How PDFrw and fillable forms improves throughput at a Covid-19 Vaccine Clinic

Haw-minn Lu‡*, José Unpingco‡

Abstract—PDFrw was used to prepopulate Covid-19 vaccination forms to improve the efficiency and integrity of the vaccination process in terms of federal and state privacy requirements. We will describe the vaccination process from the initial appointment, through the vaccination delivery, to the creation of subsequent required documentation. Although Python modules for PDF generation are common, they struggle with managing fillable forms where a fillable field may appear multiple times within the same form. Additionally, field types such as checkboxes, radio buttons, lists and combo boxes are not straightforward to programmatically fill. Another challenge is combining multiple *filled* forms while maintaining the integrity of the values of the fillable fields. Additionally, HIPAA compliance issues are discussed.

Index Terms—acrobat documents, form filling, HIPAA compliance, COVID-19

Introduction

The coronavirus pandemic has been one of the most disruptive nationwide events in living memory. The frail, vulnerable, and elderly have been disproportionately affected by serious hospitalizations and deaths. Notwithstanding the amazing pace of vaccine development, logistical problems can still inhibit large-scale vaccine distribution, especially among the elderly. Vaccination centers typically require online appointments to facilitate vaccine distribution by State and Federal governments, but many elderly do not have Internet access or know how to make online appointments, or how to use online resources to coordinate transportation to and from the vaccination site, as needed.

As a personal anecdote, when vaccinations were opened to all aged 65 and older, one of the authors tried to get his parents vaccinated and discovered that the experience documented here [Let21] was unfortunately typical and required regularly pinging the appointment website for a week to get an appointment. However, beyond persistence, getting an appointment required monitoring the website to track when batches of new appointments were released --- all tasks that require an uncommon knowledge of Internet infrastructure beyond most patients, not just the elderly.

To help San Diego County with the vaccine rollout, the Gary and Mary West PACE (WestPACE) center established a pop-up point of distribution (POD) for the COVID-19 vaccine [pre21] specifically for the elderly with emphasis on those who are most vulnerable. The success in the POD was reported in the local news media [Lit21] [Col21] and prompted the State of California to ask WestPACE's sister organization (the Gary and Mary West Health Institute) to develop a playbook for the deploying a pop-up POD [pod21].

This paper describes the logistical challenges regarding the vaccination rollout for WestPACE and focuses on the use of Python's `PDFrw` module to address real-world sensitive data issues with PDF documents.

This paper gives a little more background of the effort. Next the overall infrastructure and information flow is described. Finally, a very detailed discussion on the use of python and the `PDFrw` library to address a major bottleneck and volunteer pain point.

Background

WestPACE operates a Program of All-Inclusive Care for the Elderly (PACE) center which provides nursing-home-level care and wrap-around services such as transportation to the most vulnerable elderly. To provide vaccinations to WestPACE patients as quickly as possible, WestPACE tried to acquire suitable freezers (some vaccines require special cold storage) instead of waiting for San Diego County to provide them; but, due to high-demand, acquiring a suitably-sized freezer was very problematic. As a pivot, WestPACE opted to acquire a freezer that was available but with excess capacity beyond what was needed for just WestPACE, and then collaborated with the County to use this excess capacity to establish a walk-up vaccination center for all San Diego senior citizens, in or out of WestPACE.

WestPACE coordinated with the local 2-1-1 organization responsible for coordination of community health and disaster services. The 2-1-1 organization provided a call center with in-person support for vaccine appointments and transportation coordination to and from WestPACE. This immediately eased the difficulty of making online appointments and the burden of transportation coordination. With these relationships in place, the vaccination clinic went from concept to active vaccine distribution site in about two weeks resulting in the successful vaccination of thousands of elderly.

Although this is a technical paper, this background describes the real impact technology can make in the lives of the vulnerable and elderly in society in a crisis situation.

Infrastructure

The goal of the WestPACE vaccine clinic was to provide a friendly environment to vaccinate senior citizens. Because this was a non-profit and volunteer effort, the clinic did not have any pre-existing

* *Corresponding author: hlu@westhealth.org*
‡ *Gary and Mary West Health Institute*

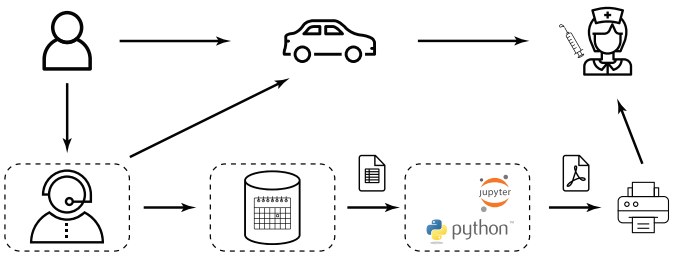

Fig. 1: *Vaccination Pipeline*

record management practices with corresponding IT infrastructure to handle sensitive health information according to Health Insurance Portability and Accountability Act (HIPAA) standards. One key obstacle is paperwork for appointments, questionnaires, consent forms, and reminder cards (among others) that must be processed securely and at speed, given the fierce demand for vaccines. Putting the burden of dealing with this paperwork on the patients would be confusing for the patient and time-consuming and limit the overall count of vaccinations delivered. Thus, the strategy was to use electronic systems to handle Protected Health Information (PHI) wherever possible and comply with HIPAA requirements [MF19] for data encryption at rest and in-transit, including appropriate Business Associate Agreements (BAA) for any cloud service providers [FKR+16]. For physical paper, HIPAA requirements mean that PHI must always be kept in a locked room or a container with restricted access.

Figure 1 shows a high level view of the user experience and information flow. Making appointments can be challenging, especially those with limited caregiver support. Because the appointment systems were set up in a hurry, many user interfaces were confusing and poorly designed. In the depicted pipeline, the person (or caregiver) telephones the 2-1-1 call center and the live operator collects demographic and health information, and coordinates any necessary travel arrangements, as needed. The demographic and health information is entered into the appointment system managed by the California Department of Public Health. The information is then downloaded to the clinic from the appointment system the day before the scheduled vaccination. Next, a forms packet is generated for every scheduled patient and consolidated into a PDF file that is then printed and handed to the volunteers at the clinic. The packet consolidates documents including consent forms, health forms, and CDC-provided vaccination cards.

When the patient arrives at the clinic, their forms are pulled and a volunteer reviews the questions while correcting any errors. Once the information is validated, the patient is directed to sign the appropriate forms. The crucially efficient part is that the patient and volunteer only have to *validate* previously collected information instead of filling out multiple forms with redundant information. This was crucial during peak demand so that most patients experienced less than a five minute delay between arrival and vaccine administration. While there was consideration of commercial services to do the electronic form filling and electronic signatures, they were discounted because these turned out to be too expensive and time-consuming to set up.

Different entities such as 2-1-1 and the State of California handle certain elements of the data pipeline, but strict HIPAA requirements are followed at each step. All clinic communications with the State appointment system were managed through a properly authenticated and encrypted system. The vaccine clinic utilized pre-existing, cloud-based HIPAA-compliant system, with corresponding BAAs. All sensitive data processing occurred on this system. The system, which is described at [HmLAKJU20], uses both python alone and in Jupyter notebooks.

Finally, the processed PDF forms were transferred using encryption to a server at the clinic site where an authorized operator printed them out. The paper forms were placed in the custody of a clinic volunteer until they were delivered to a back office for storage in a locked cabinet, pursuant to health department regulations.

Though all aspects of the pipeline faced challenges, the pre-population of forms turned out to be surprisingly difficult due to the lack of programmatic PDF tools that properly work with fillable forms. The remainder of the paper discusses the challenges and provides instructions on how to use Python to fill PDF forms for printing.

Programmatically Fill Forms

Programmatically filling in PDF forms can be a quick and accurate way to disseminate forms. Bits and pieces can be found throughout the Internet and places like Stack Overflow but no single source provides a complete answer. The *Medium* blog post by Vivsvaan Sharma [Sha20] is a good starting place. Another useful resource is the PDF 1.7 specification [pdf08]. Since the deployment of the vaccine clinic, the details of the form filling can be found at WestHealth's blog [Lu21]. The code is available on GitHub as described below.

The following imports are used in the examples given below.

```
import pdfrw
from pdfrw.objects.pdfstring import PdfString
from pdfrw.objects.pdfstring import BasePdfName
from pdfrw import PdfDict, PdfObject
```

Finding Your Way Around PDFrw and Fillable Forms

Several examples of basic form filling code can be found on the Internet, including the above-mentioned *Medium* blog post. The following is a typical snippet which was taken largely from the blog post.

```
pdf = pdfrw.PdfReader(file_path)
for page in pdf.pages:
    annotations = page['/Annots']
    if annotations is None:
        continue

    for annotation in annotations:
        if annotation['/Subtype']=='/Widget':
            if annotation['/T']:
                key = annotation['/T'].to_unicode()
                print (key)
```

The type of `annotation['/T']` is `pdfString`. While some sources use `[1:-1]` to extract the string from `pdfString`, the `to_unicode` method is the proper way to extract the string. According to the PDF 1.7 specification § 12.5.6.19, all fillable forms use widget annotation. The check for `annotation['/SubType']` filters the annotations to only widget annotations.

To set the value `value`, a `PDFString` needs to be created by encoding `value` with the `encode` method. The encoded `PDFString` is then used to update the `annotation` as shown in the following code snippet.

```
annotation.update(PdfDict(V=PdfString.encode(value)))
```

This converts `value` into a `PdfString` and updates the `annotation`, creating a value for `annotation['/V']`.

In addition, at the top level of the `PdfReader` object `pdf`, the `NeedAppearances` property in the interactive form dictionary, `AcroForm` (See § 12.7.2) needs to be set, without this, the fields are updated but will not necessarily display. To remedy this, the following code snippet can be used.

```
pdf.Root.AcroForm.update(PdfDict(
    NeedAppearances=PdfObject('true')))
```

Multiple Fields with Same Name

Combining the code snippets provides a simple method for filling in text fields, except if there are multiple instances of the same field. To refer back to the clinic example, each patient's form packet comprised multiple forms each with the `Name` field. Some forms even had the `Name` appear twice such as in a demographic section and then in a `Print Name` field next to a signature line. If the code above on such a form were run, the `Name` field will not show up.

Whenever the multiple fields occur with the same name, the situation is more complicated. One way to deal with this is to simply rename the fields to be different such as `Name-1` and `Name-2`, which is fine if the sole use of the form is for automated form filling. This would require access to a form authoring tool. If the form is also to be used for manual filling, this would require the user to enter the `Name` multiple times.

When fields appear multiple times, the widget annotation does not have the `/T` field but has a `/Parent` field. As it turns out this `/Parent` contains the field name `/T` as well as the default value `/V`. Each `/Parent` has one `/Kids` for each occurrence of the field. To modify the code to handle repeated occurrences of a field, the following lines can be inserted:

```
if not annotation['/T']:
    annotation=annotation['/Parent']
```

These lines allow the inspection and modifications of annotations that appear more than once. With this modification, the result of the inspection code yields:

```
pdf = pdfrw.PdfReader(file_path)
for page in pdf.pages:
    annotations = page['/Annots']
    if annotations is None:
        continue

    for annotation in annotations:
        if annotation['/Subtype']=='/Widget':
            if not annotation['/T']:
                annotation=annotation['/Parent']
            if annotation['/T']:
                key = annotation['/T'].to_unicode()
                print (key)
```

With this code in the above example, `Name` would be printed multiple times, once for each instance, but each instance points to the same `/Parent`. With this modification, the form filler actually fills the `/Parent` value multiple times, but this has no impact since it is overwriting the default value with the same value.

Checkboxes

In accordance to §12.7.4.2.3, the checkbox state can be set as follows:

```
def checkbox(annotation, value):
    if value:
        val_str = BasePdfName('/Yes')
    else:
        val_str = BasePdfName('/Off')
    annotation.update(PdfDict(V=val_str))
```

This could work if the export value of the checkbox is `Yes`, which is the default, but not when the export value is something else. The easiest solution is to edit the form to ensure that the export value of the checkbox is `Yes` and the default state of the box is unchecked. The recommendation in the specification is that it be set to `Yes`. In the event tools to make this change are not available, the `/V` and `/AS` fields should be set to the export value not `Yes`. The export value can be inspected by examining the appearance dictionary `/AP` and specifically at the `/N` field. Each annotation has up to three appearances in its appearance dictionary: `/N`, `/R` and `/D`, standing for *normal*, *rollover*, and *down* (§12.5.5). The latter two have to do with appearance in interacting with the mouse. The normal appearance has to do with how the form is printed.

There may be circumstances where the form has checkboxes whose default state is checked. In that case, in order to uncheck a box, the best practice is to delete the `/V` as well as the `/AS` field from the dictionary.

According to the PDF specification for checkboxes, the appearance stream `/AS` should be set to the same value as `/V`. Failure to do so may mean that the checkboxes do not appear.

More Complex Forms

For the purpose of the vaccine clinic application, the filling of text fields and checkboxes were all that were needed. However, for completeness, other form field types were studied and solutions are given below.

Radio Buttons

Radio buttons are by far the most complex of the form entry types. Each widget links to `/Kids` which represent the other buttons in the radio group. Each widget in a radio group will link to the same 'kids'. Much like the 'parents' for the repeated forms fields with the same name, each kid need only be updated once, but the same update can be used multiple times if it simplifies the code.

In a nutshell, the value `/V` of each widget in a radio group needs to be set to the export value of the button selected. In each kid, the appearance stream `/AS` should be set to `/Off` except for the kid corresponding to the export value. In order to identify the kid with its corresponding export value, the `/N` field of the appearance dictionary `/AP` needs to be examined just as was done with the checkboxes.

The resulting code could look like the following:

```
def radio_button(annotation, value):
    for each in annotation['/Kids']:
        # determine the export value of each kid
        keys = each['/AP']['/N'].keys()
        keys.remove('/Off')
        export = keys[0]

        if f'/{value}' == export:
            val_str = BasePdfName(f'/{value}')
        else:
            val_str = BasePdfName(f'/Off')
        each.update(PdfDict(AS=val_str))

    annotation.update(PdfDict(
        V=BasePdfName(f'/{value}')))
```

Combo Boxes and Lists

Both combo boxes and lists are forms of the form type *choice*. The combo boxes resemble drop-down menus and lists are similar to list pickers in HTML. Functionally, they are very similar in form filling. The value `/V` and appearance stream `/AS` need to be set to their exported values. The `/Op` field yields a list of lists associating the exported value with the value that appears in the widget.

To set the combo box, the value needs to be set to the export value.

```
def combobox(annotation, value):
    export=None
    for each in annotation['/Opt']:
        if each[1].to_unicode()==value:
            export = each[0].to_unicode()
    if export is None:
        err = f"Export Value: ""{value} Not Found"
        raise KeyError(err)
    pdfstr = PdfString.encode(export)
    annotation.update(PdfDict(V=pdfstr, AS=pdfstr))
```

Lists are structurally very similar. The list of exported values can be found in the `/Opt` field. The main difference is that lists based on their configuration can take multiple values. Multiple values can be set with `PDFrw` by setting `/V` and `/AS` to a list of `PdfStrings`. The code presented here uses two separate helpers, but because of the similarity in structure between list boxes and combo boxes, they could be combined into one function.

```
def listbox(annotation, values):
    pdfstrs=[]
    for value in values:
        export=None
        for each in annotation['/Opt']:
            if each[1].to_unicode()==value:
                export = each[0].to_unicode()
        if export is None:
            err = f"Export Value: {value} Not Found"
            raise KeyError(err)
        pdfstrs.append(PdfString.encode(export))
    annotation.update(PdfDict(V=pdfstrs, AS=pdfstrs))
```

Determining Form Field Types Programmatically

While PDF authoring tools or visual inspection can identify each form's type, the type can be determined programmatically as well. It is important to understand that fillable forms fall into four form types, button (push button, checkboxes and radio buttons), text, choice (combo box and list box), and signature. They correspond to following values of the `/FT` form type field of a given annotation, `/Btn`, `/Tx`, `/Ch` and `/Sig`, respectively. Since signature filling is not supported and the push button is a widget which can cause an action but is not fillable, those corresponding types are omitted from consideration.

To distinguish the types of buttons and choices, the form flags `/Ff` field is examined. For radio buttons, the 16th bit is set. For combo box the 18th bit is set. Please note that `annotation['/Ff']` returns a `PdfObject` when returned and must be coerced into an `int` for bit testing.

```
def field_type(annotation):
    ft = annotation['/FT']
    ff = annotation['/Ff']

    if ft == '/Tx':
        return 'text'
    if ft == '/Ch':
        if ff and int(ff) & 1 << 17:  # test 18th bit
            return 'combo'
        else:
            return 'list'
    if ft == '/Btn':
        if ff and int(ff) & 1 << 15:  # test 16th bit
            return 'radio'
        else:
            return 'checkbox'
```

For completeness, the following `text_form` filler helper is included.

```
def text_form(annotation, value):
    pdfstr = PdfString.encode(value)
    annotation.update(PdfDict(V=pdfstr, AS=pdfstr))
```

This completes the building blocks to an automatic form filler.

Consolidating Multiple Filled Forms

There are two problems with consolidating multiple filled forms. The first problem is that when two PDF files are merged, fields with matching names are associated with each other. For instance, if John Doe were entered in one form's name field and Jane Doe in the second. After combining the two forms John Doe will override the second form's name field and John Doe would appear in both forms. The second problem is that most simple command line or programmatic methods of combining two or more PDF files lose form data. One solution is to "flatten" each PDF file. This is equivalent to printing the file to PDF. In effect, this bakes in the filled form values and does not permit the editing the fields. Going even further, one could render the PDFs as images if the only requirement is that the combined files be printable. However, tools like `ghostscript`, `imagemagick`, and PDFUnite don't do a good job of preserving form data when rendering PDF files.

Form Field Name Collisions

Combining multiple filled PDF files was an issue for the vaccine clinic because the same form was filled out for multiple patients. The alternative of printing hundreds of individual forms was infeasible. To combine a batch of PDF forms, all form field names must be different. Thankfully, the solution is quite simple, in the process of filling out the form using the code above, rename (set) the value of `/T`.

```
def form_filler(in_path, data, out_path, suffix):
    pdf = pdfrw.PdfReader(in_path)
    for page in pdf.pages:
        annotations = page['/Annots']
        if annotations is None:
            continue

        for annotation in annotations:
            if annotation['/SubType'] == '/Widget':
                key = annotation['/T'].to_unicode()
                if key in data:
                    pdfstr = PdfString.encode(data[key])
                    new_key = key + suffix
                    annotation.update(
                        PdfDict(V=pdfstr, T=new_key))
        pdf.Root.AcroForm.update(PdfDict(
             NeedAppearances=PdfObject('true')))
        pdfrw.PdfWriter().write(out_path, pdf)
```

Only a unique suffix needs to be supplied to each form. The suffix can be as simple as a sequential number.

Combining the Files

Solutions for combining PDF files with `PDFrw` can be found on the Internet. The following recipe is typical:

```
writer = PdfWriter()
for fname in files:
    r = PdfReader(fname)
    writer.addpages(r.pages)
writer.write("output.pdf")
```

While the form data still exists in the output file, the rendering information is lost and won't show when displayed or printed. The problem comes from the fact that the written PDF does not have an interactive form dictionary (see §12.7.2 of the PDF 1.7 specification). In particular, the interactive forms dictionary contains the boolean `NeedAppearances` which needs to be set for fields to be shown. If the forms being combined have different interactive form dictionaries, they need to be merged. In this application where the source forms are identical among the various copies, any `AcroForm` dictionary can be used.

After obtaining the dictionary from `pdf.Root.AcroForm` (assuming the `PdfReader` object is stored in `pdf`), it is not clear how to add it to the `PdfWriter` object. The clue comes from a simple recipe for copying a pdf file.

```
pdf = PdfReader(in_file)
PdfWriter().write(out_file, pdf)
```

Examination of the underlying source code shows the second parameter `pdf` to be set to the attribute `trailer` of the `PdfWriter` object. Assuming `acro_form` contains the desired interactive form, the interactive form dictionary can be added to the output document by using `writer.trailer.Root.AcroForm = acro_form`.

Conclusion

A complete functional version of this PDF form filler is open source and can be found at WestHealth's GitHub repository https://github.com/WestHealth/pdf-form-filler. This process was able to produce large quantities of pre-populated forms for senior citizens seeking COVID-19 vaccinations relieving one of the bottlenecks that have plagued many other vaccine clinics.

References

[Col21] Annica Colbert. Seniors-only vaccination site. *KPBS News*, Feb 2021. URL: https://www.kpbs.org/podcasts/san-diego-news-now/2021/feb/11/seniors-only-vaccination-site/.

[FKR+16] Barbara L. Filkins, Ju Young Kim, Bruce Roberts, Winston Armstrong, Mark A. Miller, Michael L. Hultner, Anthony P. Castillo, Jean Christophe Ducom, Eric J. Topol, and Steven R. Steinhubl. Privacy and security in the era of digital health: What should translational researchers know and do about it? *American Journal of Translational Research*, 8(3):1560–1580, 2016. Publisher Copyright: © 2016, E-Century Publishing Corporation. All rights reserved.

[HmLAKJU20] Haw-minn Lu, Adrian Kwong, and José Unpingco. Securing Your Collaborative Jupyter Notebooks in the Cloud using Container and Load Balancing Services. In Meghann Agarwal, Chris Calloway, Dillon Niederhut, and David Shupe, editors, *Proceedings of the 19th Python in Science Conference*, pages 2 – 10, 2020. doi:10.25080/Majora-342d178e-001.

[Let21] U-T Letters. Opinion: Vaccination frustration grows as seniors weigh limited options. *San Diego Union-Tribune*, Jan 2021. URL: https://www.sandiegouniontribune.com/opinion/story/2021-01-22/vaccination-frustrations-grow-as-seniors-search-for-appointments.

[Lit21] Joe Little. For san diego seniors, making vaccination appointments is as easy as calling 211. *7 San Diego News*, Feb 2021. URL: https://www.nbcsandiego.com/news/local/for-san-diego-seniors-making-vaccination-appointments-is-as-easy-as-calling-211/2531346/.

[Lu21] Haw-minn Lu. Exploring fillable forms with pdfrw, Mar 2021. URL: https://westhealth.github.io/exploring-fillable-forms-with-pdfrw.html.

[MF19] Wilnellys Moore and Sarah Frye. Review of hipaa, part 1: History, protected health information, and privacy and security rules. *Journal of Nuclear Medicine Technology*, 47(4):269–272, 2019. URL: https://tech.snmjournals.org/content/47/4/269, arXiv:https://tech.snmjournals.org/content/47/4/269.full.pdf, doi:10.2967/jnmt.119.227819.

[pdf08] *Document Management - Portable Document Format - Part 1: PDF 1.7*. Adobe Systems Incorporated, 2008.

[pod21] Pop up vaccination point of distribution (pod) for seniors: A how to guide, Apr 2021. URL: https://www.westhealth.org/resource/vaccine-pod-for-seniors/.

[pre21] New covid-19 vaccine site for vulnerable seniors at west pace in san marcos. *West Health Press Releases*, Feb 2021. URL: https://www.westhealth.org/press-release/new-covid-19-vaccine-site-for-vulnerable-seniors-at-west-pace-in-san-marcos/.

[Sha20] Vivsvaan Sharma. Filling editable pdf in python, Aug 2020. URL: https://medium.com/@vivsvaan/filling-editable-pdf-in-python-76712c3ce99.