Deliverable 2: Project Report

Team Members: Autrin Hakimi, Bradley Gaines, Varun Jain
Date: April 25, 2025

---

1. Selected Project

Project Name: Signal

Project Description: Signal is a cross-platform secure messaging application

Repository Link: https://github.com/signalapp/Signal-Desktop, https://github.com/signalapp/Signal-iOS, and https://github.com/signalapp/Signal-Android.

Justification for Selection

- **Community Activity:** Reviewing https://github.com/signalapp/Signal-Desktop/pulse shows us that the repo is active. There have been recent releases, merges, pull requests, conversations, issues opened, and issues closed.
- **Relevance:** Signal aligns perfectly with our skills in JavaScript, React, and UI development, directly applying course objectives to improve a widely used privacy-focused platform. Enhancing Signal's user interface contributes to the ease of secure communication, thus supporting its underlying privacy mission. It is also open source, which allows us to contribute to it.
- **Feasibility:** While Signal is a complex application, our task allows us to focus on a specific area (sticker synchronization) and gradually expand our knowledge of the codebase. We can use our existing React, JavaScript, and Java skills to grasp the relevant UI components and data handling mechanisms quickly. The codebase, while substantial, is well-structured, enabling us to isolate the relevant modules and focus our efforts on targeted learning and development.
- **Comparison:** We also considered working on an issue from Mozilla Firefox. However, the build process, as well as finding open, unassigned issues accurately, was too complex. Alongside this, the variety of products under Firefox and their relation were hard to determine. Due to the bugs' complexity and the abovementioned reasons, we decided not to work on Firefox.

---

2. Project Context and Business Model

Signal is a privacy-first, open-source messaging application developed to address growing concerns about surveillance and data exploitation in digital communication. Launched in 2014 by Moxie Marlinspike and Stuart Anderson, Signal emerged from earlier projects like TextSecure and RedPhone, which focused on encrypted texting and calling. In 2018, WhatsApp co-founder Brian Acton joined Signal, forming the Signal Foundation, a nonprofit organization, to ensure sustainable development free from profit-driven motives.

Competitive Landscape

Signal competes with both closed-source platforms (e.g., WhatsApp, iMessage) and open-source alternatives (e.g., Telegram, Matrix). Unlike WhatsApp (owned by Meta), Signal does not monetize user data or serve ads. While Telegram offers similar features, it lacks Signal's end-to-end encryption by default and a transparent, auditable codebase. Signal's commitment to privacy distinguishes it as a trusted tool for activists, journalists, and privacy-conscious users.

Business Model

Signal operates as a nonprofit funded primarily through donations and grants, ensuring alignment with its mission rather than shareholder interests. The Signal Foundation's funding model avoids conflicts of interest inherent in ad-supported platforms, allowing it to prioritize user privacy. This structure fosters community trust, as the project's survival depends on user support rather than data exploitation.

Significance

Signal exists to provide a secure, accessible alternative to mainstream messaging platforms. Its open-source nature enables transparency, allowing independent verification of its security claims. In an era of pervasive data harvesting, Signal's resistance to surveillance capitalism makes it a critical tool for safeguarding digital rights. The project's sustainability relies on its community-driven ethos, ensuring it remains accountable to users rather than corporate interests.

---

3. Task Description

Implementing Sticker Usage Synchronization in Signal Desktop

The implemented changes solve a critical user experience issue in Signal Desktop where recently used stickers were not synchronizing between devices. This issue created a disjointed experience for users who had to re-select their favorite stickers when switching between Windows, iOS, or other platforms.
Our implementation follows Signal's existing patterns for synchronizing data between devices and draws inspiration from how sticker pack installations are already synchronized. The solution involves several coordinated changes across the codebase:

Mechanism of Synchronization

1. Creating the Sync Message Protocol: We added a new StickerUsageSync message type to Signal's protocol buffer definitions in SignalService.proto, following the existing pattern for sync messages. This message contains essential sticker identification (packId, stickerId) and usage timestamp information.
2. Sending Sync Messages: When a user selects a sticker on one device, the doUseSticker function in stickers.ts now calls sendRecentStickerSync to dispatch this information to all linked devices. The implementation in textsecure.ts enqueues this sync message for transmission through Signal's existing job queue infrastructure.
3. Message Generation: In SendMessage.ts, we implemented getStickerUsageSync to generate the properly formatted protocol message. This maintains consistency with Signal's architectural pattern for sync messages.
4. Message Reception: When a device receives a sticker usage sync message, handleSyncMessage in MessageReceiver.ts processes it by updating the local database with the sticker usage information using updateStickerLastUsed.
5. Integration with Message Types: We added the 'stickerUsageSync' type to the message type enumeration in handleMessageSend.ts to properly categorize these messages within Signal's messaging system.

Design Considerations

Our implementation addresses several key concerns in synchronizing sticker usage:

1. Battery and Data Efficiency: The sync messages are small and only sent when a sticker is used, minimizing battery drain and data usage. Stickers are represented by their IDs rather than sending the actual sticker data again.
2. Seamless User Experience: Users can now maintain consistent access to their recently used stickers across all their devices without manual re-selection.
3. Architectural Consistency: The solution follows Signal's existing patterns for device synchronization, maintaining architectural integrity, and ensuring compatibility with future updates.
4. Privacy Preservation: The synchronization mechanism maintains Signal's privacy-focused approach, only sharing necessary information between a user's devices.

This implementation represents a meaningful improvement to the user experience while adhering to Signal's design principles and architectural patterns. The changes were carefully crafted to integrate with the existing codebase, maintaining a consistent approach to data synchronization across the application.

Returning to the app from Captcha

There is a critical bug that we did not anticipate. After entering the phone number, the app directs the user to https://signalcaptchas.org/staging/registration/generate. There, the user solves a 2-step captcha. But the problem is that it is impossible to return to the app with the Captcha token. So the running app, Electron, is never able to get the Captcha from the browser.

To work around the captcha

https://github.com/signalapp/Signal-Desktop/issues/6455#issuecomment-2829063275

- Go to the Signal-Desktop directory
- pnpm start
- Relink the account
- Solve the captcha in a browser that lets you see the captcha through the Network tab in Inspect (e.g., Edge or Chrome). Firefox doesn't show it.
- Copy the captcha URL.
    - It starts like this signalcaptcha://signal-hcaptcha-short…
- Then in another terminal:
- Go to the Signal-Desktop directory
- pnpm start -- "captcha url here"
- -- tells pnpm to pass the argument through to Electron
- There you go! You passed the captcha verification.
- Now enter the code you received through sms or call and register your account.

---

4. Submitted Artifacts

- Code & Documentation:
  - https://github.com/autrin/Signal-Desktop/tree/fix/recent-stickers-sync
  - Below you will find the code changes I made:

The function below is in ts\shims\textsecure.ts:

```
export async function sendRecentStickerSync(
  packId: string,
  stickerId: number,
  timestamp: number
): Promise<void> {
  if (window.ConversationController.areWePrimaryDevice()) {
    log.warn(
      'shims/sendRecentStickerSync: We are primary device; not sending sync'
    );
    return;
  }

  try {
    await singleProtoJobQueue.add(
      MessageSender.getStickerUsageSync({
        packId,
        stickerId,
        timestamp,
      })
    );
  } catch (error) {
    log.error(
      'sendRecentStickerSync: Failed to queue sync message',
      Errors.toLogFormat(error)
    );
  }
}
```

this one in protos\SignalService.proto:

```
  message StickerUsageSync {
    optional bytes packId = 1;
    optional uint32 stickerId = 2;
    optional int64 timestamp = 3;
  }

  optional StickerUsageSync stickerUsageSync = 26;
```

This one in ts/state/ducks/stickers.ts:

```
import { sendRecentStickerSync, sendStickerPackSync } from
'../../shims/textsecure';
```

and in function doUseSticker():

```
  // Send sync message to other devices
  // This is similar to how sticker pack installation sync
works
  drop(sendRecentStickerSync(packId, stickerId, time));
```

The below one in ts/textsecure/MessageReceiver.ts:

```
import { redactPackId } from '../types/Stickers';
import { DataWriter } from '../sql/Client';

const { updateStickerLastUsed } = DataWriter;

Then in function #handleSyncMessage():

    if (syncMessage.stickerUsageSync) {
      const { packId, stickerId, timestamp } =
syncMessage.stickerUsageSync;

      // Convert binary packId to hex string
      const packIdHex = packId ? Bytes.toHex(packId) : '';


```

```
        log.info(
          'MessageReceiver: processing sticker usage sync',
          `${redactPackId(packIdHex)} ${stickerId}`
        );

        // Update the local database with the sticker usage
    information
        await updateStickerLastUsed(
          packIdHex,
          Number(stickerId),
          Number(timestamp)
        );
      }
```

This one is added to ts/util/handleMessageSend.ts:

```
      added   'stickerUsageSync',
```

to export const sendTypesEnum = z.enum([])

The implementation below is in ts/textsecure/SendMessage.ts:

```
       public static getStickerUsageSync({
           packId,
           stickerId,
           timestamp,
         }: {
           packId: string;
           stickerId: number;
           timestamp: number;
         }): SingleProtoJobData {
           const myAci =
    window.textsecure.storage.user.getCheckedAci();
           const { ContentHint } =
    Proto.UnidentifiedSenderMessage.Message;

           const syncMessage =
    MessageSender.createSyncMessage();
```

```
          const stickerUsageSync = new
  Proto.SyncMessage.StickerUsageSync();

          stickerUsageSync.packId = Bytes.fromHex(packId);
          stickerUsageSync.stickerId = stickerId;
          stickerUsageSync.timestamp =
  Long.fromNumber(timestamp);

          syncMessage.stickerUsageSync = stickerUsageSync;

          const contentMessage = new Proto.Content();
          contentMessage.syncMessage = syncMessage;

          return {
            contentHint: ContentHint.RESENDABLE,
            serviceId: myAci,
            isSyncMessage: true,
            protoBase64: Bytes.toBase64(
              Proto.Content.encode(contentMessage).finish()
            ),
            type: 'stickerUsageSync',
            urgent: false,
          };
        }
```

- I also commented about a major problem that we had with captcha: https://github.com/signalapp/Signal-Desktop/issues/6455#issuecomment-2829063275

---

5. QA Strategy

Our QA strategy for implementing sticker synchronization functionality in Signal Desktop involved multiple layers of testing to ensure robustness and adherence to Signal's high standards for privacy and reliability.

Automated Testing

We leveraged Signal's extensive test infrastructure to validate our changes:

1. Unit Tests: We ran the existing test suite using pnpm test to ensure our changes didn't break existing functionality. Signal Desktop has a comprehensive test suite built with Mocha and Chai that covers components, services, and utilities.
2. Static Analysis:
    - TypeScript's type checking ensured type safety across our implementation
    - ESLint enforced code style and caught potential errors
    - The repository's CI workflows automatically ran these checks on our code
3. Integration Tests: By running pnpm start after making changes, we verified that the application still functioned correctly, with a particular focus on the sticker-related components.

Manual Testing

Given the complexity of Signal's synchronization mechanisms, manual testing was crucial:

1. Functionality Testing: We manually validated that:
    - Recently used stickers were properly stored locally
    - The sync message was correctly generated and sent when a sticker was used
    - Received sync messages updated the sticker history appropriately
2. Cross-Platform Testing: We attempted to test synchronization between:
    - Multiple Signal Desktop instances running in development mode
    - Between Desktop and mobile platforms (though captcha issues limited this)
3. Edge Case Testing: We explored potential failure scenarios:
    - Network interruptions during sync message transmission
    - Out-of-order message receipt
    - Handling of invalid sticker data

Documentation

As part of our QA process, we documented:

1. The changes made to various system components
2. The interaction flow between components
3. The protocol additions required for sticker sync messages
4. Known limitations and potential edge cases

Testing Challenges

Our QA efforts were constrained by several factors:

1. Captcha Issues: As detailed in our "Unanticipated Risk" section, difficulties with the captcha system limited our ability to test cross-device synchronization in the staging environment
2. Setup Complexity: The multi-step process to create a functioning development environment across platforms created overhead for thorough testing
3. Limited Documentation: Some testing scenarios were difficult to replicate due to sparse documentation on testing cross-device functionality

---

6. QA Evidence

- After the modifications to the code, the app still runs, and all the features are still usable. The user can send messages and stickers.
  You can find the results of some automated tests in the URL below:
  https://docs.google.com/document/d/14c-oV0WrvpifmRXOZulkEbq4yOHdfj6LAKCDFMVrwOM/edit?usp=sharing
- Note: In the "Unanticipated Risk" section of 7, you will read about the problems we faced with Captcha. We were able to solve the Captcha issue on Desktop, but IOS also had another issue with Captcha. This blocked our way to test our code on IOS to see if the stickers are getting synchronized.

---

7. Plan Updates

Deviations from Deliverable 1

We were able to implement a solid fix for the bug on time. However, the further we got into understanding the signal codebase and creating our solution, we realized our client-side synchronization could only go so far. While we modified messaging methods to implement our sticker tracking, saving, storing, and sending as well as receiving, these changes can not be fully realized by us alone, due to the many anticipated risks discussed below, our time spent setting up environments for testing was drastically larger than planned. We made up for this by taking more time for code analysis, theoretical implementations, and implementation plans. This way, we were more prepared when we got the staging environments working. We also worked more closely on tackling larger issues on setting up individual environments (Desktop, IOS, Android) to not burn one person out on an issue and have a more varied skill set and perspective.

Unanticipated Risk

Our primary sources of unanticipated risk and hardship were Captcha issues and undescriptive and outdated documentation for building staging environments across all platforms Signal supports. Many of the documentations assumed high technical knowledge of associated platforms (Xcode, etc.) and had very few nondescript instructions on how/what to modify for setting up environments. And they also did not give clear instructions about what modifications needed to be made to move from production to staging. This meant that while we could read the code and come to understand its functionality and our implementations, creating a working staging environment that could be run and tested proved to be a far more strenuous and time-consuming challenge than we had anticipated. We also ran into several instructions with outdated data on variable names, UI layouts, and instructions on tools being used. This requires lots of time to troubleshoot and learn how to translate these instructions into the new environment. An example of outdated functionalities is `window.reduxActions.app.openInstaller();` not being used anymore in the newer Signal versions after v6. An example of not having complete instructions was not mentioning different staging endpoints and the 3 CDNs, or how to run the IOS app in the staging environment instead of production.

Another big issue we ran into was the captcha. The captcha gave a multitude of issues and errors across every platform and environment during our attempts to make staging environments. Despite using the staging configurations given to us in the repository created by the Signal developers, the staging URLs they pointed to in Xcode would timeout and never receive captcha tokens for us to validate our sessions. Signal Desktop suffered from a different captcha issue of not properly receiving the captcha complete token from the captcha staging server. Despite following all the instructions for building, the captcha system seemed very flawed and did not work as per their instructions. Signal uses its own captcha system, and it has bugs that need to be fixed.

---

8. Team Reflection

8.a Teamwork

Our team embraced a collaborative approach throughout this project. We initially divided responsibilities by platform expertise, but quickly adapted to a more flexible, support-oriented model when we encountered unexpected challenges. Regular communication and knowledge sharing were key to our success, particularly when facing complex issues with the development environment and captcha integration. Working together allowed us to leverage each person's unique perspectives and skills, which proved invaluable when developing our solution for sticker synchronization. Learning to pivot from individual work to collective problem-solving was one of our most valuable takeaways from this experience.

8.b Agile Practices

Our team would have benefited from adopting more agile practices earlier in the project. Daily stand-ups would have helped us identify blockers sooner, particularly the captcha issues that consumed significant time. Pair programming proved extremely valuable when we did use it, especially when tackling complex parts of the codebase. Test-driven development would have provided more confidence in our changes, given the complexity of the Signal protocol and synchronization mechanisms. Our productivity improved significantly when we eventually adopted more agile principles like incremental development and frequent communication. These experiences reinforced how agile practices can be especially valuable when working on complex, established codebases.

8.c Software Maintenance Principles

Signal enforces strict coding standards that we carefully followed, modeling our implementation after existing patterns for synchronization messages. We maintained architectural consistency by studying how sticker pack installations were synchronized and applying similar patterns to sticker usage. The project employs thorough automated testing, which we leveraged to verify our changes didn't break existing functionality. One area where Signal could improve is documentation quality. We encountered outdated and sometimes incomplete references that complicated the onboarding process. Better documentation of internal message flows, protocol structures, and development environment requirements would significantly reduce barriers for new contributors and improve maintainability

8.d Open-Source

This project has significantly enhanced our appreciation for open-source development. While Signal's complexity presented challenges, working on real-world code that impacts millions of users was incredibly rewarding. The experience taught us how open-source projects balance security, privacy, and usability concerns while maintaining a quality codebase. We were particularly impressed by Signal's architectural consistency and commitment to privacy-first design. Though the initial learning curve was steep, the knowledge gained about protocol design, secure messaging, and cross-platform synchronization is invaluable. We're all more

likely to contribute to open-source projects in the future, having experienced firsthand how our contributions can improve the user experience for a widely used application.
