Automating Formal Proofs for Reactive Systems

Daniel Ricketts* Valentin Robert* Dongseok Jang* Zachary Tatlock† Sorin Lerner*

*University of California, San Diego †University of Washington

{daricket,vrobert,d1jang,lerner}@cs.ucsd.edu, ztatlock@cs.washington.edu

Abstract

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.

Categories and Subject Descriptors F.3.1 [*Logics and meanings of programs*]: Mechanical verification; D.2.4 [*Software Engineering*]: Correctness proofs, formal methods

General Terms Languages, Verification

Keywords interactive proof assistants, proof automation, domain-specific languages, reactive systems, dependent types

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.

PLDI '14, June 09 - 11 2014, Edinburgh, United Kingdom.

ACM 978-1-4503-2784-8/14/06…$15.00.

http://dx.doi.org/10.1145/2594291.2594338

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 [12], operating systems [7], web servers [14], database systems [13], and web browsers [6]. 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. [22] demonstrate *experimentally* that the fully formally verified CompCert C compiler [12] 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 paper, 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).

Unlike general purpose verification frameworks, like Ynot [3, 17] and Bedrock [2], where the unbounded expressiveness of programs and properties make 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 ac-

cessible 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. [14].

As a case study, we applied REFLEX to build and automatically prove properties about *privilege separated systems* [18], 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 [19], (2) an SSH daemon which provides remote, secure terminal access for unmodified SSH clients in the style of [18] and (3) a web server providing file access subject to user authentication and an access control policy.

To summarize, this paper 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 realistic *reactive systems* (Sections 2, 3, and 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 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 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 7).

2. Overview

Figure 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.

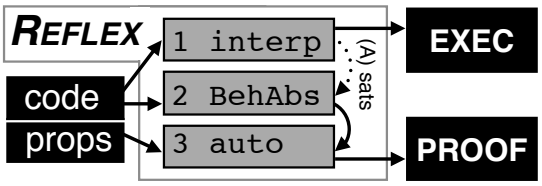


Figure 1: REFLEX Overview. REFLEX programs implement reactive systems and specify their properties. REFLEX provides (1) an interpreter to run programs, (2) a function BehAbs which, given a program, computes its behavioral abstraction and (3) Coq proof search tactics which attempt to formally guarantee the user-specified properties hold over a given program's behavioral abstraction. We proved once and for all that (A), any trace induced by running the interpreter on a program is included in that program's behavioral abstraction. When the user provides a program P, REFLEX generates an executable to run P using the interpreter.

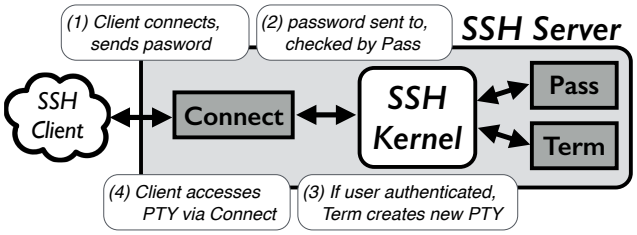


Figure 2: Simplified SSH Architecture. REFLEX enabled us to implement and formally verify an SSH server whose kernel orchestrates interactions between three component types, each restricted to just the capabilities they require: the **Password** component checks against the system password file to determine if user login attempts are successful, the **Terminal** component creates pseudo terminals upon request and passes back the resulting file descriptor, and the **Connection** component manages incoming requests from the network. Among other properties, REFLEX is able to formally prove that the kernel never grants a Connection component access to a terminal for user U unless the Connection first authenticates as U.

Example: SSH Server. We illustrate REFLEX through a running example: the kernel of a privilege separated SSH server. Figure 2 shows the architecture of our SSH server, which closely follows the privilege separation approach of Provos et al. [18]. 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 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.

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 initialized 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 6, may spawn many `Tab` s.

Properties. The last part of Figure 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 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, 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 paper 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: Simplified SSH Kernel in REFLEX DSL. The user specifies the types of components that constitute the system, the messages those components can exchange with the kernel, and how the kernel should respond when a component of a given type sends a particular type of message. Finally, the user specifies properties that should hold on every execution of the system, in this case requiring a user to authenticate before being granted access to a login terminal.

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.

3.1 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 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 implementa-

```
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 4: REFLEX Interpreter.

tion. 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.

3.2 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 [17] 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 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" [14]. 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 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 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 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 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 BehAbs(P). Second, *program dependent* verification is performed automatically by REFLEX tactics (described in Section 5), and proves that a given program P's user-specified properties hold on all possible execution traces included in BehAbs(P). Following LAC guided our design of REFLEX to exhibit this clean separation (discussed in Section 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.

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 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 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.

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 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`.

```
Components :
  Engine "engine.c"
  Doors  "doors.c"
  Radio  "radio.c"

Init :
  E <= spawn(Engine)
  D <= spawn(Doors)
  R <= spawn(Radio)

Properties :
  NoInterfere
    [Engine] [E]

Messages :
  Crash()
  Accelerating()
  DoorsM(string)
  Volume(string)

Handlers :
  Engine=>Crash():
    send(D, DoorsM("unlock"))
  Engine=>Accelerating():
    send(R, Volume("crank it up"))
  Doors=>DoorsM(s):
    if s == "open":
      send(R, Volume("mute"))
```

Figure 5: Simplified REFLEX Kernel for Car Controller. In this hypothetical automobile controller, the user wants to ensure that low-criticality Doors and Radio components do not interfere with the high-criticality Engine component. Intuitively, component C_ℓ *interferes* with component C_h, if C_h can receive any messages from the kernel which depend on the kernel's interaction with C_ℓ. In this example, interaction between components corresponds to both safety critical messages, e.g. ensuring that after a crash the doors unlock, and convenience messages, e.g. advising the radio to increase volume while the vehicle is accelerating.

```
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).
```

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 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.

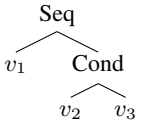
To address this issue, we adapt ideas on Reactive Non-Interference from [1]. As in the traditional setting, we specify non-interference by partitioning components into high and low sets. We would now like to define non-interference to hold iff, for any two executions where the kernel receives the same, possibly infinite, *sequence* of inputs from high components, it sends the same, possibly infinite, *sequence* of outputs to the high components. Unfor-

tunately, simply adapting ideas from [1] 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 [1]. 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 [15]. 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 [23]; 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.

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.

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 the relevant branch conditions cannot be satisfied without also satisfying the 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 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.

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 *h-vals*(θ_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 $\mathrm{NI}_{inv}(\theta_c, \theta_v, s, s')$ 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*

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

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 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:*

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

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

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 $\mathrm{Agree}(s_1, s_2)$ 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.

5.3 Incompleteness of REFLEX Automation

While our tactics provide effective automation for many important properties in large, realistic examples (shown in Section 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.

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.

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

Table 1: REFLEX benchmarks and their sizes (lines of code). Each benchmark is fully functional and consists of a verified REFLEX kernel with safety properties and surrounding sandboxed components built on existing infrastructures like WebKit for the browser and OpenSSH for the SSH server. The surrounding, sandboxed, components are unverified and written in C, C++, or Python.

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 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 [6] browser kernel, which mediates access between different components of 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 6), 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. [8] 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.

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 6 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.

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.

6.4 REFLEX Performance

As shown in Figure 6 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.

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 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

	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

Figure 6: Benchmark Properties. The time column comprises both the proof search and the proof term type-checking. The quark variants explore implementation trade-offs for handling cookies using cookie processes, and stress the robustness and performance of the automation. The ssh_2 variant uses a separate component to count authentication attempts. All properties are proved fully automatically using our tactics.

to the development effort required in [6], 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.

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 handcrafted 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.

8. Related Work

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 [3, 17] and Bedrock [2] 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. [4] 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 [10], Rhodium [11], PEC [9], and XCert [20]. 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 [17], which in turn builds on Hoare Type Theory [16]. Furthermore, the trace-based approach we use in our generated Coq code builds on the approach of Malecha et al. [14], which uses Ynot to implement and prove properties about servers. The Quark web browser [6] 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 [18]. 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 [19], Gazelle [21], and OP [5]. However, none of this previous research on privilege separation used proof assistants like Coq to formally verify properties of these systems.

9. 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.

Acknowledgments

We would like to thank the anonymous reviewers for helping us improve the paper. This work was supported in part by the National Science Foundation through grants 0964702, 1228967 and 1219172, and a generous gift from Google.

References

[1] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In *CCS*, 2009.

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

[3] A. Chlipala, G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP*, 2009.

[4] G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. In *ICFP*, 2006.

[5] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *IEEE Symposium on Security and Privacy*, 2008.

[6] D. Jang, Z. Tatlock, and S. Lerner. Establishing browser security guarantees through formal shim verification. In *USENIX Security*, 2012.

[7] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, et al. seL4: formal verification of an OS kernel. In *SOSP*, 2009.

[8] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, et al. Experimental security analysis of a modern automobile. In *IEEE Symposium on Security and Privacy (SP)*, 2010.

[9] S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *PLDI*, 2009.

[10] S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *PLDI*, 2003.

[11] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL*, 2005.

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

[13] G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. In *POPL*, 2010.

[14] G. Malecha, G. Morrisett, and R. Wisnesky. Trace-based verification of imperative programs with I/O. In *Journal of Symbolic Computation*, 2011.

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

[16] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare type theory. In *ICFP*, 2006.

[17] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: dependent types for imperative programs. In *ICFP*, 2008.

[18] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *USENIX Security*, 2003.

[19] C. Reis, A. Barth, and C. Pizano. Browser security: lessons from Google Chrome. In *CACM*, 2009.

[20] Z. Tatlock and S. Lerner. Bringing extensibility to verified compilers. In *PLDI*, 2010.

[21] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the gazelle web browser. Technical Report MSR-TR-2009-16, MSR, 2009.

[22] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI*, 2011.

[23] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *CSF Workshop*, 2003.