Establishing Browser Security Guarantees through Formal Shim Verification

Dongseok Jang
UC San Diego

Zachary Tatlock
UC San Diego

Sorin Lerner
UC San Diego

Abstract

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.

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, 42, 17, 41, 31] and defenses against specific attacks [26, 20, 22, 8, 36] to alternative security policies [25, 40, 21, 8, 39, 5] and improved JavaScript safety [14, 23, 38, 6, 44]. 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 [17, 13, 41, 12]) 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 paper 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 [17]. 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 paper is organized as follows. Section 2 provides background on browser security techniques and formal verification. Section 3 presents an overview of the QUARK browser. Section 4 details the design of the QUARK kernel and its implementation. Section 5 explains the tools and techniques we used to formally verify the implementation of the QUARK kernel. Section 6 evaluates QUARK along several dimensions while Section 7 discusses lessons learned from our endeavor.

2 Background and Related Work

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, 42, 17, 41, 31, 13, 12]. 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 [42], and OP [17] have been designed using *privilege separation* [35], 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 [42] 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 [17] 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 [27], Compilers [28, 1], Database Management Systems [29], Web Servers [30], and Sandboxes [32]. Some of these results have even *experimentally* been shown to to drastically improve software reliability: Yang et al. [43] 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 [34] 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

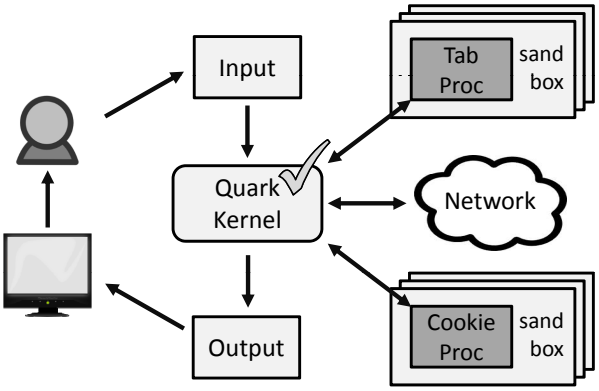

Figure 1: QUARK *Architecture.* This diagram shows how QUARK factors a modern browser into distinct components which run in separate processes; arrows indicate information flow. We guarantee our security properties by formally verifying the QUARK Kernel in the Coq proof assistant, which allows us to avoid reasoning about the intricate details of other components.

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 [33]. Specifically, Ynot enables *trace-based verification*, used extensively in [30] 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 [16], SLAM [7], BLAST [18] and Terminator [15], 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.

3 QUARK Architecture and Design

Figure 1 diagrams QUARK's architecture. Similar to Chrome [10] and OP [17], 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 paper, we assume that an attacker can compromise any QUARK component which is exposed to content from the Internet, except for the kernel which we 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).

3.1 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 where the kernel has complete control over the address bar. With this design, the graphical output process is never able to spoof the address bar.

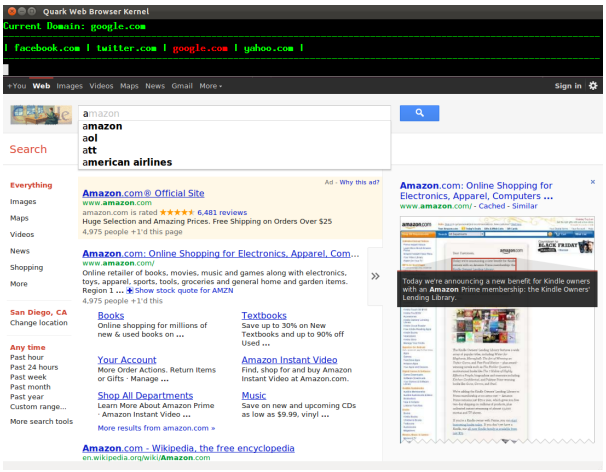

Figure 2: QUARK *Screenshot.* This screenshot shows QUARK running a Google search, including an interactive drop-down suggesting query completions and an initial set of search results from a JavaScript event handler dispatching an "instant search" as well as a page preview from a search result link. (Location blurred for double-blind review.)

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 "unprintable" 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.

3.2 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.

3.3 Efficiency

With a few simple optimizations, we achieve performance comparable to WebKit on average (see Section 6 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.

3.4 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 3.5). In fact, our address bar, located at the very top of the browser (see Figure 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.

3.5 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`).

3.6 Security Properties of QUARK

We provide intuitive descriptions of the security properties we proved for QUARK's kernel; formal definitions appear later in Section 4. 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 3.4).

4. **Cookie Integrity/Confidentiality**: The kernel disallows any cross-domain cookie stores or retrieves (as described in Section 3.5).

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.

4 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 [34]. 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 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 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 [24].

```
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 3: *Body for Main Kernel Loop.* This Coq code shows how our QUARK kernel receives and responds to requests from other browser components. It first uses a Unix-style select to choose a ready input channel, reads a request from that channel, and responds to the message appropriately. For example, if the user enters "+", the kernel creates a new tab and sends it the `Render` message. In each case, the code returns the new kernel state resulting from handling this request.

`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 `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 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 3.4, 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 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 3.5 and forwarding the message to the appropriate cookie process (note that for simplicity, we did not show the cookie processes in Figure 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 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`.

5 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.

5.1 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 [30].

```
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 4: *Traces and Actions.* This Coq code defines the type of externally visible actions our kernel can take. A *trace* is simply a list of such actions. We reason about our kernel by proving properties of the traces it can have. Traces are like other Coq values; in particular, we can write functions that return traces. `Read` is a helper function to construct a trace fragment corresponding to reading a single byte.

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 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 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`.

5.2 Kernel Specification

Figure 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

```
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 5: *Kernel Specification.* `step_correct` is a predicate over triples containing a past trace, a request trace, and a response trace; it holds when the response is valid for the given request in the context of the past trace. `tcorrect` defines a correct trace for our kernel to be a sequence of correct steps, *i.e.,* the concatenation of valid request and response trace fragments.

`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.

5.3 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 [24].

Following the approach of Malecha et al. [30], 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.

```
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 6: *Example Monadic Types.* This Coq code shows the monadic types for the `readn` primitive and for the `read_msg` function which is implemented in terms of `readn`. In both cases, the first expression between curly braces represents a pre-condition and the second represents a post-condition. The asterisk (`*`) may be read as normal conjunction in this context.

Thus `(traced tr)` can only be true for the current trace `tr`.

Figure 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.

5.4 Back to the Kernel

We now return to our kernel from Figure 3 and show how we prove that it satisfies the spec from Figure 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 runtime, 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 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 [24], but discuss proof effort in Section 6.

5.5 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 7 shows these key theorems, which correspond precisely to the security properties outlined in Section 3.6.

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 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 re-

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

sponse 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 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 pre-

cludes 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 [11] 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 [11], 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 [11] 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 [11] will be useful in achieving this goal. Unlike [11], 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, `no_xdom_sockets`, 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.

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 8: QUARK *Components by Language and Size.*

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.

6 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 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 paper. 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 [27], compiling QUARK's ML-like browser kernel with the MLCompCert compiler [1], and sandboxing other QUARK components with RockSalt [32], 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.

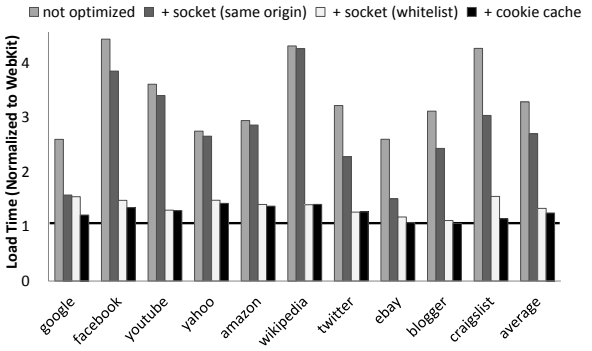

Figure 9: QUARK *Performance.* This graph shows QUARK load times for the Alexa Top 10 Web sites, normalized to stock WebKit's load times. In each group, the leftmost bar shows the unoptimized load time, the rightmost bar shows the load time in the final, optimized version of QUARK, and intermediate bars show how additional optimizations improve performance. Smaller is better.

Performance. We evaluate our approach's performance impact by comparing QUARK's load times to stock WebKit. Figure 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 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 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* [37] (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 [19] 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.

7 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 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 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.

8 Conclusions

In this paper, 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.

9 Acknowledgments

We thank Kirill Levchenko for many fruitful conversations regarding shim verification. We would also like to thank our shepherd, Anupam Datta, and the anonymous reviewers for helping us improve our paper.

References

[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] AKHAWE, D., BARTH, A., LAMY, P. E., MITCHELL, J., AND SONG, D. Towards a formal foundation of web security. In *Proceedings of CSF 2010* (July 2010), M. Backes and A. Myers, Eds., IEEE Computer Society, pp. 290–304.

[6] ANSEL, J., MARCHENKO, P., ERLINGSSON, Ú., TAYLOR, E., CHEN, B., SCHUFF, D. L., SEHR, D., BIFFLE, C., AND YEE, B. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *PLDI* (2011), pp. 355–366.

[7] BALL, T., MAJUMDAR, R., MILLSTEIN, T., AND RAJAMANI, S. K. 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] BARTH, A., JACKSON, C., AND MITCHELL, J. C. Robust defenses for cross-site request forgery. In *ACM Conference on Computer and Communications Security* (2008), pp. 75–88.

[9] BARTH, A., JACKSON, C., AND MITCHELL, J. C. Robust defenses for cross-site request forgery. In *To appear at the 15th ACM Conference on Computer and Communications Security (CCS 2008)* (2008).

[10] BARTH, A., JACKSON, C., REIS, C., AND THE GOOGLE CHROME TEAM. The security architecture of the Chromium browser. Tech. rep., Google, 2008.

[11] BOHANNON, A., PIERCE, B. C., SJÖBERG, V., WEIRICH, S., AND ZDANCEWIC, S. Reactive noninterference. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009).

[12] CHEN, E. Y., BAU, J., REIS, C., BARTH, A., AND JACKSON, C. App isolation: get the security of multiple browsers with just one. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011).

[13] CHEN, S., MESEGUER, J., SASSE, R., WANG, H. J., AND MIN WANG, Y. A systematic approach to uncover security flaws in GUI logic. In *IEEE Symposium on Security and Privacy* (2007).

[14] CHUGH, R., MEISTER, J. A., JHALA, R., AND LERNER, S. Staged information flow for javascript. In *PLDI* (2009).

[15] COOK, B., PODELSKI, A., AND RYBALCHENKO, A. Terminator: Beyond safety. In *CAV* (2006).

[16] DAS, M., LERNER, S., AND SEIGLE, M. ESP: Path-sensitive program verification in polynomial time. In *PLDI* (2002).

[17] GRIER, C., TANG, S., AND KING, S. T. Secure web browsing with the OP web browser. In *IEEE Symposium on Security and Privacy* (2008).

[18] HENZINGER, T. A., JHALA, R., MAJUMDAR, R., AND SUTRE, G. Lazy abstraction. In *POPL* (2002).

[19] HOWELL, J., JACKSON, C., WANG, H. J., AND FAN, X. MashupOS: operating system abstractions for client mashups. In *HotOS* (2007).

[20] HUANG, L.-S., WEINBERG, Z., EVANS, C., AND JACKSON, C. Protecting browsers from cross-origin css attacks. In *ACM Conference on Computer and Communications Security* (2010), pp. 619–629.

[21] JACKSON, C., AND BARTH, A. Beware of finer-grained origins. In *In Web 2.0 Security and Privacy (W2SP 2008)* (May 2008).

[22] JACKSON, C., BARTH, A., BORTZ, A., SHAO, W., AND BONEH, D. Protecting browsers from dns rebinding attacks. In *ACM Conference on Computer and Communications Security* (2007), pp. 421–431.

[23] JANG, D., JHALA, R., LERNER, S., AND SHACHAM, H. An empirical study of privacy-violating information flows in JavaScript Web applications. In *Proceedings of the ACM Conference Computer and Communications Security (CCS)* (2010).

[24] JANG, D., TATLOCK, Z., AND LERNER, S. Establishing browser security guarantees through formal shim verification. Tech. rep., UC San Diego, 2012.

[25] JANG, D., VENKATARAMAN, A., SAWKA, G. M., AND SHACHAM, H. Analyzing the cross-domain policies of flash applications. In *In Web 2.0 Security and Privacy (W2SP 2011)* (May 2011).

[26] JIM, T., SWAMY, N., AND HICKS, M. Defeating script injection attacks with browser-enforced embedded policies. In *WWW* (2007), pp. 601–610.

[27] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: formal verification of an OS kernel. In *SOSP* (2009).

[28] LEROY, X. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *PLDI* (2006).

[29] MALECHA, G., MORRISETT, G., SHINNAR, A., AND WISNESKY, R. Toward a verified relational database management system. In *POPL* (2010).

[30] MALECHA, G., MORRISETT, G., AND WISNESKY, R. Trace-based verification of imperative programs with I/O. *J. Symb. Comput. 46* (February 2011), 95–118.

[31] MICKENS, J., AND DHAWAN, M. Atlantis: robust, extensible execution environments for web applications. In *SOSP* (2011), pp. 217–231.

[32] MORRISETT, G., TAN, G., TASSAROTTI, J., TRISTAN, J.-B., AND GAN, E. Rocksalt: Better, faster, stronger sfi for the x86. In *PLDI* (2012).

[33] NANEVSKI, A., MORRISETT, G., AND BIRKEDAL, L. Polymorphism and separation in Hoare type theory. In *ICFP* (2006).

[34] NANEVSKI, A., MORRISETT, G., SHINNAR, A., GOVEREAU, P., AND BIRKEDAL, L. Ynot: Dependent types for imperative programs. In *ICFP* (2008).

[35] PROVOS, N., FRIEDL, M., AND HONEYMAN, P. Preventing privilege escalation. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12* (2003), USENIX Association.

[36] RATANAWORABHAN, P., LIVSHITS, V. B., AND ZORN, B. G. Nozzle: A defense against heap-spraying code injection attacks. In *USENIX Security Symposium* (2009), pp. 169–186.

[37] RUDERMAN, J. The same origin policy, 2001. http://www.mozilla.org/projects/security/ components/same-origin.html.

[38] SAXENA, P., AKHAWE, D., HANNA, S., MAO, F., MCCAMANT, S., AND SONG, D. A symbolic execution framework for javascript. In *IEEE Symposium on Security and Privacy* (2010), pp. 513–528.

[39] SINGH, K., MOSHCHUK, A., WANG, H. J., AND LEE, W. On the incoherencies in web browser access control policies. In *IEEE Symposium on Security and Privacy* (2010), pp. 463–478.

[40] STAMM, S., STERNE, B., AND MARKHAM, G. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web* (2010), WWW '10, pp. 921–930.

[41] TANG, S., MAI, H., AND KING, S. T. Trust and protection in the illinois browser operating system. In *OSDI* (2010), pp. 17–32.

[42] WANG, H. J., GRIER, C., MOSHCHUK, A., KING, S. T., CHOUDHURY, P., AND VENTER, H. The multi-principal OS construction of the gazelle web browser. Tech. Rep. MSR-TR-2009-16, MSR, 2009.

[43] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. Finding and understanding bugs in C compilers. In *PLDI* (2011).

[44] YU, D., CHANDER, A., ISLAM, N., AND SERIKOV, I. Javascript instrumentation for browser security. In *POPL* (2007), pp. 237–249.

```
Definition kstep:
   forall (s: kstate),
   ST kstate {kcorrect s} {fun s' => kcorrect s'}.
Proof.
 intros; destruct s as [ctab tabs {tr}]; refine (
  s <- (iselect stdin tabs {tr});
    { tr := ISelect stdin tabs :: tr }
  match s with
  | Stdin =>
     c <- (read stdin {tr});
        { tr := Read c stdin :: tr }
     match c with
     | "+" =>
         t <- (mktab {tr});
            { tr := MkTab t :: tr }
         write_msg t Render {tr};
            { tr := WroteMsg t Render ++ tr }
         return (kstate t (t::tabs) {tr})
     | ...
     end
  | Tab t =>
     m <- (read_msg t {tr});
        { let tr := ReadMsg t m ++ tr }
     match m with
     | GetSoc host port =>
        if (safe_soc host (domain_suffix t)) then
          send_soc t host port;
             { tr := SentSoc t host port ++ tr }
          return (kstate ctab tabs {tr})
        else
          write_msg t (Error);
             { tr := WroteMsg t Error ++ tr}
          return (kstate ctab tabs {tr})
     | GetURL url =>
        data <- wget url;
           { let tr := Wget url data :: tr}
        write_msg t (ResURL data);
           { let tr := WroteMsg t (ResURL data) ++ tr }
        return (kstate ctab tabs {tr})
     | ...
     end
  end); sep fail auto.
  ...
Qed.
```

Figure 10: Kernel Single Step Code in Coq

A Formal Details

Here we present our browser kernel and its verification in greater detail. Figures 10 and 11 show a more nuanced view of our kernel implementation in Coq. For now, we will focus on the code of the kernel, which can be found in the `refine(...)` statements, and we will ignore anything in curly braces. The curly braces are notation we use for presentation purposes only – they contain static Hoare-predicates about run-time traces, which we will later use for proving correctness.

As mentioned previously, we use Ynot monads to encode our kernel in Coq's pure and terminating function language. Figures 10 and 11 use Ynot's monadic type `ST`. Although the name `ST` comes from the Ynot lineage, here we use `ST` only for I/O, not for mutable data structures (in contrast to [30] which uses it for both). Since for now we are ignoring curly braces, the type constructor `ST` takes only one parameter: `ST T` represents a computation which may perform some I/O, after which it returns a value of type `T`. The two parameters in curly braces are pre and post conditions which we will use in the verification.

A.1 Kernel Code

As discussed earlier, the kernel is a loop that continuously responds to requests from tabs or the user. The `kstep` function in Figure 10 performs one iteration of this loop by responding to a single request. The type `kstate` is a record type representing the functional state of the kernel in Coq; it contains a field `tabs` for the list of tabs, and a field `ctab` for the currently selected tab. The `kstep` function has the type `forall (s: kstate), ST kstate.` This means that it takes a Coq kernel state and returns a monad, which when executed performs some I/O and returns a new kernel state.

Before execution of `kstep` begins, the kernel state `s` is "destructed" into its components (this happens outside of the `refine` statement): `ctab` is bound to the currently selected tab, and `tabs` is bound to the list of tabs (recall that we are ignoring curly braces, so we don't need to worry about `tr` for now). Once this is done, the `kstep` function follows pretty much the same steps as in Figure 3.

The main kernel loop is described in Figure 11 using the `kloop` function, which essentially calls `kstep` and then calls itself. The code is slightly complicated by the fact that we are using Ynot's `fix` primitive. The `main` function, which is the kernel's entry point, starts a new tab and enters the kernel loop by calling `kloop`.

A.2 Kernel Verification

We now give more details on how the verification works. Recall from Section 5.3 that the type `ST T P Q` represents a computation which, if started in a state where `P` holds, will either diverge or will perform some side-effecting, I/O-inducing operations and return a value of type `T` in a state where `Q` holds. We use the version of monads from Ynot that are based on separation logic, so that the pre- and post-conditions are separation logic predicates, whose type in Ynot is `hprop`. `P` is simply an `hprop` and `Q` is a function from a return value of type `T` to `hprop` (so that the post-condition can talk about the return value of the monad).

The 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.

```
Definition kloop:
  forall (s: kstate),
  ST kstate {kcorrect s} {fun s' => kcorrect s'}.
Proof.
  intros; refine (
    fix {fun s => kcorrect s} {fun s s' => kcorrect s'}
        (fun self s =>   s <- kstep s;
                           { tr := s.tr }
                         s <- self s;
                           { tr := s.tr }
                         return s)
        s
  ); sep fail auto.
Qed.

Definition main:
  ST kstate {traced nil} {fun s => kcorrect s}.
Proof.
  intros; refine (
      { tr := nil }
    t <- (mktab {tr});
      { tr := MkTab t :: nil }
    write_msg t Render;
      { tr := WroteMsg t Render ++ tr}
    s <- (kloop (kstate t (t :: nil)) {tr})
      { tr := s.tr }
    return s
  );
  sep fail auto.
Qed.
```

Figure 11: Kernel Loop Code in Coq

If post-conditions don't refer to the return value, then the type of `bind` is as follows:

```
Axiom bind: forall T1 T2,
            ST T1 P1 Q1 ->
            (forall v: T1, ST T2 P2 Q2) ->
            (Q1 ==> P2) ->
            ST T2 P1 Q2
```

This is the same type as a traditional monad, except that there is one more parameter of type `(Q1 ==> P2)`, which represents the proof that `Q1` implies `P2`. Note that `==>` here is *not* regular implication `->` since the predicates involved are not Coq `Prop`s, but rather separation logic predicates; indeed, `==>` desugars into a relation defined in Ynot that represents separation logic implication, for which there are Lemmas and tactics that enable us to effectively reason with `==>`. The full type of `bind` is actually even more complex, because it needs to account for post-conditions that depend on the return value. Details can be found in [34].

To account for the additional parameter to `bind`, the syntactic sugar for `x <- a;b` is updated so that it passes the wildcard "_" for the proof `(Q1 ==> P2)`. The effect this has in Coq is that after the `refine` statement has been processed, Coq will enter into an interactive mode that allows the user to construct values for the missing "_". This amounts to proving all the implications originating from `bind` in Coq's interactive proof environment.

To show the intermediate predicates at various points in the computation, Figures 10 and 11 use a special notation for presentation purposes. In particular, after each monad invocation, we use the notation `{ tr := ... }` to capture the post-condition of that monad; the semantics of the annotation is: (1) we are creating a new binding for `tr` so that `tr` now captures the trace so far (2) `(traced tr)` holds at this point.

As mentioned in Section 5.3, Coq will ask us to prove implications between the post-condition of one monad and the pre-condition of the next. The Coq scripts that perform these proofs are written immediately after the `refine` statement. All proofs begin by invoking the `sep` tactic from Ynot, which for `main` and `kloop` discharges all the obligations. For `kstep`, however, there are numerous cases remaining, and this is where much of the proof effort was spent.

A.3 Additional Challenges

There are a variety of additional challenges that we needed to address in our Coq implementation and proof. These additional complexities make the development much more difficult and nuanced than presented here. In the following we provide intuition for a few of these additional challenges.

Separation Logic Implication Recall from the type of `bind` that the proof connecting the post-condition of one monad to the pre-condition of the next uses separation logic implication `==>`, rather than regular implication. Some of these proofs are automatically discharged by the `sep` tactic, but others are not. These proofs must be crafted in a different way than regular implication proofs, since hypothesis cannot be freely copied or cleared away. Instead, many obligations require tediously permuting hypotheses or the conclusion in order line up the various components so that the proof can go through. To handle this challenge, we defined some key invariants as a pure predicate and make them available throughout the pre- and post- conditions of the kernel. By defining a few key lemmas, we were easily able to dispatch the associated proof overhead of propagating these facts throughout the kernel.

Unbounded Separating Conjunction In some cases we need to capture a separating conjunction where the number of conjuncts is not statically know. For example, in `kcorrect` we need to state that the channels of all tabs are open – yet the list of tabs is not known statically. To express this invariant for an arbitrary number of tabs,

we define the following recursive function to create the appropriate separation logic predicate:

```
Fixpoint tabs_open (tabs: list tab) : hprop :=
  match tabs with
  | t::ts => open (tchan t) * tabs_open ts
  | nil   => emp
  end.
```

These kinds of conjunctions are not handled by the standard separation logic tactics in Ynot, which required us explicitly reason about such predicates using inductive arguments.

Frame Predicates The frame rule in Ynot's separation logic allows predicates that are unaffected by a monad to be preserved across that monad. In some cases, Coq is able to automatically infer what the frame predicate is, but in many cases, the frame predicate has to be explicitly given, which increases the annotation burden when writing the implementation code, and also adds additional complexity to the proofs. The additional complexity is particularly daunting when frame predicates interact with unbounded separating conjunction.

Consider for example the commonly occuring case where we know that `all_tabs_open` holds, and we want to invoke a monad such as `read_msg`, which in the pre/post conditions only mentions `open (tchan t)` for a *single* tab `t`. Due to the preservation of separation logic conjuncts, the fact that `t` is open gets "used up" to establish the pre-condition of `read_msg`; the post-condition then re-establishes the fact. This means that the frame predicate now needs to state that all tabs *aside* from `t` are open. We do this by defining a predicate `(tabs_open_x tabs x)` which says that all tabs except `x` are open (the predicate is similar to `tabs_open` above).

We then need a way to extract out of `tabs_open` the tab `t` on which we want to call `read_msg`, and we need a way to place `t` back into `tabs_open` after `read_msg`:

```
Lemma unpack_tabs_open: forall ts t, In t ts ->
  (tabs_open ts ==> open (tchan t) * tabs_open_x ts t).
Lemma repack_tabs_open: forall ts t, In t ts ->
  (open (tchan t) * tabs_open_x ts t ==> tabs_open ts).
```

Note again that `==>` here is *not* implication `->` since the predicates involved are not Coq `Prop`s, but separation logic predicates.

One character at a time We found it highly useful in the spec to encode every user command as a single ASCII character. This is because several functions "replay" the trace to define characteristics of the overall trace, e.g. `proj_user_control` from the first theorem in Figure 7. Such functions become tremendously more complicated if we allow multi-character commands from the user, since they would then have to look ahead several characters to uniquely determine what command the user is entering. In cases where the trace needs to be reasoned about in the middle of a user command, we are left in a messy state. Many proofs become simpler when we allow only "atomic" user actions. As mentioned before, we rely on "un-printable" ASCII characters to expand the set of "atomic" commands we can receive from the user.