that all low handlers in a program never send messages to, or spawn, high components and never update any state variables labeled as high. Let $\text{Agree}(s_1, s_2)$ hold iff states s_1 and s_2 agree on all high inputs, high outputs, and high variable values. Then, NI_{hi} requires that, if Agree holds on any two states and the same handler is run on both, then Agree will also hold on the resulting states. We implemented a tactic which attempts to automatically verify that the above two conditions hold on user-provided programs for given components and variable labeling functions by symbolically evaluating each handler.

While the reasoning for proving non-interference in the hypothetical car kernel is relatively simple, the non-interference tactic must perform more sophisticated reasoning when branches appear in handlers. In particular, the tactic uses facts implied by the labeling of variables and of the component list to try to show that in a handler run on two arbitrary states satisfying Agree, branches will take the same path. This is possible because of the interdependent design of branch commands, a strengthened definition of non-interference, and automation for non-interference.

3.5.3 Incompleteness of REFLEX Automation

While our tactics provide effective automation for many important properties in large, realistic examples (shown in Section 3.6), they are not complete. Thus, they may fail to find proofs for some properties expressible in REFLEX which in fact hold on a given REFLEX program.

As described earlier, our tactics attempt to use branch conditions to infer global inductive invariants. This often works in practice when programmers follow common programming idioms, but the heuristic is incomplete in general. For non-interference, this flaw is partially alleviated by the variable labeling function, provided by the programmer or by an automated, static taint-tracking engine.

Additionally, implementing robust, general tactics in Ltac, the tactic language of Coq presents a significant engineering challenge: tactics are inherently brittle and difficult to debug, *e.g.,* simple errors in Ltac pattern matching expressions are often easy to fix, but nearly impossible to detect early. Low-level logical errors, e.g. losing facts when inverting equalities with dependent types in the context, instantiating component lookup branch facts with the wrong term, etc., can also cause REFLEX automation to fail even when the conceptual proof search should succeed.

3.6 Evaluation

REFLEX aims to greatly reduce the cost of building reactive systems with strong, machine-checkable, correctness guarantees to the point that such systems can be built even by programmers unfamiliar with formal verification and proof assistants. We evaluate REFLEX by first showing that it is expressive enough to implement realistic applications and capture important safety and security properties. We next demonstrate that REFLEX's automation is sufficiently powerful to completely automate proofs for these key properties. We illustrate how REFLEX helped us catch bugs, discuss REFLEX's performance in terms of time to verify properties and usability of systems implemented in REFLEX, and finally report on the effort required to develop REFLEX.

3.6.1 REFLEX Expressiveness

To demonstrate REFLEX's expressiveness, we discuss three large, fully functional applications developed with REFLEX: a web browser, an SSH server, and a web server. To further evaluate the expressiveness of our policies and proof automation, we additionally implemented a substantially more detailed version of the hypothetical automobile controller sketched in the previous section and specified its key safety properties. Each benchmark consists of both: (1) a kernel performing security-critical operations and its safety properties, written in REFLEX, and (2) the implementation of surrounding components, built on existing infrastructures like WebKit for the browser, or OpenSSH for the SSH server, and sandboxed so that they can only communicate via the kernel. Table 3.1 shows their respective sizes. We next describe each of these benchmarks in further detail.

Web Browser. We used REFLEX to build a web browser kernel similar to Jang et al.'s Quark [38] browser kernel, which mediates access between different components of

Table 3.1. REFLEX benchmarks and their sizes (lines of code).

Component	Language	LOC
SSH Kernel Code / Properties	REFLEX	64 / 22
Sandboxed SSH Components	C, Python	89,567
Browser Kernel Code / Properties	REFLEX	81 / 37
Sandboxed Browser Components	C++, Python	970,240
Web Server Kernel Code / Properties	REFLEX	56 / 29
Sandboxed Web Server Components	Python	386

the browser. In particular, we reused Quark's publicly available browser components, but built a new browser kernel in REFLEX, providing the same functionality as Quark's kernel. Unlike the Quark authors, who took several months to *manually* verify their kernel, we *automatically* verify our browser kernel's security properties in minutes.

As in Quark, our browser runs each tab in a separate process, and the kernel mediates tab communication and access to system resources (*e.g.,* mouse, screen, and network). Furthermore, cookies are cached by separate cookie processes, one per domain, and the kernel mediates interactions between tabs and cookie processes. The original Quark kernel comprised 859 lines of Coq code; our kernel comprises only 81 lines of REFLEX code while providing the same functionality: we were able to browse popular, complex websites including Google Maps, GMail, Amazon, and Facebook.

Implementing a replacement browser kernel in REFLEX required some deviations from Quark's design. For example, Quark broadcasts cookie updates to all tabs of the relevant domain. Our kernel instead establishes private communication channels between tabs and the cookie process for their domain, which are then used for updates. Using REFLEX, we proved that this alternate architecture does not compromise our security guarantees. Our *domain non-interference* policy implies both *cookie confidentiality and integrity*, as well as a slightly relaxed version of Quark's *tab non-interference*: we allow tabs of the same domain to interfere, which morally provides the same level of security

while improving compatibility with modern websites. While Quark's manual proof of non-interference required a nearly 1,000 line custom Ltac script, our proof is carried out fully automatically.

SSH Server. As illustrated in previous sections, we also used REFLEX to build the kernel of an SSH server that mediates access between (A) an untrusted SSH Client module that processes raw network data from standard, unmodified, remote SSH clients and (B) root-privileged system resources. The kernel oversees authentication, and afterward provides file descriptors enabling direct communication with the PTY process to the SSH slave, thus eliminating any post-authentication overhead. We verified two SSH kernel properties: (1) the untrusted clients must first authenticate as a valid user to create a PTY process running as *that* user, and (2) untrusted SSH clients can attempt authentication at most 3 times. We encoded this second policy using four different properties (see Figure 3.2), demonstrating that despite the restricted design of our property language, it can be used to express sophisticated security policies. Moreover, future updates to REFLEX will include syntax for expressing common patterns such as at most n of some action. This syntax will immediately desugar to our existing primitives, so the power of our proof automation will remain.

Web Server. Our web server implements a simple file server with authentication. It comprises four components: one listens on the network, one performs access control checks, one accesses the filesystem, and one handles successfully-connected clients. The listener waits and notifies the kernel of connection attempts, which in turn consults the access controller to check permissions. Upon successful authentication, the kernel spawns a client component to handle this connection, otherwise the connection is dropped. Each client component handles its own connected client, and forwards file requests to the kernel, which checks them by consulting the access control component. On success, the kernel delivers the request to the disk component and forwards back the result.

Automobile. Koscher et al. [43] dramatically demonstrated the practical security vulnerabilities of modern automobiles. In particular, untrusted components of a car, such as the telematics unit, are able to inappropriately influence safety critical components, such as the engine and brakes. By inserting a verified kernel to act as a mediator between these components, we could ensure that, *e.g.,*, a radio controlled by an attacker cannot engage the brakes. Toward building such secure automobile controllers, we used REFLEX to implement a simple hypothetical kernel for controlling and coordinating the components of a car. The kernel mediates potential communication between components such as the engine, airbags, and door locks. For example, when the engine sends a message to the kernel indicating a crash, the kernel sends a message to deploy the airbags and unlock the doors, and sets a state variable preventing any component from locking the doors in the future. The kernel contains 60 lines of REFLEX code and properties. While REFLEX lacks several features be necessary for a realistic automobile kernel (e.g. real time constraints), our hypothetical kernel demonstrates both the robustness of our proof automation and the potential utility of LAC in another important domain.

3.6.2 Automation Effectiveness

We evaluate the effectiveness of REFLEX's proof automation by proving 41 important properties of our benchmarks, in Coq, fully automatically. This is the major benefit of REFLEX: by exploiting the constrained structure of both programs and their properties, we allow the user to reap significant benefits of formally verifying a system without requiring knowledge of advanced verification techniques such as dependent types and interactive proof assistants.

Figure 3.2 describes the properties for each benchmark and time taken to automatically discover a proof on a 3.4 GHz Intel Core i7 running Linux. These properties span every policy primitive and express important security and safety properties of real,

running systems. Many of these properties are non-trivial; being global in nature, they require reasoning across interactions of multiple handlers within a kernel.

3.6.3 REFLEX Utility

We illustrate the utility of REFLEX anecdotally: While developing our proof automation, we kept the web server benchmark completely separate and untested while guiding and debugging our development with smaller, earlier versions of other benchmarks. Once REFLEX automation was stable, we tested its robustness on the untried web server, where it failed to prove three properties. One of these failures revealed a low-level tactic bug that was easily fixed. However, the other two policies turned out to be false, and were successfully proven automatically once we corrected their statement. In another instance, during substantial modification of the web browser to use a different communication protocol, we inadvertently introduced subtle bugs which we did not discover until our proof automation failed to prove the affected properties.

3.6.4 REFLEX Performance

As shown in Figure 3.2 our slowest verification took just under nine minutes, and over 80% of our properties are proven automatically in under two minutes. In all cases, the verification time is many orders of magnitude less than the time required to construct traditional, manual Coq proofs, even excluding the extensive training required to master Coq. While early implementations of our proof search tactics suffered from severe performance issues, we were able to obtain tremendous speedups (80x on average and over 1000x for some benchmarks) and radically reduce memory usage (5x on average and over 35x for some benchmarks) by implementing several optimizations, including domain-specific reduction strategies and skipping symbolic evaluation of handlers for which a simple syntactic check suffices (both benefits of LAC), and saving subproofs at

key cut points to reduce the size of proof terms. Future work can explore incremental verification in order to further reduce the time required for re-verification.

Notice that we implemented three different versions of the web browser kernel and two different versions of the SSH kernel. Developing these variants required little additional effort after the first version of the kernel was built. One major advantage of REFLEX is that the modification of a kernel written in REFLEX incurs no additional proof burden: we only need to re-run the proof automation.

The generated executables run at interactive speeds: we used the web browser to access feature-rich, popular websites such as GMail, and ran the SSH and web servers without noticeable delay.

3.6.5 Development Effort

The REFLEX system consists of 7,635 lines of Coq code, including 2,827 lines for the REFLEX syntax and semantics, 2,786 lines of manual proofs, and 254 lines for the non-interference infrastructure. There are a total of 193 lines of REFLEX primitives implemented in OCaml. These include primitives that are exposed to the DSL user, namely `send` and `spawn`, but also primitives that are used internally like `recv` and `select`. Finally, there are 1,768 lines of tactic code to implement the formal Coq proof automation as described in Section 3.5.

The entire REFLEX implementation was written once and for all by us, the REFLEX developers. Following LAC, we designed our system so that programmers can build and verify reactive systems using REFLEX, while ignoring all details about its implementation and Coq. This design amortizes our substantial development costs: balancing the expressiveness and tractability of property primitives and building robust proof automation required many months of iteration. Moreover, this manual development effort is comparable to the development effort required in [38], yet it allowed us to

implement not only a fully formally verified browser similar to Quark, but also a realistic, verified SSH server and web server.

3.7 Discussion and Lessons Learned

We have seen how REFLEX design decisions enabled automatically verifying realistic reactive systems. Here we describe general lessons learned from our experience.

Language and Automation Co-design. Our experience building REFLEX suggests a general methodology for applying LAC, consisting of the following principles:

Constrain the implementation language to enable automated construction of behavioral abstractions. In our case, this took the form of BehAbs, a function to compute an inductively defined predicate characterizing the reachable states for a given REFLEX program. Abstractions like BehAbs enable a separation of concerns between handling gritty implementation details and automating higher level proof search. For REFLEX, we designed handlers to always terminate so that we could formally prove, once and for all, that behavioral abstractions computed by BehAbs are sound.

Give a uniform and structured semantics to both the language of programs and properties. This simplifies both inference of inductive invariants during proof search and user specifications that help guide this inference for richer properties like non-interference. In REFLEX, this principle manifests in several decisions including: (A) Using only a single event handling loop so that tactics need only consider highly structured traces. (B) Constraining handlers to be loop-free, enabling REFLEX tactics to easily symbolically evaluate all execution paths of a handler. (C) Designing trace properties so that branch conditions are often sufficient to strengthen policies into inductive invariants. (D) Using only a single event handling loop so that specifications for our strengthened non-interference definition only require a simple high and low global variable labeling; with this labeling, properties can typically be strengthened to inductive invariants

automatically.

Adapt language design to account for proof automation challenges. In REFLEX, the most interesting instance of this principle arose from the `lookup` primitive: We originally provided a more general `broadcast` primitive which sent a message to all components satisfying a predicate. However, `broadcast` complicated reasoning because a single `broadcast` command could generate an unbounded number of send actions; handling this unbounded behavior proved extraordinarily difficult. We instead use `lookup`, a simpler command which allowed us to maintain the invariant that each command generates a statically bound number of actions.

DSL Embedding vs Compiler. Initially, we implemented REFLEX using a compiler written in OCaml. This compiler would parse REFLEX code and properties, and generate not only the corresponding Coq code, but also tactics/proofs in Coq to establish the given properties on the generated code. While this approach allowed us to quickly build a prototype, maintenance soon became intractable. Each compiler bug we encountered required debugging the generated Coq code, including the generated tactics. Coq's tactic language, Ltac, is notoriously difficult to debug even when hand-crafted by experts; debugging *machine-generated* Ltac became an insurmountable challenge. To manage the complexity, we rewrote the system using a deeply embedded DSL in Coq. Although this required a large upfront cost, it provided several long-term benefits. First, it consolidated tactics into a single, more debuggable library. Second, Coq's dependent types caught many errors statically, errors which previously would have been found only after debugging machine-generated Coq code. Finally, the embedding also aided proof automation: since REFLEX programs are now well-typed by construction, there are many tedious corner cases concerning undefined behavior that simply no longer arise.

3.8 Conclusion

We described REFLEX, a deeply embedded Coq DSL, which eliminates the manual proof burden for verifying many properties for programs in the *class* of reactive systems. Building REFLEX only required effort comparable to the manual verification of a single realistic reactive system. Unlike general verification frameworks, where full formal proof automation is intractable due to the unrestricted nature of programs and properties, we achieved a high level of automation by following the general principle of *Language and Automation Co-design* and simultaneously designing both (1) REFLEX's languages for programs and properties and (2) REFLEX's proof automation tactics which exploit the constrained structure of programs and properties to automatically guarantee programs satisfy their user-provided properties. We evaluated REFLEX by verifying several important security properties for three critical systems: a modern web browser, an SSH server, and a web server.

3.9 Acknowledgements

This chapter, in full, is adapted from material as it appears in Programming Language Design and Implementation 2014. Rickets, Daniel, Robert, Valentin, Jang, Dongseok; Tatlock, Zachary; Lerner, Sorin, ACM Press, 2014. The dissertation author was a primary investigator and author of this paper.

Table 3.2. Benchmark Properties.

	Policy description	T (s)
car	Components do not interfere with the engine	13
	Airbags do deploy when there has been a crash	6
	Airbags are deployed immediately after crash	4
	Cruise control turns off immediately after braking	5
	Doors unlock when there is a crash	6
	Doors unlock immediately after airbags deployed	6
	Doors can not lock after a crash	21
	Airbags only deploy if there has been a crash	6
browser	Tab processes have unique IDs	70
	Cookie processes are unique per domain	75
	Cookies stay in their domain (tab, cookie process)	37
	Tabs are correctly connected to their cookie process	38
	Different domains do not interfere	229
	Tabs can only open sockets to allowed domains	94
browser_2	Tab processes have unique IDs	80
	Cookie processes are unique per domain	130
	Cookies stay in their domain (tab)	64
	Cookies stay in their domain (cookie process)	70
	Tabs are correctly connected to their cookie process	88
	Different domains do not interfere	338
	Tabs can only open sockets to allowed domains	106
browser_3	Tab processes have unique IDs	295
	Cookie processes are unique per domain	193
	Cookies stay in their domain (tab)	83
	Cookies stay in their domain (cookie process)	91
	Tabs are correctly connected to their cookie process	151
	Different domains do not interfere	532
	Tabs can only open sockets to allowed domains	78
ssh	Each login attempt enables the next one	54
	The first attempt to login disables itself	58
	The second attempt to login disables itself	297
	The third attempt to login disables all attempts	53
	Succesful login enables pseudo-terminal creation	55
ssh_2	Succesful login enables pseudo-terminal creation	113
	Login attempts approved by counter component	37
webserver	A client is only spawned on successful login	26
	Clients are never duplicated	70
	Files can only be requested after login	87
	Files are only requested after authorization	23
	Kernel only sends a file where the disk indicates	34
	Authorized requests are forwarded to disk	22

Chapter 4

Related Work

4.1 Related Work to Extensible Compilers

Our work is closely related to three lines of research: verified compilers, extensible compilers, and translation validation.

Verified Compilers Verified compilers are accompanied by a fully checked correctness proof which ensures that the compiler preserves the behavior of programs it compiles. Examples of such compilers include Leroy's CompCert compiler [49], Chlipala's compilers within the Lambda Tamer project [18, 17], and Nick Benton's work [12]. At a lower level, Sewell et. al.'s work [71] on formalizing the semantics of real-world hardware like the x86 instruction set provides a formal foundation for other verified tools to build on.

However, none of these compilers are easily extensible – extending these compilers with additional optimizations requires either modifying the proofs or trusting the new optimizations without proofs. The main goal of our work is to devise a mechanism to cross this extensibility barrier for verified compilers. Although our work was done in the context of the CompCert compiler, the general approach that we took for integrating PEC into a verified compiler could be applied to other verified compilers.

Extensible Compilers There has been a long line of work on making optimizers extensible. The Gospel language [86] allows compiler writers to express their optimizations in a domain-specific language, which can then be analyzed to determine interactions between optimizations. The Broadway compiler [29] allows the programmer to give detailed domain-specific annotations about library function calls, which can then be optimized more effectively. None of these systems, however, are geared at proving guarantees about correctness. The Rhodium [47] and PEC [44] work took the extensible compilers work in the direction of correctness checking. In these systems, correctness is checked fully automatically, but the execution engine is still trusted. Our current work shows how to bring a trusted execution engine to such systems.

Translation Validation Translation validation [64, 63, 69] is a technique for checking the correctness of a program transformation after it has been performed. Indeed, it is often easier to check that a particular instance of a transformation is correct than to show that transformation will always be correct. Although these techniques may increase our confidence that a compiler is producing correct code, only a *verified* translation validator can guarantee the correctness of the *a posteriori* check performed by the validator. Tristan et. al. examine such techniques for using verified translation validation to add more aggressive optimizations to CompCert while keeping the verification burden manageable [81, 82, 83].

4.2 Background and Related Work for Formally Verified Web Browswers

This section briefly discusses both previous efforts to improve browser security and verification techniques to ensure programs behave as specified.

Browser Security As mentioned in the Introduction, there is a rich literature on techniques to improve browser security [10, 84, 27, 75, 57, 16, 15]. We distinguish ourselves from all previous techniques by verifying the actual implementation of a modern Web browser and formally proving that it satisfies our security properties, all in the context of a mechanical proof assistant. Below, we survey the most closely related work.

Previous browsers like Google Chrome [10], Gazelle [84], and OP [27] have been designed using *privilege separation* [66], where the browser is divided into components which are then limited to only those privileges they absolutely require, thus minimizing the damage an attacker can cause by exploiting any one component. We follow this design strategy.

Chrome's design compromises the principles of privilege separation for the sake of performance and compatibility. Unfortunately, its design does not protect the user's data from a compromised tab which is free to leak all cookies for every domain. Gazelle [84] adopts a more principled approach, implementing the browser as a multi-principal OS, where the kernel has exclusive control over resource management across various Web principals. This allows Gazelle to enforce richer policies than those found in Chrome. However, neither Chrome nor Gazelle apply any formal methods to make guarantees about their browser.

The OP [27] browser goes beyond privilege separation. Its authors additionally construct a model of their browser kernel and apply the Maude model checker to ensure that this model satisfies important security properties such as the same origin policy and address bar correctness. As such, the OP browser applies insight similar to our work, in that OP focuses its formal reasoning on a small kernel. However, unlike our work, OP does not make any formal guarantees about the actual browser implementation, which means there is still a formality gap between the model and the code that runs. Our formal shim verification closes this formality gap by conducting all proofs in full formal detail

using a proof assistant.

Formal Verification Recently, researchers have begun using proof assistants to fully formally verify implementations for foundational software including Operating Systems [41], Compilers [51, 1], Database Management Systems [53], Web Servers [54], and Sandboxes [58]. Some of these results have even *experimentally* been shown to to drastically improve software reliability: Yang et al. [89] show through random testing that the CompCert verified C compiler is substantially more robust and reliable than its non-verified competitors like GCC and LLVM.

As researchers verify more of the software stack, the frontier is being pushed toward higher level platforms like the browser. Unfortunately, previous verification results have only been achieved at staggering cost; in the case of seL4, verification took over 13 person years of effort. Based on these results, verifying a browser-scale platform seemed truly infeasible.

Our formal verification of QUARK was radically cheaper than previous efforts. Previous efforts were tremendously expensive because researchers proved nearly every line of code correct. We avoid these costs in QUARK by applying *formal shim verification*: we structure our browser so that all our target security properties can be ensured by a very small browser kernel and then reason only about that single, tiny component. Leveraging this technique enabled us to make strong guarantees about the behavior of a million of lines of code while reasoning about only a few hundred in the mechanical proof assistant Coq.

We use the Ynot library [61] extensively to reason about imperative programming features, *e.g.,* impure functions like `fopen`, which are otherwise unavailable in Coq's pure implementation language. Ynot also provides features which allow us to verify QUARK in a familiar style: invariants expressed as pre- and post-conditions over program states,

essentially a variant of Hoare Type Theory [59]. Specifically, Ynot enables *trace-based verification*, used extensively in [54] to prove properties of servers. This technique entails reasoning about the sequence of externally visible actions a program may perform on any input, also known as *traces*. Essentially, our specification delineates which sequences of system calls the QUARK kernel can make and our verification consists of proving that the implementation is restricted to only making such sequences of system calls. We go on to formally prove that satisfying this specification implies higher level security properties like tab isolation, cookie integrity and confidentiality, and address bar integrity and correctness. Building QUARK with a different proof assistant like Isabelle/HOL would have required essentially the same approach for encoding imperative programming features, but we chose Coq since Ynot is available and has been well vetted.

Our approach is fundamentally different from previous verification tools like ESP [23], SLAM [7], BLAST [30] and Terminator [22], which work on existing code bases. In our approach, instead of trying to prove properties about a large existing code base expressed in difficult-to-reason-about languages like C or C++, we rewrite the browser inside of a theorem prover. This provides much stronger reasoning capabilities.

4.3 Related Work for Formal Verification of Reactive System Implementations

Proof Automation and Reducing Proof Burden. A long line of work addresses proof automation and reducing proof burden, both in the context of proof assistants and automated theorem provers. The main difference between previous work and our approach in this space lies in the co-design of the DSL and the automation techniques, which allows us to completely eliminate the per-program manual proof burden for many properties in a class of applications.

In proof assistants, tactics allow programmers to encode automation strategies.

The Ynot [62, 20] and Bedrock [19] projects provide frameworks for proving arbitrary properties of general purpose imperative programs, and make extensive use of tactic libraries furnishing automation to handle common proof obligations that arise from programs written in the framework, *e.g.,* the powerful `sep` tactic to dispatch separation logic obligations. However, the tactics in these systems do not entirely eliminate the proof burden. In practice, the user must still manually construct formal Coq proofs for any novel properties of their program and register those lemmas as hints at appropriate program locations, *e.g.,* proving and telling `sep` to use a commutativity lemma concerning data structure operations. Thus, writing our benchmarks directly in Ynot or Bedrock would still require large amounts of per-program manual proof effort (requiring deep Coq expertise), despite their powerful tactics.

The recent work of Gonthier et al. [26] presents a clever and general way of using Coq's canonical structures to automate certain parts of Coq proofs in a way that is more robust than tactics. While this approach can be used to automate certain parts of proofs, it still requires deep programmer expertise in Coq.

Other approaches have used DSLs to gain leverage for proof automation in the context of compiler correctness, for example Cobalt [46], Rhodium [48], PEC [45], and XCert [79]. Besides targeting a different domain and different properties, REFLEX also distinguishes itself in two additional ways. First, REFLEX provides a mechanism for specifying not just a program, but also properties of that program, whereas in compiler correctness, the property being verified is fixed. Second, REFLEX achieves a guarantee that is far stronger than previous systems: all the proofs have been done in Coq, from the ground up, for each program written in REFLEX, while still providing pushbutton automation.

More generally, there has been a substantial amount of work in automatically verifying properties of programs using techniques such as abstract interpretation and

tools such as SMT solvers. However, this work often assumes, without a fully formal proof, that (1) queries to some automated solver (e.g. an SMT solver) are sufficient to prove the intended properties and (2) the particular solver (e.g. Z3) is correct. REFLEX avoids such gaps through end-to-end fully formal verification in Coq.

Formal Verification in Proof Assistants. In addition to work on reducing the proof burden, REFLEX is also related to a long line of work on using interactive proof assistants for formal verification. Most importantly, our Coq implementation of REFLEX makes heavy use of the Ynot library [62], which in turn builds on Hoare Type Theory [60]. Furthermore, the trace-based approach we use in our generated Coq code builds on the approach of Malecha et al. [55], which uses Ynot to implement and prove properties about servers. The Quark web browser [38] also uses Ynot to verify several trace-based properties, including tab non-interference. However, the Quark verification effort, as with all previous verification efforts in Coq, does not achieve nearly the level of pushbutton automation provided by REFLEX.

Privilege Separation. Our work also builds on the idea of privilege separation introduced by Provos et al. in their privilege separated OpenSSH [65]. In particular, the architecture we use for our web browser, SSH, and web server benchmarks follow the principles outlined by Provos et al. Privilege separation is also the conceptual linchpin behind the kernel-based architectures used in web browsers like Chrome [68], Gazelle [85], and OP [28]. However, none of this previous research on privilege separation used proof assistants like Coq to formally verify properties of these systems.

4.4 Acknowledgements

This chapter is adapted from material as it appears in Programming Language Design and Implementation 2010. Tatlock, Zachary; Lerner, Sorin, ACM Press, 2010. Usenix Security Symposium 2012. Jang, Dongseok; Tatlock, Zachary; Lerner, Sorin,

Usenix Association, 2012. Programming Language Design and Implementation 2014. Rickets, Daniel, Robert, Valentin, Jang, Dongseok; Tatlock, Zachary; Lerner, Sorin, ACM Press, 2014. The dissertation author was a primary investigator and author of these papers.

Chapter 5

Conclusion and Future Work

5.1 Future Work

My past projects have led to a broader, two-phase research strategy. In the first phase, I will focus on vital areas where software correctness is essential. Within these areas, I will identify *verification pinch points*, key components that, if properly implemented, guarantee important correctness properties for the entire system. Next I will develop techniques to formally verify these pinch points, and as a result extend the frontier of tractable verification problems. In the second phase, I will identify common patterns that occur while verifying such systems and expose them in new language constructs that enable non-expert programmers to build robust, formally verified applications without writing any proofs. My background in formal verification, static correctness checking, and domain specific languages (DSLs) will guide me as I carry out this research strategy and collaborate with colleagues to apply the resulting insights across domains.

Low-level, high performance cryptographic implementations serve as an example where I intend to apply my research strategy. Recent developments in proof techniques for cryptographic schemes have enabled practitioners to make strong formal guarantees regarding the security of their protocols [11]. However, these proofs are carried out on abstract, high-level models of the protocol, which leaves a disconcerting *formality*

gap between what is proven correct and the low-level, high performance code that is actually run (*e.g.,* GnuTLS). Thus, to ensure the trustworthiness of these crucial code bases, I propose investigating techniques to bridge the gap between such verified specifications and their low-level implementations. To achieve this goal, I will build on my background in compiler verification, which centers around reasoning about program semantics and equivalence between high- and low-level programs. This work will require new techniques to precisely reason about probabilistic program behavior for low-level code and also the aggressive optimizations found in cryptographic implementations.

Another application of my research strategy will explore verifying high-level synthesis (HLS) tools. These increasingly prevalent tools take programs written in a higher-level language like C and compile them to hardware description languages like VHDL and Verilog. Unfortunately, such translations are very subtle and error prone; I have already gathered a set of examples where HLS tools incorrectly generate hardware designs whose behavior differs from the behavior specified by the input. Such errors can be enormously expensive due to the high capital costs of hardware manufacturing and the difficulty of recalling or servicing deployed products. Thus, we would like to guarantee that design tools never introduce any errors as they translate high-level specifications down to low-level hardware descriptions. My background in compiler verification, which I have already applied in the context of HLS tools [77], will serve as a foundation as we address new challenges including the highly concurrent nature of hardware description languages like Verilog and bridging the semantic gaps between high-level languages and lower-level hardware specifications. This work will be informed by, connect, and extend the rich literature on both hardware and compiler correctness.

One advantage of my research strategy is that it naturally forms a pipeline between the first and second phases. While the above two projects concern the first phase of identifying and verifying pinch points, I also intend to develop new projects that extend

my past work and make its insights accessible even to non-expert programmers. In particular, the formal proof burden represents the greatest barrier to broader adoption of formal verification techniques. Thus, my ultimate goal is to enable non-expert programmers to build real systems which provide formally proven correctness guarantees without requiring the programmer to write any proofs. Currently, most developments contain several times more interactive proof code than real, runnable code. Worse, writing proof code is far more difficult than traditional programming and it requires a deep background in the logic and proof language of the theorem prover. As a veteran of several large formal verification projects, I have experienced first hand how quickly the proof burden can become unmanageable, keeping it in check requires significant, hard-earned experience. Building on my experience from the PEC, XCert, and QUARK projects, I intend to address these issues by developing tools that, within specific domains, ease or entirely eliminate the proof burden.

As a starting point, I will focus on eliminating the proof burden for kernel-based systems like QUARK. While my insights in formal shim verification made it possible to build and verify a real, modern web browser like QUARK, that work still required a significant amount of proof effort and verification expertise. However, during the course of the project, general patterns emerged for designing and formally verifying kernel-based systems. I will investigate techniques to expose these patterns in new language constructs which allow programmers to more easily build robust systems and verify their security properties without writing any proofs. This goal is motivated by the observation that many important security properties for such systems are highly structured, and thus more amenable to automation. Building on this initial research direction for kernel-based systems, I will then broaden the scope and investigate how to provide similar benefits for software systems in other domains.

Working on these projects will require close collaboration with colleagues to

ensure that we address key challenges and that the resulting solutions are accessible to the typical programmer in the domain. As a result, my work will simplify writing robust, provable secure programs within several critical domains.

Bibliography

[1] http://gallium.inria.fr/~dargaye/mlcompcert.html.

[2] Chrome security hall of fame. http://dev.chromium.org/Home/chromium-security/hall-of-fame.

[3] Public suffix list. http://publicsuffix.org/.

[4] Pwn2own. http://en.wikipedia.org/wiki/Pwn2Own.

[5] Devdatta Akhawe, Adam Barth, Peifung E. Lamy, John Mitchell, and Dawn Song. Towards a formal foundation of web security. In Michael Backes and Andrew Myers, editors, *Proceedings of CSF 2010*, pages 290–304. IEEE Computer Society, July 2010.

[6] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff Biffle, and Bennet Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *PLDI*, pages 355–366, 2011.

[7] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 2001.

[8] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *ACM Conference on Computer and Communications Security*, pages 75–88, 2008.

[9] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *To appear at the 15th ACM Conference on Computer and Communications Security (CCS 2008)*, 2008.

[10] Adam Barth, Collin Jackson, Charles Reis, and The Google Chrome Team. The security architecture of the Chromium browser. Technical report, Google, 2008.

[11] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pages 90–101. ACM, 2009.

[12] Nick Benton and Nicolas Tabareau. Compiling functional types to relational specifications for low level imperative code. In *TLDI*, 2009.

[13] Aaron Bohannon, Benjamin C. Pierce, Vilhelm Sjöberg, Stephanie Weirich, and Steve Zdancewic. Reactive noninterference. In *Proceedings of the 16th ACM conference on Computer and communications security*, 2009.

[14] Aaron Bohannon, Benjamin C. Pierce, Vilhelm Sjöberg, Stephanie Weirich, and Steve Zdancewic. Reactive noninterference. In *CCS*, 2009.

[15] Eric Yawei Chen, Jason Bau, Charles Reis, Adam Barth, and Collin Jackson. App isolation: get the security of multiple browsers with just one. In *Proceedings of the 18th ACM conference on Computer and communications security*, 2011.

[16] Shuo Chen, Jos Meseguer, Ralf Sasse, Helen J. Wang, and Yi min Wang. A systematic approach to uncover security flaws in GUI logic. In *IEEE Symposium on Security and Privacy*, 2007.

[17] Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *PLDI*, 2007.

[18] Adam Chlipala. A verified compiler for an impure functional language. In *POPL*, 2010.

[19] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, 2011.

[20] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP*, 2009.

[21] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *PLDI*, 2009.

[22] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond safety. In *CAV*, 2006.

[23] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI*, 2002.

[24] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[25] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[26] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc. In *ICFP*, 2006.

[27] Chris Grier, Shuo Tang, and Samuel T. King. Secure web browsing with the OP web browser. In *IEEE Symposium on Security and Privacy*, 2008.

[28] Chris Grier, Shuo Tang, and Samuel T. King. Secure web browsing with the OP web browser. In *IEEE Symposium on Security and Privacy*, 2008.

[29] Samuel Z. Guyer and Calvin Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of IEEE*, 93(2), 2005.

[30] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *POPL*, 2002.

[31] Jon Howell, Collin Jackson, Helen J. Wang, and Xiaofeng Fan. MashupOS: operating system abstractions for client mashups. In *HotOS*, 2007.

[32] Lin-Shung Huang, Zack Weinberg, Chris Evans, and Collin Jackson. Protecting browsers from cross-origin css attacks. In *ACM Conference on Computer and Communications Security*, pages 619–629, 2010.

[33] Collin Jackson and Adam Barth. Beware of finer-grained origins. In *In Web 2.0 Security and Privacy (W2SP 2008)*, May 2008.

[34] Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh. Protecting browsers from dns rebinding attacks. In *ACM Conference on Computer and Communications Security*, pages 421–431, 2007.

[35] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An empirical study of privacy-violating information flows in JavaScript Web applications. In *Proceedings of the ACM Conference Computer and Communications Security (CCS)*, 2010.

[36] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing browser security guarantees through formal shim verification. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 8–8, Berkeley, CA, USA, 2012. USENIX Association.

[37] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing browser security guarantees through formal shim verification. Technical report, UC San Diego, 2012.

[38] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing browser security guarantees through formal shim verification. In *USENIX Security*, 2012.

[39] Dongseok Jang, Aishwarya Venkataraman, G. Michael Sawka, and Hovav Shacham. Analyzing the cross-domain policies of flash applications. In *In Web 2.0 Security and Privacy (W2SP 2011)*, May 2011.

[40] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW*, pages 601–610, 2007.

[41] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *SOSP*, 2009.

[42] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, et al. seL4: formal verification of an OS kernel. In *SOSP*, 2009.

[43] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, et al. Experimental security analysis of a modern automobile. In *IEEE Symposium on Security and Privacy (SP)*, 2010.

[44] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. In *PLDI*, 2009.

[45] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. In *PLDI*, 2009.

[46] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *PLDI*, 2003.

[47] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL*, 2005.

[48] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL*, 2005.

[49] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL*, 2006.

[50] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL*, 2006.

[51] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *PLDI*, 2006.

[52] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *POPL*, 2010.

[53] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *POPL*, 2010.

[54] Gregory Malecha, Greg Morrisett, and Ryan Wisnesky. Trace-based verification of imperative programs with I/O. *J. Symb. Comput.*, 46:95–118, February 2011.

[55] Gregory Malecha, Greg Morrisett, and Ryan Wisnesky. Trace-based verification of imperative programs with I/O. In *Journal of Symbolic Computation*, 2011.

[56] Daryl McCullough. Noninterference and the composability of security properties. In *Security and Privacy*, 1988.

[57] James Mickens and Mohan Dhawan. Atlantis: robust, extensible execution environments for web applications. In *SOSP*, pages 217–231, 2011.

[58] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. Rocksalt: Better, faster, stronger sfi for the x86. In *PLDI*, 2012.

[59] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in Hoare type theory. In *ICFP*, 2006.

[60] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in Hoare type theory. In *ICFP*, 2006.

[61] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent types for imperative programs. In *ICFP*, 2008.

[62] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: dependent types for imperative programs. In *ICFP*, 2008.

[63] George C. Necula. Translation validation for an optimizing compiler. In *PLDI*, 2000.

[64] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *TACAS*, 1998.

[65] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *USENIX Security*, 2003.

[66] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*. USENIX Association, 2003.

[67] Paruj Ratanaworabhan, V. Benjamin Livshits, and Benjamin G. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *USENIX Security Symposium*, pages 169–186, 2009.

[68] Charles Reis, Adam Barth, and Carlos Pizano. Browser security: lessons from Google Chrome. In *CACM*, 2009.

[69] Martin Rinard and Darko Marinov. Credible compilation with pointers. In *Workshop on Run-Time Result Verification*, 1999.

[70] Jesse Ruderman. The same origin policy, 2001. http://www.mozilla.org/projects/security/ components/same-origin.html.

[71] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, , and Jade Alglave. The semantics of x86-cc multiprocessor machine code. In *POPL*, 2009.

[72] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *IEEE Symposium on Security and Privacy*, pages 513–528, 2010.

[73] Kapil Singh, Alexander Moshchuk, Helen J. Wang, and Wenke Lee. On the incoherencies in web browser access control policies. In *IEEE Symposium on Security and Privacy*, pages 463–478, 2010.

[74] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 921–930, 2010.

[75] Shuo Tang, Haohui Mai, and Samuel T. King. Trust and protection in the illinois browser operating system. In *OSDI*, pages 17–32, 2010.

[76] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *POPL*, January 2009.

[77] Zachary Tatlock. Parameterized program equivalence checking. In Sudipta Kundu, Sorin Lerner, and Rajesh K. Gupta, editors, *High-Level Verification: Methods and Tools for Verification of System-Level Designs*, chapter 8, pages 123 – 145. Springer, 2011.

[78] Zachary Tatlock and Sorin Lerner. Bringing extensibility to verified compilers. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 111–121, New York, NY, USA, 2010. ACM.

[79] Zachary Tatlock and Sorin Lerner. Bringing extensibility to verified compilers. In *PLDI*, 2010.

[80] Zachary Tatlock, Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. Deep typechecking and refactoring. In *Proceedings of the 23rd ACM SIGPLAN*

conference on Object-oriented programming systems languages and applications, OOPSLA '08, pages 37–52, New York, NY, USA, 2008. ACM.

[81] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *POPL*, 2008.

[82] Jean-Baptiste Tristan and Xavier Leroy. Verified validation of lazy code motion. In *PLDI*, 2009.

[83] Jean-Baptiste Tristan and Xavier Leroy. A simple, verified validator for software pipelining. In *POPL*, 2010.

[84] Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, and Herman Venter. The multi-principal OS construction of the gazelle web browser. Technical Report MSR-TR-2009-16, MSR, 2009.

[85] Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, and Herman Venter. The multi-principal OS construction of the gazelle web browser. Technical Report MSR-TR-2009-16, MSR, 2009.

[86] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, November 1997.

[87] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 283–294, New York, NY, USA, 2011. ACM.

[88] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *PLDI*, 2011.

[89] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *PLDI*, 2011.

[90] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. Javascript instrumentation for browser security. In *POPL*, pages 237–249, 2007.

[91] Steve Zdancewic and Andrew C Myers. Observational determinism for concurrent program security. In *CSF Workshop*, 2003.

[92] Lenore Zuck, Amir Pnueli, Benjamin Goldberg, Clark Barrett, Yi Fang, and Ying Hu. Translation and run-time validation of loop transformations. *Form. Methods Syst. Des.*, 27(3):335–360, 2005.