FacetBook

Thomas Schmitz Cormac Flanagan

June 11, 2019

Abstract

This report describes FacetBook, a prototype social networking website that we built to ascertain the usefulness of the security library FIO, which implements (in Haskell) the Faceted Values technique for dynamic information flow control. We conduct our experiment by creating two versions of FacetBook: one that uses FIO and one that does not. We compare the two versions by counting the number of lines of code in the trusted computing base, which is the portion of code that must be carefully audited to ensure the absence of security bugs.

1 Research questions

An important goal for achieving security is to minimize the size of the *trusted computing base* (TCB), which is the portion of code that must be carefully audited for security [7]. (We refer to the remaining code as the *untrusted computing base* (UCB).)

Our hypothesis is that faceted execution (as implemented by the FIO library) makes it *easier* to minimize the size of the TCB in realistic applications. In particular, we have two research questions:

1. Does FIO help minimize TCB size when coding a secure application?
2. Does FIO help minimize TCB size when changing an existing application to meet new requirements?

Our experimental design to investigate these questions is as follows:

- Create Design V1 for a prototype application called FacetBook.
- Create Implementation V1-FIO using FIO, minimizing the TCB.
- Create Implementation V1-NoFIO without FIO, minimizing the TCB.
- Measure TCB size of the two implementations.
- Create Design V2 by making a small change to Design V1.
- Create Implementation V2-FIO by modifying V1-FIO.

Version	Lines of code			Total application-specific code
	FIO	TCB	UCB	
V1-FIO	108	99	352	451
V2-FIO	108	99	360	459
V1-NoFIO	0	118	295	413
V2-NoFIO	0	419	0	419
V2-NoFIO-minTCB	0	128	298	426

Table 1: The number of lines of code in each version of FacetBook. The emphasized entries are useful for quantifying security.

Version	Changes (measured in lines of code)			
	Modified	Moved	Inserted	Deleted
V1-FIO				
V2-FIO	1	0	8	0
V1-NoFIO				
V2-NoFIO	2	3	6	0
V2-NoFIO-minTCB	4	6	7	0

Table 2: The differences between each version of FacetBook. Each row in the table lists the differences from the version in the row above it. The emphasized entries are useful for quantifying *ease* of achieving security.

- Create Implementation V2-NoFIO by modifying V1-NoFIO.
- Create Implementation V2-NoFIO-minTCB from V2-NoFIO by minimizing the TCB.
- Quantify the effect on security by comparing the increase in TCB size when going from V1 to V2.
- Quantify the *ease* of achieving security by comparing the number of lines of code changed when minimizing the TCB size in V2.

Table 1 shows the number of lines of code for each version. Table 2 shows the number of edit actions required to change each version to the next. The full source code is available at `https://github.com/tommy-schmitz/facetbook`. In the sections below, we discuss these results.

2 Design V1: FacetBook

2.1 Overview

FacetBook is a prototype social networking website. Users can submit *posts* (pieces of text that are visible to a subset of other users of the website) and

can play *Tic Tac Toe* with other users, which is a simple and well-known game that children commonly play using pencil and paper. (In this case, the game is played using two computers equipped with web browsers and mouse pointer devices.)

For the purposes of our experiment, the "posts" feature exists so that FacetBook has a rich TCB (because the information flow requirements are complex), while the "Tic Tac Toe" feature exists so that it has a rich UCB (because the information flow requirements are simple, but the other computations are relatively complex).

2.2 User interface

Figure 1 illustrates the structure of FacetBook's webpages.

The `login` page allows typing a username and clicking the "Submit" button to go to the `dashboard` page. For simplicity, authentication always succeeds with no password required—sophisticated authentication machinery would remain constant throughout all six versions of FacetBook, and so would simply add a constant number of lines of code to the TCB. Unlike other work [3], we make no attempt here to remove authentication code (i.e. password-checking code) from the TCB.

The `dashboard` page shows a list of 20 recent posts created by users of FacetBook. The list comes from the server's database of all posts, but contains only those that the currently authenticated user is permitted to view. The page also has two links: one going to the `post` page and one to the `tictactoe` page.

The `post` page allows users to compose posts, and so has a form with two fields: the `permissions` field expects a space-delimited list of usernames indicating who is allowed to see the post, and the `content` field expects any string. Upon clicking "Submit," the form is submitted via HTTP POST protocol to the `/post` endpoint, and the server saves the submitted data in a database.

The `tictactoe` page initially shows a form with a single field `partner` expecting the username of the person with whom to play Tic Tac Toe. Upon clicking "Submit," the Tic Tac Toe board and its controls appear on the page. If this pair of users (the currently authenticated user and the specified `partner`) has never played Tic Tac Toe together before, then the server begins by adding a fresh game to the list of ongoing games in the database. Then the server retrieves the game (whether freshly-created or pre-existing) from the database and renders it into HTML when serving the page. Thereafter, if the user clicks on the controls of the game, then the web browser sends a request (using Javascript) specifying what action to take, and the server updates the game in the database as appropriate. Then the server replies with updated HTML, which replaces (using Javascript) the display in the browser.

2.3 Information security

In FacetBook, restricted information arrives via HTTP POST protocol at the `/post` endpoint. This endpoint is how users express their information flow

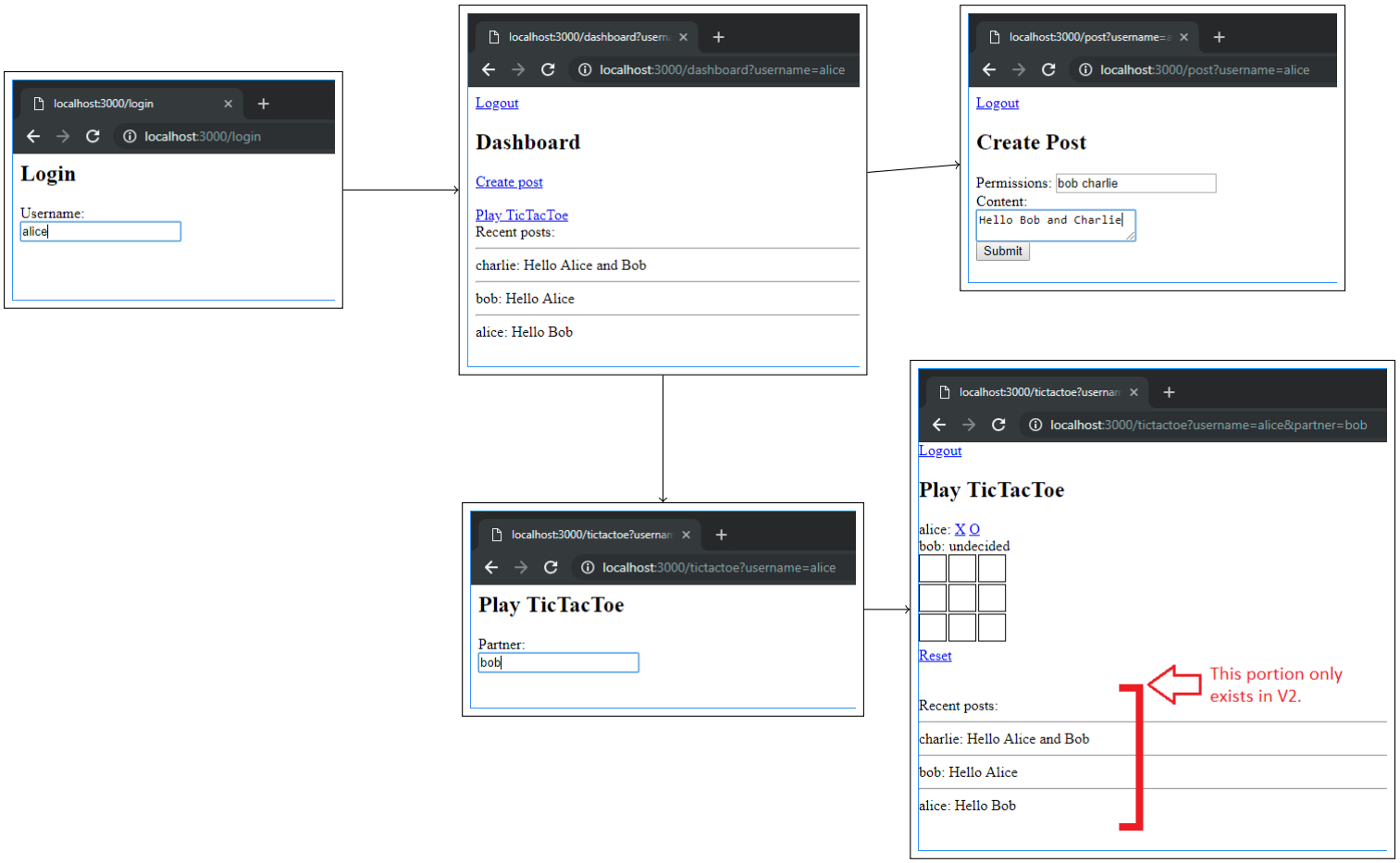

Figure 1: Screenshots of FacetBook.

desires, namely that only the users specified in the `permissions` field can know about this post and its content (in the `content` field).

The restricted output channel is the server's response to any incoming HTTP request—unless that request contains credentials of an appropriate user. In FacetBook, requests specify credentials in the HTTP GET parameter `username` (rather than in a cookie).

These information security specifications implicitly define a specific attacker model that considers some potential attacks and ignores others. Notably, our model ignores the correctness of the user interface, which is important because we intend to place the client code in the UCB. If an attacker controls the UCB, then the attacker could interfere with the creation of the POST request by, for instance, adding an extra entry to the `permissions` field before submitting the POST request. In the design of FacetBook, we explicitly ignore such an attack and choose instead to assume that the POST parameters received at the server correctly reflect the user's intentions.

3 FIO library

Figure 2 shows the interface of the FIO library. The main difference from previous work [6] is that this code now supports using an arbitrary security lattice [5], rather than specifically a power set security lattice over the set of `String`s. As a result, the type constructors `Fac`, `FIORef`, `FIO`, and `PC` now take an additional type parameter for specifying the security lattice. The corresponding `Lattice` type class (lines 1 through 4) specifies the methods (`leq`, `lub`, and `bot`) that the security lattice must implement.

In addition to the extra type parameter, we change slightly the representation of the `PC` datatype, so now `PC ks1 ks2` denotes the set of lattice elements k such that

- $k' \sqsubseteq k$ for all $k' \in$ `ks1`, and
- $k' \not\sqsubseteq k$ for all $k' \in$ `ks2`.

The main library function is `runFIO`, which runs an `FIO` computation safely, namely by respecting the information flow requirements specified by any faceted values used in the computation. The computation bifurcates if necessary.

The FIO library contains 108 lines of code. (Only the interface is shown in Figure 2.)

4 V1-FIO

FacetBook V1-FIO is the initial version of the code, which implements Design V1, uses the FIO library, and is organized so as to minimize the size of the TCB.

```
class Lattice a where
  leq :: a -> a -> Bool
  lub :: a -> a -> a
  bot :: a

data Fac l a where
  Undefined :: Fac l a
  Raw       :: a -> Fac l a
  Fac       :: l -> Fac l a -> Fac l a -> Fac l a
  BindFac   :: Fac l a -> (a -> Fac l b) -> Fac l b

data FIORef l a = FIORef (IORef (Fac l a))

data FIO l a where
  Return  :: a -> FIO l a
  BindFIO :: FIO l a -> (a -> FIO l b) -> FIO l b
  Swap    :: Fac l (FIO l a) -> FIO l (Fac l a)
  IO      :: l -> IO a -> FIO l a
  New     :: a -> FIO l (FIORef l a)
  Read    :: FIORef l a -> FIO l (Fac l a)
  Write   :: FIORef l a -> Fac l a -> FIO l ()

data PC l = PC [l] [l]

runFIO :: Lattice l => PC l -> FIO l a -> IO a
```

Figure 2: The interface of the FIO library in all versions of FacetBook.

```
data Label = Whitelist [User]
           | Bot
instance Lattice Label where
  leq Bot _ = True
  leq _ Bot = False
  leq (Whitelist us1) (Whitelist us2) =
    let subset xs ys = all (\x -> x `elem` ys) xs  in
    us2 `subset` us1
  lub Bot k = k
  lub k Bot = k
  lub (Whitelist us1) (Whitelist us2) =
    Whitelist (List.intersect us1 us2)
  bot = Bot
```

Figure 3: The code for the `Label` datatype in all versions of FacetBook.

```
type Post     = String
data FList a  = Nil
              | Cons a (Fac Label (FList a))
type PostList = FList Post
type Database = (FIORef Label PostList, FIORef Label [TicTacToe])
```

Figure 4: The code for the `FList` datatype and associated type definitions in V1-FIO.

4.1 Tour of TCB

4.1.1 Security lattice

The lattice of security labels is defined in Figure 3. The label `Bot` is for public data; the label `Whitelist` *users* is for data visible only to the users listed in the list *users*. The datatype `Label` forms a lattice, as evidenced by the type class instance `Lattice Label` and its three methods `leq`, `lub`, and `bot`.

4.1.2 Database format

The database format is defined in Figure 4. For simplicity, we keep the database in memory rather than on disk (unlike other work on using faceted values with databases [8, 2]). The `Database` type is a pair of two mutable references (`FIORef`s), one for holding the current list of posts and a second for holding the current list of ongoing Tic Tac Toe games. The `PostList` type makes use of a custom datatype `FList`, which is a singly-linked list datatype whose "next" pointer is always faceted. The `Post` type is simply an alias for Haskell's built-in `String` type.

```
44 main :: IO ()
45 main = do  --IO
46   database <- runFIO (Constraints [] []) $ do  --FIO
47     r1 <- New Nil
48     r2 <- New []
49     return (r1, r2)
50   let port = 3000
51   Warp.run port $ \request respond -> do  --IO
52     let (k1, k2) = policy request
53     let fio_respond = \x -> IO k2 $ do  --IO
54           respond x
55           return ()
56     let faceted_request = Fac k1 (Raw request) Undefined
57     runFIO (Constraints [] []) $
58         UCB.handle_request faceted_request database fio_respond
59     return ResponseReceived
```

Figure 5: The code for the `main` function in V1-FIO.

The faceted values in an `FList` potentially allow the "list" to be structured actually as a tree with branching factor 2. However, in practice, when appending to the list, each facet shares a suffix with the opposing facet, so in fact the structure in memory forms a directed acyclic graph whose size is linear in the total number of posts.

4.1.3 Main function

Figure 5 shows the `main` function. Its purpose is to start the web server and set up appropriate security sandboxes before handling each request.

Line 47 initializes the database with an empty list of posts, and line 48 initializes it with an empty list of Tic Tac Toe games. Line 51 creates a socket (using the Haskell library function `Warp.run`) for listening for incoming HTTP requests, which are handled by the code on lines 52 through 59. Line 58 calls `UCB.handle_request`, which is outside the TCB; however, its inputs (`database`, `faceted_request`, and `fio_respond`) are all faceted appropriately, and its side effects are sandboxed appropriately by `runFIO (Constraints [] [])` on line 57.

4.1.4 Policy function

The function `policy` (called on line 52) computes the appropriate labels to use in FacetBook. Its code is shown in Figure 6. We parse the request to determine its meaning, and then we return two labels: one for the confidentiality of the request, and one for the label of the output channel for returning an HTTP

```
60 policy :: WAI.Request -> (Label, Label)
61 policy request =
62   if WAI.pathInfo request == ["login"] then
63     (Bot, Bot)
64   else case check_credentials request of
65     Nothing ->
66       (Bot, Bot)
67     Just username -> case WAI.pathInfo request of
68       ["post"] ->
69         let permissions = get_parameter request "permissions"  in
70         let users = words permissions  in
71         if all valid_username users then
72           (Whitelist (username : users), Whitelist [username])
73         else
74           (Whitelist [username], Whitelist [username])
75       _ ->
76         (Bot, Whitelist [username])
```

Figure 6: The code for the `policy` function in V1-FIO.

response to the user.

Specifically, this policy assigns `Bot` for both labels (lines 63 and 66) when the user is not logged in, which is the case when requesting the login page (line 62) or when lacking credentials on any other page (line 65). When the user has valid credentials, the HTTP response label is `Whitelist [username]` (lines 72, 74, and 76), indicating that the response can contain private information belonging to the authenticated user. For most pages, the confidentiality label on the request is `Bot` (line 76), which means that the request itself carries no sensitive information; however, on the `"post"` page, the label `Whitelist (username : users)` (line 72) indicates that the request is visible only to the users named in the `permissions` parameter of the request (and the currently authenticated user too). This label ensures that when the submitted post is written to the database, it will be faceted appropriately. The label `Whitelist [username]` on line 74 is used in case a client sends a malformed request where the `permissions` parameter contains invalid entries.

4.1.5 Import statements

The TCB includes the import statements at the top of each file. Primarily, we must verify that the `UCB` module imports (Figure 7) do not include `FIO(runFIO, FIO(IO), Fac(Raw, Fac, Undefined, BindFac))`. As a result, these import statements are actually part of the TCB.

The import statements in the `TCB` and `Shared` modules are also in the TCB, naturally, and help auditors determine which standard libraries must be trusted.

```haskell
{-# LANGUAGE OverloadedStrings #-}
module UCB where
import qualified Data.List as List
import Data.Monoid((<>))
import Data.String(fromString)
import qualified Data.ByteString.Lazy.Char8 as ByteString(intercalate)
import Network.HTTP.Types.Status(status200, status404)
import qualified Network.Wai as WAI(Request, pathInfo, ResponseLBS)
import Shared
import FIO(FIO(FIO(Read, Write, Swap), Fac(), FIORef)
```

Figure 7: The import statements for the `UCB` module in V1-FIO.

```haskell
{-# LANGUAGE OverloadedStrings #-}
module Shared where
import Data.String(fromString)
import Data.ByteString.Char8(unpack)
import qualified Network.Wai as WAI(Request, queryString)
import qualified Data.List as List(intersect)
import FIO
```

Figure 8: The import statements for the `Shared` module in V1-FIO.

```haskell
{-# LANGUAGE OverloadedStrings #-}
module TCB where
import qualified Network.Wai.Handler.Warp as Warp(run)
import qualified Network.Wai as WAI(Request, pathInfo)
import Network.Wai.Internal(ResponseReceived(ResponseReceived))
import Shared
import FIO
import qualified UCB as UCB(handle_request)
```

Figure 9: The import statements for the `TCB` module in V1-FIO.

```
102 check_credentials :: WAI.Request -> Maybe User
103 check_credentials request =
104   let username = get_parameter request "username"  in
105   if valid_username username  then  Just username
106                               else  Nothing
107
108 get_parameter :: WAI.Request -> String -> String
109 get_parameter request key =
110   case lookup (fromString key) (WAI.queryString request) of
111     Just (Just value) -> unpack value
112     _                 -> ""
113
114 valid_username :: String -> Bool
115 valid_username s =
116   s /= ""  &&
117   all (\c -> (c>='0' && c<='9') ||
118              (c>='a' && c<='z') ||
119              (c>='A' && c<='Z') ||
120              c=='_'                ) s
```

Figure 10: The code for the helper functions in V1-FIO.

4.1.6 Helper functions

For completeness, we include the TCB's helper functions, which are shown in Figure 10. `check_credentials` is the password-checking function. It gets the username from the HTTP GET parameters. For simplicity, it always succeeds without any password. When no username is supplied, it returns `Nothing`, indicating invalid credentials. `get_parameter` extracts an HTTP GET parameter from a request. `valid_username` checks that a string is non-empty and contains only letters, numbers, and underscores.

4.1.7 Summary

In summary, the TCB of FacetBook V1-FIO contains 99 lines: 41 in `TCB.hs`, 48 in `Shared.hs`, and 10 import statements in `UCB.hs`.

4.2 Tour of UCB

4.2.1 Handle-request function

The entry point to the UCB is `handle_request`, called on line 58 in `main`. Figure 11 shows its code. Its purpose is to "unfacet" the request (i.e. bifurcate if necessary, using `Swap` to do so), and then defer to the helper function `parse_request` and its return value `handler` to do the actual processing. At

```
121 type Handler = Database -> (WAI.Response -> FIO ()) -> FIO ()
122 handle_request :: Fac Label WAI.Request -> Handler
123 handle_request faceted_request database respond = do  --FIO
124   Swap $ do  --Fac
125     request <- faceted_request
126     return $ do  --FIO
127       let handler = parse_request request
128       handler database respond
129   return ()
```

Figure 11: The code for the `handle_request` function in V1-FIO.

the call site (line 58 in `main`), the faceted request always has a specific shape, namely with `Undefined` in the low-security facet. As a result, the bifurcation at line 124 executes the high-security path like normal (with a changed PC), and then the low-security path is a no-op.

This code illustrates a typical interaction between the two monads `Fac` and `FIO`. Line 124 uses `Swap` to change the current monad from `FIO` to `Fac` to allow extracting `request` from `faceted_request` on line 125. Then line 126 uses `return` to change the current monad back from `Fac` to `FIO` to allow executing the action on line 128. By using two monads, we can delimit the scope of the bifurcation to be lines 125 to 128. The computations join back together at line 129.

4.2.2 Parse-request function

The `parse_request` function translates an incoming web request (of type `WAI.Request`, imported from Haskell's `WAI` library for web servers) into an appropriate action (of type `Handler`) to take in response to that request. Figure 12 shows its code. It duplicates some functionality (checking whether the request is for the "login" page, checking credentials, etc.) from the `policy` function in the TCB, so it would be reasonable to refactor the code to reduce redundancy. We decided against doing so because the function names `policy` and `parse_request` document their purposes well, whereas it is nontrivial to choose a good name for the newly created functions and intermediate datatypes in the refactored version; in any case, the amount of duplicated code is small.

4.2.3 Handler functions

The `parse_request` function delegates functionality to eight other functions called `Handler`s, namely:

- `login`: sends to the client a login page.
- `authentication_failed`: sends a page to redirect back to the login page.

```
130 parse_request :: WAI.Request -> Handler
131 parse_request request =
132   if WAI.pathInfo request == ["login"] then
133     login
134   else case check_credentials request of
135     Nothing ->
136       authentication_failed
137     Just username -> case WAI.pathInfo request of
138       ["post"] ->
139         let content = get_parameter request "content"  in
140         let permissions = get_parameter request "permissions"  in
141         let users = words permissions  in
142         if content /= "" && all valid_username users then
143           do_create_post username content users
144         else
145           compose_post username
146       ["dashboard"] ->
147         dashboard username
148       ["tictactoe"] ->
149         let partner = get_parameter request "partner"  in
150         if valid_username partner then
151           let action = get_parameter request "action"  in
152           tictactoe_play username partner action
153         else
154           tictactoe_select_partner username
155       _ ->
156         not_found
```

Figure 12: The code for the `parse_request` function in V1-FIO.

- `do_create_post` *username content users*: inserts a new post into the database and redirects to the dashboard page.
- `compose_post` *username*: sends to the client a page displaying a form in which the user can compose a new post.
- `dashboard` *username*: sends a page displaying a few links to other pages, as well as a list of recent posts.
- `tictactoe_play` *username partner action*: updates a Tic Tac Toe game in the database (if necessary) and sends to the client a page displaying the current state of the game.
- `tictactoe_select_partner` *username*: sends to the client a page prompting the user to type the name of another user.
- `not_found`: sends a page with "404 bad request" on it.

The `Handler` type is defined on line 121

```
type Handler = Database -> (WAI.Response -> FIO ()) -> FIO ()
```

and its definition means that it takes as input the database reference cells (type `Database` defined on line 43) and a callback function (of type `WAI.Response -> FIO ()`) whose behavior when called is to send an HTTP response to the user's web browser. Thanks to the code in `main`, the database contents are secure (inside `FIORef`s) and the response callback function will not work if the current control flow has been influenced by information that the user should not know (in that case, the callback would behave as a no-op).

4.2.4 Summary

The UCB of FacetBook V1-FIO contains 352 lines: 362 in `UCB.hs` minus the 10 import statements at the top of the file, which are actually part of the TCB.

5 V1-NoFIO

FacetBook V1-NoFIO is the next version of the code, which implements Design V1, does not use the FIO library, and is organized so as to minimize the size of the TCB. In this section, we highlight the differences between V1-FIO and V1-NoFIO.

5.1 Removing undesirable dependence on FIO

The FIO library is unnecessary in this version of FacetBook, so we can simplify the code by removing dependence on FIO.

First, and most obviously, we remove the file `FIO.hs` from the codebase. As a result, we remove all calls to `Swap`, which is now unnecessary due to the lack of

```
158 main :: IO ()
159 main = do  --IO
160   r1 <- newIORef []
161   r2 <- newIORef []
162   let database = (r1, r2)
163   let port = 3000
164   Warp.run port $ \request respond -> do  --IO
165     let unit_respond = \x -> do  --IO
166           respond x
167           return ()
168     handle_request request database unit_respond
169     return ResponseReceived
```

Figure 13: The code for the `main` function in V1-NoFIO.

faceted values. Similarly, we replace uses of `New`, `Read`, and `Write` with uses of `newIORef`, `readIORef`, and `writeIORef`, respectively. Continuing likewise, we remove the `FList` datatype (which uses faceted values) and update the `PostList` type definition:

```
157 type PostList = [(Label, Post)]
```

These simple changes affect the line count very little (aside from removing the 108-line FIO library).

5.2 Removing desirable dependence on FIO

Next, we completely remove the `policy` function and the lines in `main` that depend on it. Figure 13 shows the new `main` function. At this point, the functionality of FacetBook is intact, but its security guarantees have disappeared—in particular, all posts are now visible to all users, regardless of any permission settings on any posts. To reimplement this security feature, we define a new function `filter_posts`:

```
170 filter_posts :: Label -> PostList -> PostList
171 filter_posts k = filter (\(k',p) -> leq k' k)
```

and we call it inside the `dashboard` function just after reading the posts from the database:

```
172 labeled_posts <- readIORef (fst database)
173 let posts = filter_posts (Whitelist [username]) labeled_posts
```

We must also add a line to the `do_create_post` function to label posts just before they are written into the database (line 175):

```
174 d <- readIORef (fst database)
175 let labeled_data = ( Whitelist (username : users) ,
176                        username ++ ": " ++ content  )
177 writeIORef (fst database) (labeled_data : d)
```

5.3 Minimizing the TCB

With only the changes mentioned so far, the file `UCB.hs` is poorly named because it now contains code that belongs in the TCB. To rectify this situation, we begin by moving four functions from `UCB.hs` to `TCB.hs`, namely `handle_request`, `parse_request`, `do_create_post`, and `dashboard`. Finally, to keep the TCB as small as possible, we must rewrite `parse_request` so that it uses sandboxing for the other six types of request (besides `do_create_post` and `dashboard`). The new code is in Figure 14. Line 179 defines the `sandbox` function, which simply arranges for the posts to be censored from the database before calling a given handler `h`. By calling it on lines 182, 185, 194, 201, 203, and 205, we avoid the need to move any more functions from `UCB.hs` to `TCB.hs`.

5.3.1 Summary

In V1-NoFIO, the TCB contains 118 lines of code: 63 in `TCB.hs`, 45 in `Shared.hs`, and 10 import statements in `UCB.hs`. The UCB contains 295 lines of code: 305 in `UCB.hs` minus the 10 import statements at the top of the file.

Qualitatively comparing V1-FIO to V1-NoFIO is largely subjective. The application-specific TCB is smaller in V1-FIO; on the other hand, since FIO is part of the TCB, the total TCB size is less in V1-NoFIO.

Furthermore, the TCB code is qualitatively different in the two implementations. In V1-FIO, the structure of the TCB (especially the `policy` function) relieves auditors from digging through the codebase to find and verify security-critical operations, such as filtering the list of posts before displaying it, and correctly labeling new posts before inserting them into the database. On the other hand, one can argue that the `policy` function complicates the control flow. The control flow in V1-NoFIO is more straightforward, since there is no need to parse the request twice.

6 Design V2: Adding a widget

Design V2 is the same as Design V1 except that the `tictactoe` page should now also display recent posts below the Tic Tac Toe game board. Figure 1 highlights the design change in the screenshot of the `tictactoe` page.

This design change affects the information flow of FacetBook because the `tictactoe` page now includes information from both portions of the database: the posts and the games.

```
parse_request request =
  let sandbox h = \database respond ->
       let censored = (undefined, snd database)  in
       h censored respond  in
  if WAI.pathInfo request == ["login"] then
    sandbox $ UCB.login
  else case check_credentials request of
    Nothing ->
      sandbox $ UCB.authentication_failed
    Just username -> case WAI.pathInfo request of
      ["post"] ->
        let content = get_parameter request "content"  in
        let permissions = get_parameter request "permissions"  in
        let users = words permissions  in
        if content /= "" && all valid_username users then
          do_create_post username content users
        else
          sandbox $ UCB.compose_post username
      ["dashboard"] ->
        dashboard username
      ["tictactoe"] ->
        let partner = get_parameter request "partner"  in
        if valid_username partner then
          let action = get_parameter request "action"  in
          sandbox $ UCB.tictactoe_play username partner action
        else
          sandbox $ UCB.tictactoe_select_partner username
      _ ->
        sandbox $ UCB.not_found
```

Figure 14: The code for the `parse_request` function in V1-NoFIO.

```
respond $ WAI.responseLBS status200 headers $
    render_tictactoe new_game username partner
```

Figure 15: Excerpt of the code to display a Tic Tac Toe game in V1-FIO.

```
d <- Read (fst database)
Swap $ do  --Fac
  all_posts <- flatten d
  return $ do  --FIO
    respond $ WAI.responseLBS status200 headers $
        render_tictactoe new_game username partner <>
        "<br /><br />Recent posts:<hr />" <>
        ByteString.intercalate "<hr />" (map escape (take 20 all_posts))
return ()
```

Figure 16: The new code to display a Tic Tac Toe game in V2-FIO.

7 V2-FIO

FacetBook V2-FIO implements Design V2, uses the FIO library, and is organized so that the change from V1 to V2 is as convenient as possible.

Figures 15 and 16 show the differences between V1-FIO and V2-FIO. Only these lines must change to implement the new widget.

In V2-FIO, the TCB is the same as in V1-FIO. The UCB contains 8 more lines of code.

Since the TCB is the same in V1-FIO and V2-FIO, no further changes are needed to minimize the TCB, which suggests that the information security is no worse than it was before. Furthermore, no special effort is required to maintain confidence in security when making the change from Design V1 to Design V2.

8 V2-NoFIO

FacetBook V2-NoFIO implements Design V2 without using the FIO library, and is organized so that the change from V1 to V2 is as convenient as possible.

Figures 17 and 18 show the differences between V1-NoFIO and V2-NoFIO.

```
respond $ WAI.responseLBS status200 headers $
    render_tictactoe new_game username partner
```

Figure 17: Excerpt of the code to display a Tic Tac Toe game in V1-NoFIO.

```
220 labeled_posts <- readIORef (fst database)
221 let d = filter_posts (Whitelist [username]) labeled_posts
222 let posts = flatten d
223 respond $ WAI.responseLBS status200 headers $
224     render_tictactoe new_game username partner <>
225     "<br /><br />Recent posts:<hr />" <>
226     ByteString.intercalate "<hr />" (map escape (take 20 posts))
```

Figure 18: The new code to display a Tic Tac Toe game in V2-NoFIO.

Aside from these changes, we must also remove the call to `sandbox` on line 202, which ruins the carefully audited boundary between the TCB and UCB. As a result, in V2-NoFIO, the file `UCB.hs` is poorly named because its contents must now be audited for information leaks. The TCB includes the whole codebase: 429 lines of code.

Note that V2-NoFIO is still secure (thanks to the call to `filter_posts` on line 221), just like all the other versions of FacetBook; however, the auditing effort to confirm its information security increased significantly when we removed the call to `sandbox` on line 202.

9 V2-NoFIO-minTCB

FacetBook V2-NoFIO-minTCB implements Design V2 without using the FIO library, and is organized so as to minimize the size of the TCB. In this section, we highlight the differences from V2-NoFIO.

To minimize the TCB, we must move the `tictactoe_play` function from `UCB.hs` to `TCB.hs`. To keep the TCB as small as possible, we also refactor it to call three new functions: `UCB.tictactoe_error_response`, `UCB.update_game`, and `UCB.tictactoe_play_response`.

Figure 19 shows the new code for `tictactoe_play`. Lines 231 and 234 set up appropriate sandboxes for calling the UCB functions on lines 232 and 236, which relieves auditors from reading the code in `UCB.hs` (aside from its import statements).

Compared to V1-NoFIO, the TCB is 10 lines larger, which suggests that the change has reduced confidence in the security of the system. Compared to V2-NoFIO, we modified 4 lines, moved 6 lines, and inserted 7 new lines; these changes were necessary to minimize the size of the TCB, suggesting that some nontrivial effort is required to maintain confidence in security. When FIO is unavailable, the next best sandboxing techniques lead to an inflexible architecture that becomes outdated when requirements change.

```
227 tictactoe_play username partner action database respond =
228   if partner == username then
229     respond $ UCB.tictactoe_error_response
230   else do  --IO
231     let censored_database = (undefined, snd database)
232     new_game <- UCB.update_game username partner action censored_database
233     labeled_posts <- readIORef (fst database)
234     let d = filter_posts (Whitelist [username]) labeled_posts
235     let posts = flatten d
236     respond $ UCB.tictactoe_play_response new_game username partner posts
```

Figure 19: The code for the `tictactoe_play` function in V2-NoFIO-minTCB.

10 Conclusions

To quantitatively answer the question of whether FIO makes it easier to achieve information security, we constructed the prototype social network application FacetBook, and measured the code changes required to add a widget for displaying recent posts alongside the Tic Tac Toe game.

10.1 Research question 1

Does FIO help minimize TCB size when coding a secure application?

The FIO library has 108 lines of code, and the application-specific TCB in V1-FIO has 99 lines of code. The application-specific TCB in V1-NoFIO has 118 lines of code.

In terms of total size, the TCB is smaller in V1-NoFIO. On the other hand, the code in FIO is not application-specific, and so the burden of auditing it for correctness can be amortized over many applications. So our results our inconclusive on this question, as FIO could be considered helpful or not, depending on one's point of view.

10.2 Research question 2

Does FIO help minimize TCB size when changing an existing application to meet new requirements?

In the FIO version of FacetBook, the feature extension requires no significant refactoring:

- We merely add code for getting the posts and displaying them in a widget. The extension adds 0 lines of code to the TCB, and no special refactoring is required.

On the other hand, in the non-FIO codebase, we have two unappealing options:

- We could simply remove the sandboxing and implement the extension without refactoring any module boundaries. By taking this approach, we greatly increase the size of the TCB, which now includes all of the code pertaining to Tic Tac Toe, including all helper functions: 419 lines of code altogether.
- We could carefully refactor the modules so that we only add to the TCB the code related to displaying the new widget; the other helper functions can remain outside of the TCB. The net result is still a larger TCB (10 more lines) and extra developer effort (17 changes) spent on refactoring.

From this experiment, we conclude that the FIO library makes it possible in some situations to extend the functionality of applications at no extra cost (in terms of TCB lines and refactoring effort). In comparison, without FIO, this feature extension either significantly decreases security (via a larger TCB) or requires additional refactoring effort to mitigate such a decrease.

11 Discussion

One design decision is the richness of the security policy. For instance, we could include all of the rules of the Tic Tac Toe game in the policy, thus enforcing fair and correct playing of the game. However, since the security policy lies within the TCB, a larger policy means greater difficulty auditing the policy itself for correctness. Therefore, since correct functionality of the Tic Tac Toe game is less important than enforcing post visibility settings, we choose to include in the policy only the code pertaining to the latter criterion.

Another design choice is whether to make the policy a "transparent" wrapper around the functioning system (analogous to higher-order contracts being projections [4] that do not modify the behavior of correct programs) or to integrate the policy into the functioning system itself. For instance, in FacetBook, the policy code must inspect the request parameters to determine the request's meaning; should this part of the code be duplicated in the functioning system, which also needs to determine each request's meaning? We have chosen to duplicate this code, so there are some similarities in the control flow of functions `policy` and `parse_request` (Figures 6 and 12).

For the database, we use the FIORef type from our FIO library to keep persistent state in memory. For the list of ongoing Tic Tac Toe games, the FIORef will never become faceted because that data is public for everyone to see; however, for the faceted list of posts, the situation is more complicated. Specifically, since faceted execution works by refusing to update the facets that are forbidden from seeing the effects of the currently executing code, the data structure must operate in an append-only manner, lest we degrade performance by creating an exponentially large faceted structure. Some work by Algehed, Russo, and Flanagan [1] will address this performance-related limitation of faceted execution. For now, in FacetBook, we simply use two separate FIORefs: one for the

list of Tic Tac Toe games (a non-faceted, non-append-only data structure), and one for the list of posts (a faceted, append-only data structure).

References

[1] Maximilian Algehed, Alejandro Russo, and Cormac Flanagan. "Optimizing Faceted Secure Multi-Execution". In: *Computer Security Foundations Symposium (CSF'19)*. IEEE. 2019.

[2] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. "Secure serverless computing using dynamic information flow control". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), p. 118.

[3] Ethan Cecchetti, Andrew C Myers, and Owen Arden. "Nonmalleable information flow control". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017, pp. 1875–1891.

[4] Robert Bruce Findler and Matthias Blume. "Contracts as pairs of projections". In: *International Symposium on Functional and Logic Programming*. Springer. 2006, pp. 226–241.

[5] Minh Ngo, Nataliia Bielova, Cormac Flanagan, Tamara Rezk, Alejandro Russo, and Thomas Schmitz. "A Better Facet of Dynamic Information Flow Control". In: *WWW'18 Companion: The 2018 Web Conference Companion*. 2018, pp. 1–9.

[6] Thomas Schmitz, Dustin Rhodes, Thomas H. Austin, Kenneth Knowles, and Cormac Flanagan. "Faceted Dynamic Information Flow via Control and Data Monads". In: *Principles of Security and Trust - 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. 2016.

[7] Michael D Schroeder. "Engineering a security kernel for multics". In: *ACM SIGOPS Operating Systems Review*. Vol. 9. 5. ACM. 1975, pp. 25–32.

[8] Jean Yang, Travis Hance, Thomas H Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. "Precise, dynamic information flow for database-backed applications". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2016.