Making It Easier To Publish Your Work

Shailesh M. Pandey
The University of Texas at Austin
shailesh@cs.utexas.edu

Jack Stenglein
The University of Texas at Austin
jackstenglein@utexas.edu

Abstract

With increasingly high number of submissions and reviewers to the top research conferences, the assignment of papers to reviewers is a difficult problem facing conference organizers. In this paper, we show that it is possible to modify a paper submission in such a way as to force the Toronto Paper Matching System (TPMS), a popular solution to this problem, to pick arbitrarily chosen favorable reviewers for the paper. Specifically, we embed keywords into the PDF so that there are no visible changes to the document but TPMS determines the paper is related to the desired reviewer's area of expertise. We show the effectiveness of our attack through various experiments, and find that an attacker can get her paper reviewed by a favorable reviewer with a probability as high as 80%. We also highlight potential mitigations against the proposed attack and exciting directions for future research in this area.

1 Introduction

One of the most important, and difficult, tasks facing an academic conference organizing committee is the assignment of papers to reviewers. Top conferences receive thousands of submissions, which have to be assigned to thousands of reviewers (NeurIPS 2019 had 6743 submissions [Charrez 2020] and more than 4000 reviewers [neu 2020a]) within a few days after the conference submission deadline. Moreover, the assignment of each paper requires knowledge of both the topics explored in the paper as well as knowledge of reviewers' expertise. Other additional constraints, such as reviewer load, make the task impossible to be done manually by the program chair. This scale also makes it difficult for reviewers to rank all the submitted papers in the order of their preference. Hence, most conferences now use partial ordering by the reviewers and an automated system for this task.

The Toronto Paper Matching System (TPMS) [Charlin and Zemel 2013] is one such popular system that has been used repeatedly by NIPS, ICML, UAI, CVPR, ECML/PKDD, etc. The system first determines reviewers' expertise by looking at their own published papers, either crawling through their Google Scholar profile[1] or using papers submitted by the reviewers. TPMS then extracts features (bag of words) from the PDFs of papers submitted to the conference. A machine learning model, assisted by the partial rankings provided by reviewers, is learned to score each paper-reviewer pair based on the similarity of a paper's features to a reviewer's expertise. For this project we work with AutoBid, an open source implementation of TPMS [Parno 2020].

In this project, we show that it is possible to trick TPMS into assigning an arbitrary favorable reviewer to a paper by embedding metadata into the submission PDF. Specifically, we achieved two major goals with this project. We automate the process of embedding arbitrary text into any PDF such that the embedded text is parsed as text by command-line tools but not rendered by GUI-based PDF viewers. Thereafter, we come up with specific adversarial keywords that trick TPMS into assigning any given arbitrary paper to a reviewer of the adversary's choice. Putting both of these goals together, we alter a conference submission, without causing any visible side-effects, in a way that causes TPMS to assign a high score to a reviewer of our choice.

Collecting the dataset was a major part of this effort. We collected and cleaned our data-set, comprising of 3191 paper submissions in PDF format and 1759 reviewers with at least 5 and at most 10 publications per reviewer in PDF format. Moreover, we modified Autobid [Parno 2020], the open source implementation of TPMS, to run at this scale where not all data is present on the local disk. Finally, we automated the task of embedding keywords into PDFs in order to force Autobid to select a favorable reviewer for the PDF. In the rest of this paper, we explain these steps, the results of our evaluations, and possible future directions in more detail.

2 Goals

With a dataset of size comparable to that of a top conference, we designed our attack on TPMS with the following goals:

1. Embed keywords in a paper PDF so that there are no visible changes to the PDF, but the keywords are read by PDF parsers.
2. Force TPMS to select a specific reviewer for a modified paper.
3. Automate the process to match any arbitrary reviewer with any arbitrary paper.

3 Dataset

In order to come up with a reliable way to trick TPMS, we need access to an archive of submitted papers, a list of sample reviewers and their representative publications. The scale of the dataset should be comparable to a real top conference for us to make conclusive statements about our methods. For this project, we focus on the data from NeurIPS 2019 [neu

[1] http://scholar.google.com

2020b]. The next two sub-sections mention our data collection process, the problems faced therein, and the statistics of our dataset.

3.1 Data Collection

The UTCS department currently sets a disk quota of 20 GB. We suspected that our data collection would require more storage than this, so we decided to store our data in Amazon S3 and download a file to the local machine only when we operate on it. We currently have 23.5 GB of data stored in Amazon S3.

As a first step in our collection process, we acquire the papers that constitute our set of submitted papers. Since the list of all submissions to a conference is generally not made public, we use the list of papers published in NeurIPS for the past 5 years as the set of our submissions. This list for the Neural Information Processing Systems (NeurIPS) conference is available at [neu 2020c]. We download all the PDF files for years spanning from 2015 to 2019 and store them in Amazon S3 for future use.

Next, we get the names of relevant reviewers, and their representative publications, for our dataset. The names of all reviewers for NeurIPS 2019 is available at [neu 2020a]. Unfortunately, this page does not provide a link to the respective websites or Google Scholar profiles of these reviewers. We search Google Scholar for the name of each of these reviewers and store the `author_id` for all the queries which return just one result. This reduces the number of uniquely identifiable reviewers to 2726. Since Google Scholar does not provide an officially supported API, we use a rate-limited crawler for this and all subsequent Google Scholar interactions. This makes our data collection process very slow.

Once we have a Google Scholar `author_id`, we use it to get the author's Google Scholar page. We assume a publication with higher number of citations is highly representative of the author. Hence, we extract the identifiers (`citation_id`) for each reviewer's top 20 publications (or as many as available if the reviewer has less than 20 publications) based on the number of citations. We have 44869 unique publication identifiers for 2720 reviewers with an average of 16.49 publications per reviewer.

To get the URL of each publication's PDF, we get the Google Scholar page of each publication using its identifier (`citation_id`). We noticed that Google Scholar blocked the client IP for a day if we made more than 40 quick requests. However, the client was allowed to continue if the number of requests does not exceed 40 in any one hour duration. Hence, we divided the dataset of publication identifiers across 8 different machines[2] with each making around 30 requests.[3] Since TPMS [Charlin and Zemel 2013] expects at least 5 publications per reviewer, we try to get the URLs for 10 publications per reviewer. For each reviewer, we process the publications in decreasing order of citations until we have 10 URLs or we exhaust the identifiers fetched in the previous step. After this step, we have a list of 25351 unique URLs with an average of 9.32 URLs per reviewer. This step took about 5 days to complete using 8 machines.

We then download each PDF file using the URLs determined in the previous step. Each PDF file is stored in Amazon S3 for future use. This process originally took about 1.5 days to complete. However, when trying to run TPMS using these PDFs in order to generate the reviewers' areas of expertise, we discovered that many of our files were not actually PDFs. Instead, we had downloaded a large number of 404 pages and redirect responses. To fix this, we checked each file in S3 to ensure that it started with the PDF magic number (%PDF). We removed any files that did not start with this magic number. This process took about 0.5 days to complete. This left us with 15225 unique PDFs and 1759 reviewers with more than 5 publications.

# of submissions	3191
Total # of reviewers	2720
Total # of reviewers' publications	15225
# of reveiwers with at least 1 publication	2669
# of reviewers with at least 5 publications	1759

Table 1. Key data-set statistics.

3.2 Statistics

Table 1 lists the key statistics of our data-set. We collect the papers accepted in the past five years in NeurIPS as our set of submitted papers [neu 2020c]. We have a total of 3191 papers as our set of submissions. This is comparable to the number of submissions received in a top conference. To compile our reviewers, we use the list of NeurIPS 2019 reviewers available at [neu 2020a]. Although the web-page lists a total of 5133 reviewers, only 2726 of these are uniquely identifiable by a Google Scholar search.

We are able to collect URLs of 25351 publications' PDF version using the reviewers' Google Scholar page. However, only 15225 of these links point to PDFs. Other links either require an authorization or point to pages that in turn point to PDFs. Fig 2 shows the distribution of the number of fetched publications per reviewer. Out of a total of 2720 reviewers, we have at least one PDF for 2669 with an average of 5.6 ± 2.6 PDFs per reviewer. TPMS recommends using reviewers with at least 5 publication PDFs. We have 1759 reviewers that fit this criteria which entails about 5.4 reviews per reviewer, assuming that each submission is reviewed by 3 reviewers.

Fig 1 shows the word clouds of reviewer interests as listed on Google Scholar and first three words of each submission's title. The shared presence of words such as 'learning', 'deep',

[2]We used the department's public UNIX hosts accessible from `linux.cs.utexas.edu`.

[3]We introduced a delay of 120 + `random(0, 5)` seconds before each request.

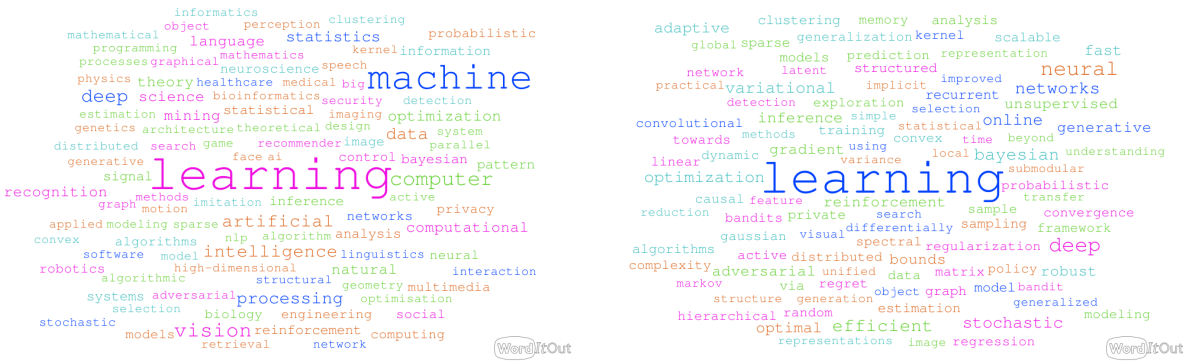


Figure 1. Word clouds of reviewer interests as listed on Google Scholar (left) and first three words of each submission's title (right). More frequent words are larger in the figure.

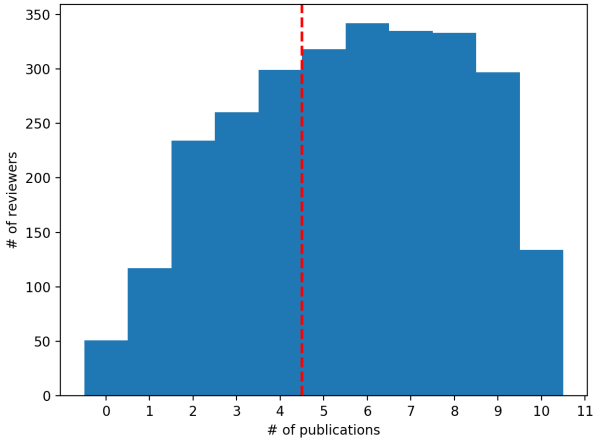


Figure 2. Distribution of the number of fetched publications per reviewer. The red line indicates the threshold of number of publications per reviewer. We plan to only use the reviewers to the right of this threshold.

'reinforcement', 'algorithms', etc., suggests that the reviewers in our dataset stand in good stead to review the submissions.

4 Running TPMS on our dataset

As previously stated, we use the open source implementation of TPMS known as Autobid [Parno 2020]. The Autobid repository contains two main branches: the `master` branch and the `prior_bids` branch. We worked with the `master` branch for this project. The two branches differ in how they create their models of reviewer expertise but not in how they scrape the PDF files. The `master` branch uses an LDA model for generating bids while the `prior_bids` branch uses StarSpace [Wu et al. 2017]. Therefore, if our work was extended to the `prior_bids` branch as well, our method of embedding key words into the PDF would not need to be changed. At most, the method of selecting key words to embed would need to be changed to fit the `prior_bid`'s reviewer expertise model.

On the `master` branch, Autobid has four main steps to generate reviewer-submission matchings. First, the reviewer publications are analyzed. Second, the submitted papers are analyzed. Third, both the reviewer publications and the submitted papers are used together to build a topic model mapping word occurrences to topics. Finally, the analysis of the reviewer publications, the analysis of the submitted papers and the topic model are all used together to generate the final matchings. Analyzing the reviewer publications took about one day, while analyzing the submitted papers took about half a day. Building the topic model originally took long enough that the UTCS department killed the task as a "long-running CPU intensive job." After this happened, we moved execution of this step to an AWS EC2 Linux `t2.xlarge` instance. On the EC2 instance, building the topic model took about one day. On the same EC2 instance, generating the final reviewer-paper rankings took about four days. Our cost for running these steps on AWS was $9.27.

Autobid expects reviewer papers as well as submitted papers to be stored locally on disk rather than in Amazon S3. Additionally, Autobid expects information on reviewers in a CSV file with a variety of fields, while we simply had a text file containing Google Scholar IDs and citation IDs. To run Autobid, we needed to change the existing code to parse our text file rather than the CSV as well as pull files from S3 instead of reading them immediately from the local disk. To avoid maxing out our disk quota on the UTCS machines, we delete the local copy of the PDF file once Autobid has finished analyzing it.

In order to run the first step (analyzing reviewer submissions), we made 122 additions and 83 deletions to existing python files in Autobid. We also added 33 lines to a new python file that handled downloading files from S3. On top of these changes, running the second step (analyzing submitted papers) required only 29 additions and 22 deletions

to existing code. Running the third step (building the topic model) required 36 additions and 17 deletions to existing code. Finally, the last step (generating the bids) required no changes in regard to downloading PDFs from AWS, as its input was simply the output of the previous steps. However, it originally output separate CSV files for each reviewer, with each file in a separate directory. We instead modified Autobid to upload each reviewer's bid file to a single directory in AWS. This required only 5 additions and 4 deletions to the existing Autobid code. All of the listed changes occurred on the `master` branch.

5 Threat Model

For this project, we assume that the attacker submits a paper for publication in one of the top conferences and intends to have her paper reviewed by a favorable reviewer. We assume that the attacker has access to the papers selected for publication in the previous editions of the conference, and that the attacker has the list of names of the reviewers set to participate in the upcoming edition. Given that most conferences make their previous publications and the list of reviewers public before the deadline of paper submission, we believe this is a reasonable and realistic assumption. However, the attacker only controls her own submission to the conference which is in the form of a PDF. We adopt a restrictive threat model where the attacker can only add some hidden words to the PDF without modifying the existing content. In real scenarios, the attacker can tailor her submission's content to be adversarial as well. In this restrictive setting, we divide our task in two parts. First, we come up with a list of adversarial keywords that need to be hidden in the submission. Second, we create a module to embed a given list of arbitrary words in a given PDF without altering the rendered view of the PDF. The next two sections describe these tasks in detail.

6 Choosing keywords to embed

Autobid builds an LDA model using a provided corpus of representative publications to learn 'topics.' As mentioned in the previous section, we use the combined set of submissions and reviewers' publications as our corpus. Each topic is associated with a list of words and a probability distribution over those words. A high probability indicates that the word is highly indicative of a document belonging to the topic. For each reviewer, Autobid extracts the list of representative topics using the words present in all of the reviewer's publications. Similarly, Autobid extracts the list of representative topics for each submission. Let p_r^t be the probability that reviewer r has expertise in topic t, and p_s^t be the probability that submission s belongs to the topic t. The bid by reviewer r for submission s, B_{rs} is calculated as $B_{rs} = \sum_{t \in T^s} p_r^t \times p_s^t$, where T^s is the set of all representative topics for submission s. The bids for each reviewer are then normalized such that the bids are integers in $[-90, 100]$.

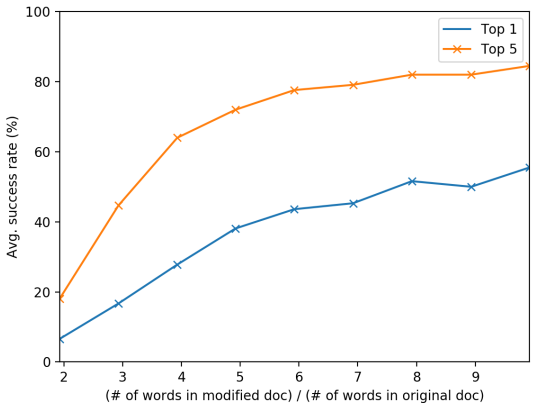
Our goal in this project is to alter a submission s such that the bid by a selected reviewer r for that submission is greater than that by any other reviewer. Hence, we hope to make our modified submission mirror the topic distribution of the favorable reviewer. For each of the top n_t topics of the reviewer, we take the top n_w words and generate a score for each distinct word. The score for word w is generated as the maximum probability of the reviewer's topic in which the word occurs. Formally, $s_w = max_{t \in T_w^r}(p_r^t)$, where T_w^r is the list of topics in which reviewer r has expertise and word w is indicative of the presence of the topic. These scores are then normalized over all words to convert it into a probability distribution.

We also tried several other ways to generate scores for each word, such as weighting the generated score by the probability of the word contributing to the topic, weighting rare topics more than those found frequently, etc. However, we found that these additional weights on the score for each word did not help with the performance. We also tried various values for n_t and n_w and found that using top three topics for each reviewer ($n_t = 3$) and top 10 words for each topic ($n_w = 10$) resulted in the best performance. Given an upper limit on the total number of hidden words, a list of words is then generated such that the distribution of words in the modified submission (containing both original and new hidden words) is as close as possible to this adversarial probability distribution.

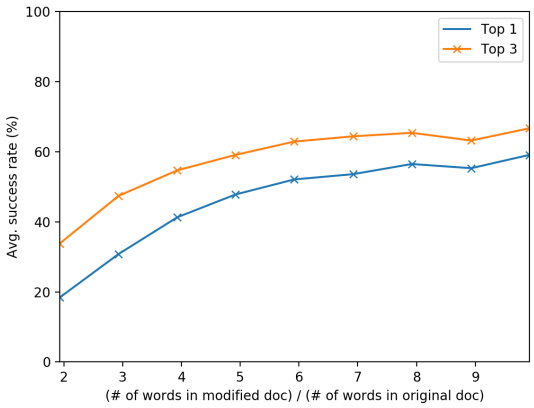
7 Embedding keywords in PDF

The open source implementation of TPMS (Autobid) [Parno 2020] uses the `pdftotext` [pdf 2020] utility for parsing text from PDF versions of the papers. Plenty of work on embedding data into PDFs already exists. However, much of this work focuses on using the PDF as a medium for passing secret messages without explicitly writing them, such as [Zhong et al. 2007] or [Lee and Tsai 2010]. To trick Autobid, on the other hand, we need to embed extra text in the PDF such that the embedded text is parsed by `pdftotext` but not rendered by GUI based PDF renderers. `pdftotext` parses the PDF's `xref` table and walks through the object tree to decode the text contents. Although Autobid does not use any additional options with `pdftotext`, `pdtotext` has a number of options to exclude text that is not readily visible, such as text outside the content bounding boxes, text drawn in PDF render mode 3 (invisible), and text with opacity less than 0.001. We therefore wanted to avoid embedding our keywords using these methods, as the developers of Autobid could mitigate our attack simply by enabling extra options when running `pdftotext`.

We decided instead to simply color our embedded text white. As it is not drawn in render mode 3 and has full opacity, `pdftotext` will not remove our keywords unless they are outside of the content bounding boxes. Therefore, we



(a) Rank of the adversarial submission in the favorable reviewer's bids.



(b) Rank of the favorable reviewer in bids for the adversarial submission.

Figure 3. Success rate of our attack when only one submission is adversarial as the size of the adversarial document is increased.

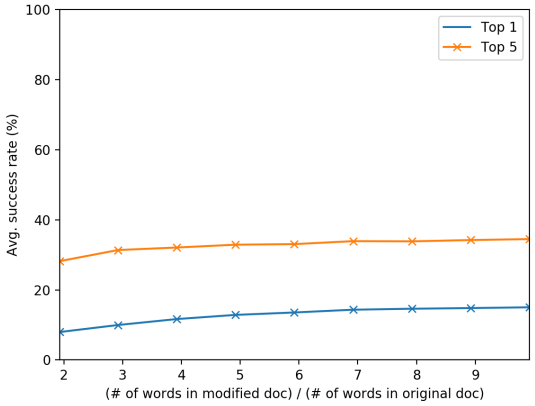
overlay our embedded keywords with the original content of the PDF. Of course, our keywords must go beneath the original content when overlaying in order for the modified document to appear the same as the original. Placing the embedded keywords behind the original words has an added benefit in that the embedding is more robust to scrutiny under text selection. If someone reading the modified document highlights a portion of the document containing our embedded text, it will appear as if they are highlighting the original text above it. The reader will need to copy their selection and paste it somewhere else in order to see the embedded text.

In order to perform the overlay, we first create a PDF containing only the keyword list generated as described in the prior section. We call this PDF the background PDF. The text in this PDF is white, so each page in the background PDF appears blank. We use the open-source library ReportLab [rep 2020] to generate the background PDF. Once we have the background PDF, we use the open-source library PyPDF4 [pyp 2020] to overlay each page of original PDF on top of the corresponding page in the background PDF. If the background PDF has more pages than the original PDF, then the keywords contained in the extra pages of the background PDF will not be present in the final modified version of the PDF. However, we expect the background PDF to almost always have less pages than the original PDF. This is because the background PDF does not have margins, tables, figures, etc., and therefore compacts the text more tightly than the original PDF. In the case that the background PDF has less pages, then the extra pages on the original PDF are unchanged in the final modified PDF.

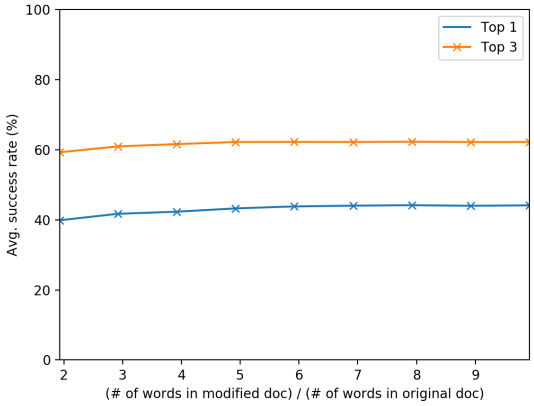
We initially tried to embed the adversarial words by adding new objects to the PDF file and hacking its 'xref' table to include the injected object in the PDF's object hierarchy. However, that approach turned out to be too complex and fragile and hence we decided to continue with this simpler method of overlaying the PDFs. Using the two previously mentioned libraries to generate our modified PDFs greatly simplified our implementation: creating a modified PDF given a list of keywords required only 67 lines of Python code. However, the trade off for this simplicity is that our tampering is easier to detect. Above, we stated that highlighting the document would not reveal the white text. Actually, this is true only if the white text is behind regular text. Our implementation does not take the layout of the original document into account and therefore places white text in locations where nothing is above it. If a reader highlights these sections of the document, they will notice that they are selecting text that seemingly does not exist and therefore will likely discover the embedded text. This problem could be fixed by writing an embedder that writes white text only to locations containing text in the original document.

8 Evaluation

We evaluate our current attack by using our system to modify a randomly selected submission in order to force TPMS into assigning the submission to a randomly selected favorable reviewer. For all the experiments, we only consider those 1759 reviewers that had at least 5 publications. Moreover, we generate our adversarial words using the top 3 topics for each reviewer and the top 10 words for each topic. We show the average of 1000 iterations for each data point in our results.

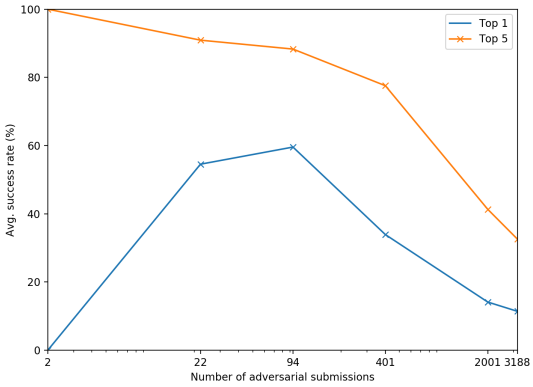


(a) Rank of the adversarial submission in the favorable reviewer's bids.

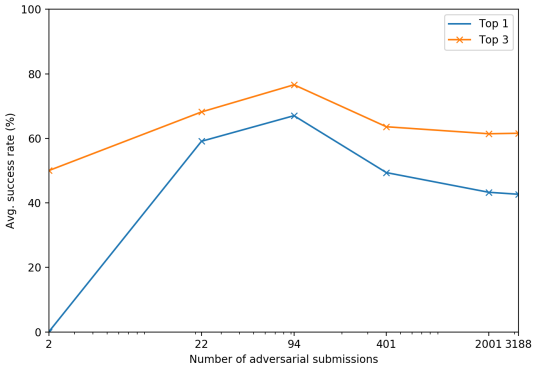


(b) Rank of the favorable reviewer in bids for the adversarial submission.

Figure 4. Success rate of our attack when all submissions are adversarial as the size of the adversarial document is increased.



(a) Rank of the adversarial submission in the favorable reviewer's bids.



(b) Rank of the favorable reviewer in bids for the adversarial submission.

Figure 5. Success rate of our attack when as the number of adversarial submissions is increased. The number of words in modified doc is fixed at nearly 4 times the number of words in original doc.

Fig 3 shows the variation in our attack's success rate as the size of the adversarial submission is increased. Fig 3a shows the percentage of trials in which the adversarial submission ranks in the top 1 or top 5 of the favorable reviewer's bids. It must be noted that since we have 3191 submissions and 1759 reviewers, and each submission has to be reviewed by 3 reviewers, each reviewer gets to work on 5.4 submissions on average. Therefore, for our attack to be successful it is sufficient that our submission features in the top 5 bids by the reviewer.

Similarly, fig 3b shows the percentage of trials in which the favorable reviewer ranks in top 1 or top 3 of the bids for the adversarial submission. Since each submission has to be reviewed by 3 reviewers, for our attack to be successful it is sufficient that the favorable reviewers' bid ranks in the top 3 bids for the adversarial submission.

As expected, the success rate of our attack improves as we include more adversarial words in the document. This is mainly because of our restrictive scenario where the attacker does not modify her paper's original content. As we include more and more adversarial words, we are able to offset the original distribution of words and tilt it towards our desired adversarial distribution. We can say that our attack is certainly successful when both the submission ranks in the top 5 bids by the reviewer and the reviewer ranks in the top 3 bids for the submission. We achieve success rate

close to 50% when the adversarial document contains nearly 3 times the number of words than in the original document. This success rate becomes close to 70% when we allow the number of words in the modified document to be more than 6 times the number of words in the original document. To put this in context, the original unmodified document never matches our success criteria in any of the numerous trials.

We also test the performance of our attack by relaxing the threat model to allow for multiple (although not necessarily collaborating) adversarial submissions. Fig 4 shows the average success rate when all submissions are adversarial. For this experiment, we choose a desired mapping from each submission to a random reviewer. Since we have almost three times as many submissions as reviewers, this setting unavoidably results in competing adversarial submissions. The dropped success rate of the adversarial submission being in the favorable reviewer's top bids is reflective of this competition (Fig 4a). However, the rank of favorable reviewer in bids for the adversarial submission is largely unaffected as expected (Fig 4b). Interestingly, in this case the performance does not vary greatly with the size of adversarial documents. Overall, nearly 30% of the submissions are assigned to their favorable reviewer as opposed to less than 1% when the submissions are not modified to be adversarial.

Finally, we run an experiment to evaluate the success rate of our attack when only a subset of the submissions made to the conference are adversarial. Once again, the submissions are not collaborating and may compete for the same reviewer. Fig 5 shows the success rate as the number of adversarial submissions is increased. As evident from Fig 5a, it is easier for a submission to rank in the reviewer's top bids when there are few adversarial submissions which implies less competition. On the other hand, as Fig 5b suggests, it is difficult to get the favorable reviewer's bid to be the top bid for an adversarial submission both when there are few adversarial submissions (it is difficult to exclusively affect bids for one reviewer) and when there are a lot of adversarial submissions (many submissions competing for the same reviewer). Hence, the best scenario for the attackers occurs when nearly 100 out of 3188 ($\sim 3\%$) submissions are adversarial. This gives each attacker about 80% chance to succeed as compared to less than 60% when they are the lone attacker. This suggests that even when the attackers are not explicitly collaborating, they help each other out as long as there are not unusually many adversarial submissions. Furthermore, it hints at the potentially disastrous results for the conference if the authors decide to collaborate and make adversarial submissions.

8.1 File Size

We were initially worried that the increase in file size between the original and modified PDFs would be an indicator of our tampering. Fig 6 shows the increase in file size as the number of words injected into the adversarial submission is increased. When the adversarial document contains approximately 10 times the number of words contained in the original document, the file size of the adversarial document increases by an average of 46 percent as compared to the original document. One reason for this is that our method of embedding text currently uses the entire area of a page to embed the adversarial words, whereas the original documents have greater margins, figures and tables. This means that the overlay PDF we use to embed the text does not require as many pages as the original document, which helps keep the overall file size lower. Another possible reason may be that because we are adding repeated words, this allows for greater compression of the text. It is easier to compress 1000 repeated occurrences of a single word than it is to compress 1000 different words.

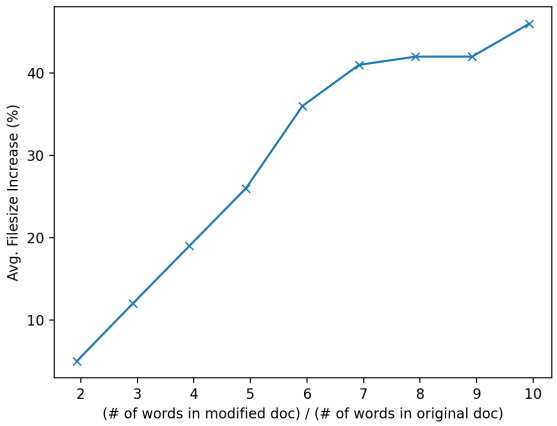


Figure 6. Average increase in file size as a function of the fractional increase in the number of words.

9 Potential Mitigations

In our current implementation, a reviewer could easily identify a modified PDF by simply using their mouse to highlight white text in an area where no text should be. However, with a more sophisticated implementation, we could embed adversarial words only behind the existing text in the unmodified document, making it more difficult for a reader to notice discrepancies in the text selection. We therefore do not see this as a valid mitigation.

In the prior section, we saw that our rate of success in tricking TPMS was approximately 50 percent when the number of words in the modified document was 3 times the number of words in the original document. At this word ratio, we see a 12 percent average increase in the file size of the modified document. For most submissions, this means the file size of the modified document should fall within any reasonable range for acceptable, unmodified submissions. Hence, mitigations based on file size are unlikely to flag modified

submissions without also experiencing some false positives on unmodified submissions. We therefore do not see this as a valid mitigation either.

As previously mentioned, `pdftotext` includes options to exclude text outside the content bounding boxes, text drawn in PDF render mode 3 (invisible), and text with opacity less than 0.001. However, as far as we are aware, it does not include an option to exclude white-colored text. This means that a mitigation based on the color of the text would require changes to `pdftotext` or the use of a different PDF parser that does provide such an option. To prevent a variety of methods of embedding text, the parser should ignore any text below a certain font size, any text without full opacity, and any text that is not colored black or nearly black. These restrictions should prevent visually hiding embedded text.

In our current approach, we embed a few adversarial keywords multiple times (repeated continuously to drive the word occurrence probability to a favorable value) in the submission. Thus, one other mitigation can be to analyze the parsed text from the submitted PDF for unusually high number of repetitions for a few words. This heuristic may be used to flag some submissions as potentially adversarial which can then be manually verified.

10 Future Work

As described earlier, our implementation of embedding text is naive to the internal structure of PDF documents. A first step toward improving the embedding process would be to take the format of the original document into account so that adversarial text is only embedded behind existing text. Additionally, it should be possible to create a document with the original text written in a font that is missing a `ToUnicode` table. Text written in such a font will not be extracted by PDF parsers. Adversarial words can then be included in the main text of the document in a regular font. The document will render normally, but only the adversarial words will be extracted by PDF parsers. The author of the submission can attempt to work the adversarial words into the actual text of the document in natural places. Because they are the only words extracted from the document, the author of the submission does not need to include the adversarial words multiple times, but the desired reviewer should still be highly ranked. This process would subvert all mitigations raised in the previous section. However, such a process could not be done fully automatically, as the author of the submission must manually include the adversarial words.

Currently we choose our adversarial keywords based on the topic and word probabilities generated by the same LDA model that is used to generate the bids. It would be interesting to use an approximation of the true model that is generated using a corpus of only reviewers' publications to generate the adversarial words. Moreover, applying our attack on a system based on a different model (e.g. StarSpace [Wu et al. 2017]) will be a good test of the generalization of this approach. In our current approach, we add a few adversarial words multiple times which may make it easy to mitigate our attack. One direction of future research can be to obtain successful attacks with an upper limit on the number of times a particular word can be repeated.

11 Conclusion

The assignment of papers to reviewers is a difficult problem facing conference organizers. In this paper, we show that it is possible to modify a paper submission in such a way as to force TPMS, a popular solution to this problem, to pick arbitrary reviewers for the paper. Specifically, we embed keywords into the PDF so that there are no visible changes to the document but TPMS determines the paper is related to the desired reviewer's area of expertise. We show the effectiveness of our attack through various experiments, and find that an attacker can get her paper reviewed by a favorable reviewer with a probability as high as 80%. The implementation of the attack described in this paper is mostly naive to the internal structure of PDF documents and could therefore be defeated by a few simple mitigations. However, more complex attacks based on the same principles but taking the internals of the PDF format into account could be more difficult to mitigate.

References

2019 (accessed Mar 11, 2020)a. *NeurIPS 2019*. https://nips.cc/Conferences/2019/Reviewers

2019 (accessed Mar 11, 2020)b. *NeurIPS 2019*. https://nips.cc/Conferences/2019

2019 (accessed Mar 11, 2020)c. *NeurIPS Proceedings*. https://papers.nips.cc/

2020 (accessed Apr 13, 2020). *PyPDF4*. https://github.com/claird/PyPDF4

2020 (accessed Apr 13, 2020). *ReportLab*. https://www.reportlab.com/opensource/

2020 (accessed Mar 18, 2020). *pdftotext*. https://www.xpdfreader.com/pdftotext-man.html

Laurent Charlin and Richard Zemel. 2013. The Toronto paper matching system: an automated paper-reviewer assignment system. (2013).

Diego Charrez. 2019 (accessed February 17, 2020). *NeurIPS 2019 Stats*. https://medium.com/@dcharrezt/neurips-2019-stats-c91346d31c8f

I-Shi Lee and Wen-Hsiang Tsai. 2010. A new approach to covert communication via PDF files. (2010).

Bryan Parno. 2018 (accessed Mar 11, 2020). *Autobid*. https://github.com/parno/autobid/tree/master

Ledell Wu, Adam Fisch, Sumit Chopra, Keith Adams, Antoine Bordes, and Jason Weston. 2017. StarSpace: Embed All The Things! arXiv:cs.CL/1709.03856

Shangping Zhong, Xueqi Cheng, and Tierui Chen. 2007. Data Hiding in a Kind of PDF Texts for Secret Communication. (2007).