Coq Framework for security policies and proof of concept application

Anders Kaseorg, Jason Gross, and Peng Wang

December 9, 2014

1 Problem

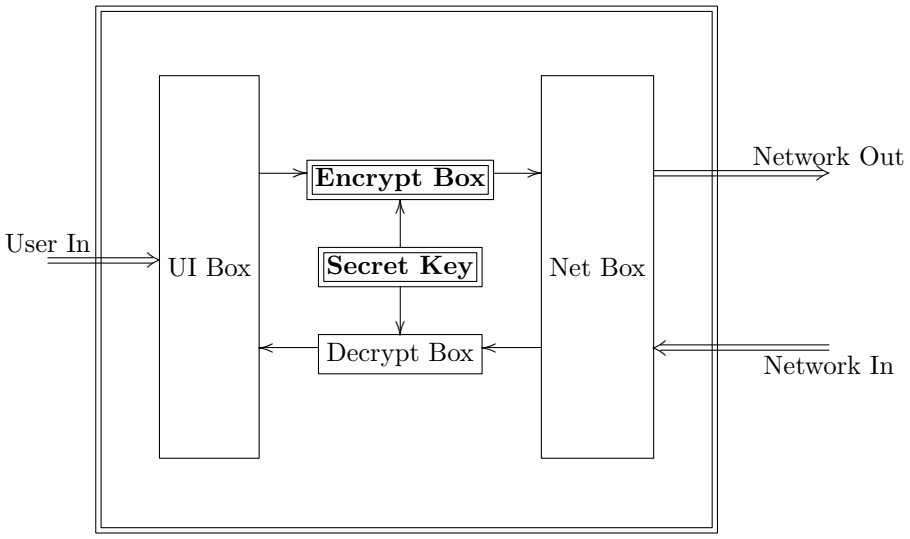
Current security policies, such as those on mobile platforms, are very coarse. We must grant applications permission "to use the internet" or "to use the camera" or "to read and write data on the phone". It would be nice to be able to talk about more fine-grained security policies. For example, we might want to allow apps to use the internet for ads, and simultaneously manage sensitive data, without having to worry about it sending sensitive data to the ad servers. We might want applications that have permission to sync data, but shouldn't be able to leak any of it; they might only have permission to transmit encrypted data.

2 Design

We have implemented a framework in the Coq proof assistant for allow modular development of code including trusted and untrusted components without the runtime costs and restrictions of sandboxing.

2.1 Ensuring Security: The Trusted Code Base

An application consists of some trusted boxes and some untrusted boxes, which communicate with each other via asynchronous message passing. The trusted code base consists of the code implementing the trusted boxes (for example, encryption algorithms), and a top-level "wiring diagram" which enforces modularity. For example, a password manager application might want to enforce the policy that the user's passwords cannot leak to the network unencrypted, and might want to enforce this restriction even in the presence of untrusted user-interface and network-communication implementations. Such an application might have the following simplified wiring diagram, with **trusted** and untrusted boxes:



2.2 Enforced Modularity and Parametricity

In the example above, the only path from user input to network output goes through the encryption box. Thus this application, by construction, can leak no unencrypted information from the user to the network via message content. By setting up the encryption box to send outgoing messages at given predefined times, we can also nearly prevent information leakage via timing side-channels (see subsection 3.2 for more details).

More precisely, this enforced modularity is a form of parametricity. Because the Net Box must be parametric in the input it receives from the Encrypt Box, and it must be parametric in the secret key (which it has no access to) it should not be able to deduce anything about the input to the Encrypt Box, and thus we don't need to trust its output. (Of course, we might insert other trusted components, such as one that prevents the Net Box from contacting most servers. Because these checks are inserted in the source code, an optimizing compiler could elide checks which are obviously true.)

Furthermore, the top-level wiring diagram is a kind of source-level sandboxing: All implementations are pure, and communicate with the outside world via OCaml shims mediated by the top-level wiring diagram. Therefore, if there is no wire connecting an untrusted box with a system call, that box cannot make use of that system call.[1]

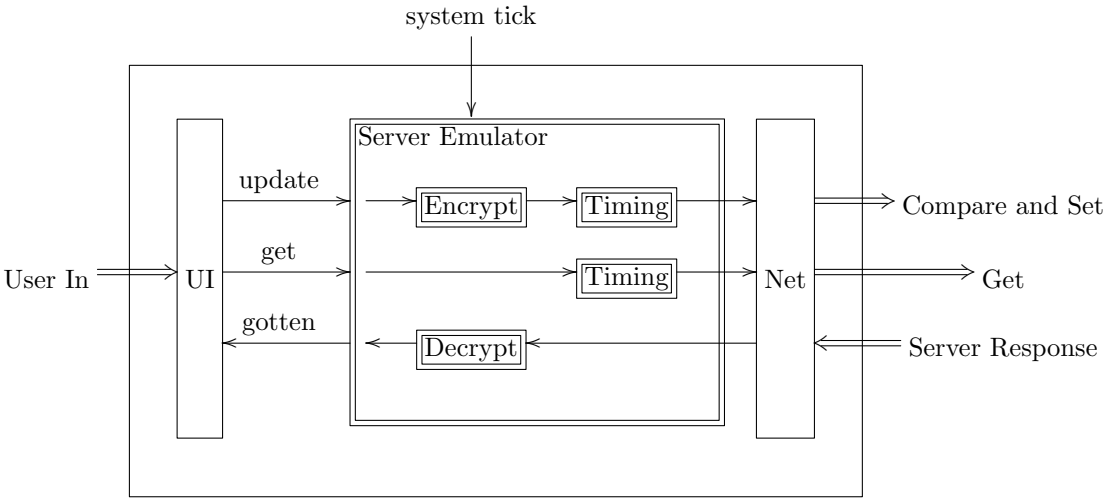
[1] This is technically not true in our current version. A malicious box can include a command like `Extract Constant evil ⇒ "Unix.system".` and thus force extraction of an innocuous function that breaks the sandbox. We plan to eventually bypass the extraction mechanism in favor of a verified compilation pipeline down to assembly, perhaps through Fiat, Facade, Cito, and Bedrock, which will not have this exploit.

3 High-Level Implementation

3.1 Proof of Concept: Password Manager Application

We implemented a proof-of-concept application: a password manager. The password manager accepts user-input of passwords associated to keys (e.g., websites), and synchronizes them with a remote server over https using a fixed key for identification (not yet verified by the server) and a fixed key for encryption (both currently hard-coded[2]). Timing side-channels are mostly avoided (again, see subsection 3.2 for more).

Unlike the diagram above, we have a single trusted box which handles encryption, decryption, and timing side-channels. It presents the interface of a simple mutable cell which handles "update" and "request value" events from the UI, and sends "got value" messages back to the UI. It interfaces with a remote server with a compare-and-set operation via the untrusted Net Box. Because the remote server uses compare-and-set (to ensure that we don't lose updates from different clients), we need to store our current view of the encrypted state of the server in trusted storage, because it is sent verbatim over the web:



This is only one possible implementation, which we found relatively simple to compose with conceptually clean interfaces. It is not the optimal implementation, but the point of the application is to provide a proof of concept that the framework is feasible, not be an optimally secure password manager in its current form. Another implementation could involve trusting only the encryption and timing-side-channel-avoidance boxes, and implementing the compare-and-set logic immediately before the net box. Yet another example implementation

[2]In production versions of this application, these would be read from a configuration file.

would have the trusted mutable cell accept decrypted text, and encrypt it in a timing-oblivious way, but present the UI with encrypted text and let it handle decryption.

3.2 Timing Attacks

An interesting side-effect of having all communication protocols be message-passing-based, and therefore asynchronous, is that it makes it rather simple to include defenses against timing side-channels in a modular way. Because every interface is asynchronous, it is a rather simple matter to tie external communication to a system clock rather than to the time that, say, encryption finishes. We implement a simple (trusted) Tick Box that does this; it handles notifications that the value has changed, and later requests the updated value (computing it is assumed to be a potentially expensive operation), and then later passes the updated value on, on it's own system-clock-based schedule.

Although our framework only allows asynchronous communication, we currently only support single-threaded applications. Therefore, a malicious UI Box could collude with a malicious Net Box to starve the Tick Box of computation cycles at the just the right time, leaking unencrypted information from the UI Box to the Net Box (and then out to the web). We currently don't defend against this attack, although we do ensure, given sufficient granularity of the system timer, that we can notice such things happening; our Tick Box will raise a warning when it does not get ticks frequently enough. Unfortunately, we currently don't manage a granularity of more than about 1 millisecond. Future versions of the framework will use a verified optimizing compiler down to assembly code, and will not suffer from this deficiency in the extracted OCaml code.

We hope that future versions of the framework will support timing proofs. By proving that the response of an untrusted box to a given message never exceeds a certain number of clock ticks, we can prove that timing side-channels are avoided, even in the presence of untrusted boxes. We currently only support proving termination of all responses to messages, but we plan to extend these theorems to ones about absolute timing.