# Critical Analysis: A 20-Year Community Roadmap for Artificial Intelligence Research in the US[1]

Eric Kuha Jake Waro

2020 February

## 1 Problem Statement

Research into artificial intelligence, as it is currently practiced, is not unified in its purpose. There is no underlying framework upon which future research can and should be conducted. The major areas that are lacking are several. Building a clear, coherent plan for the future of artificial intelligence research can pave the way for future developments in a way that will benefit academia, industry, and the consumers who would be the end users of many of the applications of this research.

This roadmap will call for a large-scale effort coordinating the efforts of academia, industry, and government to build a national infrastructure for artificial intelligence research. Furthermore, the roadmap will encourage participation of historically underrepresented demographics and incubate and fund the creation of testbeds and the research institutes dedicated to AI research. It is vital that all of this be carried out with society's best interests and ethics in mind.

## 2 Major Contributions

The paper's outlines a 20-year roadmap for research into artificial intelligence. It identifies a lack of preparation and foresight in the current state of AI research within academia as well as industry. It recommends a unified vision for how artificial intelligence research can be conducted in a safe, productive, and ethically- and socially-conscious manner.

The paper first identifies primary driving principles that must be observed:

- Security and vulnerability
- Ethics
- Resources

Then, the primary recommendations for coming decades of research fit into three broad categories:

- National AI infrastructure
- Core AI Programs

- Workforce Training

From there, the paper outlines specific projects, initiatives, and policy suggestions for driving AI research in the coming decades. For example, emphasizing interdisciplinary AI, the formation of national research centers, open platforms, encouraging AI curriculum, and engaging underrepresented groups.

Of particular note is the very explicit declaration that a need for interdisciplinary partnerships in AI research is paramount. Encouraging collaboration between the fields of AI research and the humanities, medical and social sciences, and law are suggested interdisciplinary paths. After all, what is AI research actually looking at if it is not humans and the things that they do?

# 3 Validation Methodology

As a call-to-action paper, its method of validation is a little circumspect. Citing the broad success of large-scale initiatives in the past is one major method of validating the claims in the paper. For example, they cite cases of how "substantial long-term investments drive significant advances." Specific examples include the Apollo Program, the Hubble Space Telescope, the Human Genome Project, and the LIGO Gravitational Waves project.

Another method in which the paper emphasizes the important work being performed in the field of AI is a series of (fictional) vignettes about a wide variety of places where artificial intelligence could assist actual humans in daily lives. For example, a man diagnosed with Alzheimer's disease might have a personal digital assistant that can remind him of current activities, scheduled events, and who people in his field of view are, by using augmented reality technology, machine learning, and other new technologies. If technologies like these became widespread, the overall quality of life for many people could be enhanced significantly. Case studies like these highlight the human side of AI research, which has a significant role as the driving force behind the entire roadmap.

# 4 Assumptions

There are two major assumptions that the paper makes. First, AI research is going to happen. And second, it is going to be transformative.

The paper also assumes that the resources, both in funds and human capital, are available. This is reasonable as there are high-paying jobs and much capital investment in this field already.

AI research is going to be a major draw for students, engineers, and academics as well as government dollars for the foreseeable future.

The paper asserts that a transition to a cooperative model between the government, academia, and industry will be seamless. There is historical precedent for this assumption. For example, the development of the Internet and its infrastructure was a cooperative enterprise between all of these entities. The national telephone system before it was also largely subsidized by federal grants, facilitated by industry, and encouraged and researched by academics and engineers.

Careful prioritization of AI research is of utmost importance to *all* people. It is important that it be well-funded, and the necessary infrastructure be cultivated, including research centers, staff, students, and professors. It is also of vital importance that this framework emphasizes partnerships between academia, industry, and government.

## 5 What would we change?

The paper makes very strong arguments in favor of adopting a particular set of priorities, research directions, and policies toward artificial intelligence research. It also makes it very clear that issues of ethics and privacy are of central importance to any research that is conducted, any platforms developed, and any institutes that are formed. We would have liked to see some language outlining the potential policies for sanctions (and who is responsible for enforcing them) for any organization or individual who can be demonstrated to fail to adhere to high standards of respect for privacy and ethics. Especially when such a technology has a high potential for abuse by those in the positions of power, namely, the people and organizations developing those technologies.

Another thing that was never really addressed in the paper is the process for adopting the roadmap. There is little or no discussion about obstacles that exist that might prevent this roadmap from becoming a national priority. Presumably, there are several people who are currently making the rounds at conferences, campuses, and corporations evangelizing for this roadmap in an effort to convince as many people as possible to see the benefits of a coherent plan for the future of AI research. A description of these efforts is potentially warranted.

## References

[1] Yolanda Gil and Bart Selman. A 20-year community roadmap for artificial intelligence research in the us, 2019.

# End-to-End Encryption for Printers: Proof Of Concept

Eric Kuha
kuhax005@umn.edu
University of Minnesota
Minneapolis, Minnesota

Jacob Heppner
heppn028@umn.edu
University of Minnesota
Minneapolis, Minnesota

Wamani Mbelwa
mbelw002@umn.edu
University of Minnesota
Minneapolis, Minnesota

## ABSTRACT

This paper explores of a proof-of-concept protocol for enhancing the security of printing devices with an end-to-end RC4 encryption protocol using PostScript on PostScript-enabled printers. The protocol utilizes the notion of self-decryping files to create a sidechannel into the printer protocol which we use to enhance the security of printing. This includes a key-exchange protocol as well as an encryption algorithm which will allow making hard copies of documents where the transmission of the data of the document is performed while the data is encrypted.

## CCS CONCEPTS

• **Security and privacy → Information flow control**; *Key management*; *Embedded systems security*; • **Hardware → Printers**.

## KEYWORDS

printer security, encryption, vulnerability mitigation

**ACM Reference Format:**
Eric Kuha, Jacob Heppner, and Wamani Mbelwa. 2019. End-to-End Encryption for Printers: Proof Of Concept. In *Proceedings of UMN (CSCI-5271).%* , 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Since the beginning of the digital age, digital security has been the focus of much intense research. And indeed, strong and reliable encryption protocols for all-digital communication are accessible, easy-to-use, and nearly ubiquitous. However, in the digital-to-analog conversion of a document to a hard copy (i.e. printing), there have long been considerable security issues. This is a discussion of a plausible proof-of-concept of a do-it-yourself encryption protocol for securely rendering hard copies of documents using PostScript-enabled printers.

Some strides have been made to increase security in printing, however, there is much work to do. For example, it is not unreasonable to expect that a networked printer have some form of end-to-end encryption, especially if this is a vulnerable network, such as a public network at a library that allows printing, or some other space in the commons. Yet, it is the case that for nearly all older printers and many new printers, all communication happens in plaintext over unencrypted channels [12].

For older printers or on printers that do not have encryption implemented by the company's drivers, there has not been any way for the typical user to ensure that sensitive information that needs to be printed cannot be spied upon by anyone listening in on the typical ports that printers use for receiving print jobs. The goal of our project was to enable confidentiality in printed documents during transmission to these vulnerable printers. Our method takes inspiration from a tool commonly used by the adversaries of anti-virus software to subvert detection, called self-decryption. Rather than using self-decryption to subvert detection, we plan on using self-decryption to enable a new security mechanism where updating the firmware is either not possible or not desirable.

The protocol that we introduce is capable of allowing end-to-end ecryption of the documents. For this to occur, we first show that it is possible to generate a key, save it to the memory of a printer, and transmit it to the person trying to print sensitive data by printing the key. We then are able to take the key, and create a self-decrpyting file that is described using PostScript. This file is then transmitted to a printer which finally prints the data in the input text file.

## 2 BACKGROUND

Some substantial work has been conducted to establish with little doubt that printers are a security issue [11]. And to a great extent, these issues continue to exist. Significantly, there exists a software PRinter Exploitation Toolkit (PRET)[10], which is a general-purpose toolkit, written in Python, for identifying and exploiting issues on printers, including new ones. It can exploit printers using the Printer Job Language (PJL), Printer Command Language (PCL), and PostScript. We chose to focus on PostScript as the language of our focus because it is older, more mature, and nearly ubiquitous, existing on nearly all printers, even new ones, despite its many inherent security issues.

Indeed, the PRET project also lead to the creation of a wiki devoted to the topic of hacking printers [2], which has a wealth of information about the types of printer vulnerabilities that exist, what brands of printers tend to be susceptible to which ones, and how to exploit them. There are two interesting things about the website. First, it contains a wealth of information about all aspects of exploiting insecurities of printers. And second, it apepars to show almost no real traffic or community developing it more. This can lead us to one major conclusion: after an initial burst of interest and activity in the subject, the vast majority of people, even in the industry, have a dismayingly cavalier attitude toward the security in networked printers.


*CSCI-5271, Spring 2019, Minneapolis, MN*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

A shocking example of this casual attitude toward security in networked printing is described in the article "Hacking network printers" [5] by Adrian Crenshaw. With the right Google (yes, Google) search string, one can find any number of printers which are theoretically supposed to be isolated to a corporate intranet, but which are exposed to the internet at large. Plenty of secret information can be had just by looking at the print history on a supposedly protected printer. A little JavaScript and one of these networked printers can be induced to print arbitrary documents, run arbitrary code, or even be scanned for incoming jobs [20].

Our goal is to use the inherent security issues that exist in any printer that implements PostScript to our advantage to be able to print securely (if not beautifully), a message that is more difficult to intercept.

## 2.1 PostScript

The core tool in our for our project is PostScript, the document description language developed by the Adobe Corporation and the precursor to Adobe's Portable Document Format (PDF) [18]. While older and less supported than PDF, PostScript is in some ways more powerful than PDF since it has control flow mechanisms such as "loop" and "if" which were removed during the creation of PDF to simplify the language. Additionally, PostScript is a Turing complete language. This means that with enough ingenuity we will be able to write any program that we need. While Adobe has been pushing for a migration away from PostScript to PDF, either primary or legacy support for PostScript exists on almost every printer, which makes it an optimal language choice for implementation.

PostScript is a stack-oriented, interpreted language. Few other languages are stack-oriented, which differenciates PostScript from other languages with which most people are familiar. The language is written in postfix notation, and all named variables have global scope. These factors combine to make implementation of a program using PostScript a very interesting experience. The coding style of PostScript is very similar to many functional programming methods. PostScript is different in that it discourages the use of named variables. By having all variables given global scope, too many named variable create the risk of namespace pollution and accidentally overwriting the variables used in other procedures. This makes it necessary for the programmer to pay significant attention to their stack-hygene. If the procedure has the chance of ending in an unexpected way, then there is significant risk of corruption of the final document. Additionally, there is an emphasis on understanding stack manipulation, since too many named variables pollute the namespace. Thus it is typically better to push the variable onto the stack as an "anonymous" variable and then keep track of where it is so that it can be later retrieved.

Postscript procedures differ slightly from functions as programmers typically think of them. Rather than copying parameters or taking in references to parameters, PostScript procedures are effectively pushed onto the stack. The difference is that while most methods essentially operate in a different space than the body from which they are called, a PostScript procedure is really just a shorthand for typing out the text exactly where it was invoked.

Another property of PostScript which enhance the prospects of self-decryption is that the language has procedures for changing data into executable strings, which is made possible by the fact that it is an interpreted language [17]. While uncommon, systems have enabled the ability to convert executable instructions to data and back again since the Selective Sequence Electronic Calculator (SSEC) produced by IBM in 1948 [4]. This cabability enables PostScript to produce self-decrypting files that will be encrypted up to a level where only the key retrieval and the encryption algorithm are unencrypted.

## 2.2 RC4

The cipher that we chose to use for this project was RC4. RC4 is a stream cipher which is performed in two steps. The first step is the Key Scheduling Algorithmi (KSA). The algorithm starts with an array of 256 bytes where the value of the byte in the array is equal to the value of its index. From this initialized array, the program uses the key along with the value of the elements in the array to help shuffle the array. After shuffling the array of bytes in the key scheduling algorthm, the Pseudo-Random Generation Algorithm (PRGA) produces bytes of with a pseudo-random distribution. During the process of producing these bytes as output, the pseudo-random generation algorithm continues to mix the values in the array so that no matter the number of bytes produced by the algorithm, the output continues to be pseudo-random.

While RC4 was generally considered to be a good algorithm for a long time, modern alanyses of the algorithm have shown that the algorithm has biases which can amount to ciphertext-only plaintext recovery attacks [3] when the first 256 bytes of RC4's output are used. In our implementation of RC4, we addressed this simply by added a loop which discharges the first 256 bytes of output of the PRGA to help mitigate this bias.

More recent analysis of RC4 in the context of WPA-TKIP and TLS has shown that there are longer-term biases which are inherent in the algorithm [19]. While this is a serious vulnerability in the RC4 algorithm, leading to the recomendation that TLS no longer be allowed to negotiate to use RC4 [13], the reason that this long term vulnerablilty can be used is that a TLS connection can be manipulated to encrypt many indentical bytes. Since our algorithm never reuses keys, and the fact that the number of identical bytes was around $9.5 * 2^{20}$. This number of identical bytes would only occur in printed documents with an incredible infrequency for normal printed docuements, meaning the likelihood that this issue lead to a comprmise of security is very low.

## 2.3 PRET

Another important tool in our search for understanding of PostScript and printer vulnerabilities is a program call the PRinter Exploitation Toolkit (PRET) [10] [11], which was presented at the 2017 IEEE conference. This fascinating tool simplifies the process of vulnerability testing a printer. It opens a channel to a networked printer by IP address and can communicate via most standard printer languages. In particular, it allows us to determine if a particular printer has specific vulnerabilities that allow us to, for example, set environment variables, or write to the filesystem on the device, as well as offer up insights to just how to exploit certain printer vulnerabilities to create the encryption protocol.

## 2.4 Printer Hardware

We had access to several actual hardware printers. Of note was a Brother DCP-L2540DW series laser printer/scanner, and a HP Color Laser Jet (insert model number here).

The Brother laser printer turned out to be easy enough to interface with and interact with the filesystem, but, unfortunately was not write-enabled, so writing to the filesystem (for key storage purposes) was not possible. That said, environment variables could be set, but new environment variables could not be created, which was a limitation that made this device somewhat unsuitable to our needs.

The HP Laser Printer, on the other hand, was a somewhat older networked printer that allowed us to read and write files to its filesystem. This was a more viable route to being able to create a key, write it to the filesystem, and then use it for decryption at the printer endpoint.

## 2.5 Printer Protocols and Languages

There are a variety of ways in which a printer can be networked with other machines. So it is important to understand the underlying assumptions about the network and its strtucutre. This protocol was largely tested locally using GhostScript, however, it was shown to work on one HP Color LaserJet 4600 series laser printer. This base of the printer's protocol stack is the Simple Network Management Protocol (SNMP) [9], which based on UDP.

The HP 4600 uses the Line Printer Daemon (LPD) as its primary printing channel. This protocol handles print job queue management and other bookkeeping on the printer. We used CUPS [1], the open-source printing system from Apple to communicate with the LPD on the printer. Nearing the top of the stack, we have the Printer Job Language (PJL), a language originally developed by HP to oversee printer management. We did not communicate directly with this aspect of the printer, though it is involved directly in job management at a high level on the printer.

Finally, PostScript is at the top of the stack, and this specifically what we targeted as our main vector for implementing this protocol.

## 2.6 GhostScript

For testing the protocol, the open-source program GhostScript, a GNU open-source PostScript interpreter was invaluable for testing, as it behaves like a printer, rendering the output of a PostScript program to the screen just as a printer would, but doesn't use nearly as much paper. It also has an interactive interpreter which allows for line by line debuggging of the program, with the ability to see the stack state and values of variables.

## 2.7 Secure File Transfer

There has been extensive research on the ways in which files can be transfered securely. Over time, there have been many propsed protocols for file transfers with only a few now dominating the majority of file transfers. Among these protocols are SFTP and HTTPS [15]. These protocols are defined on top of SSH [21] and TLS [6] respectively. These protocols negotiate the key exchange and encryption algorithm as part of the setup. Whether by public key cryptography or Diffie-Hellman key exchange, the two programs will coordinate to generate a session key which will be used

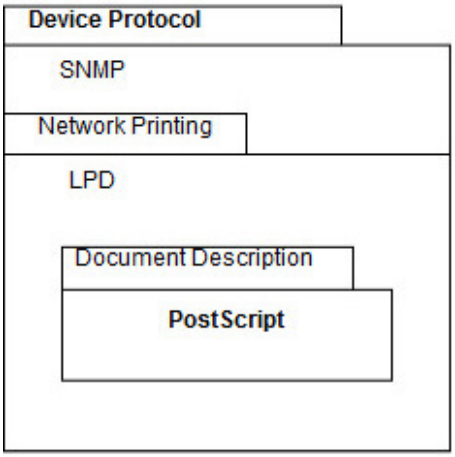


**Figure 1: Diagram of the layout of our target print protocol stack**

to encrypt all future communication between the two communicating programs. It is noteworthy for the purpose of this paper that these protocols require that a dedicated program be in charge of encrypting and decypting the data that will flow between the communicating programs. This method can be distinguished from other kinds of code which are in charge of modifying themselves using cryptographic techniques rather than have another program perform the modifications.

## 2.8 Self-Modifying Code

Self-modifying code has had multiple uses in the context of security, both in attacking systems and in the defense of confidential information. The premise of self-modifying code is that the transmitted document contains all of the details needed to extract the confidential information besides the key. This allows for the transmission of confidential information to a recipient who already has the necessary key without the need to install any software dedicated to the decypting the document. Attackers also commonly use self-modifying code to obfuscate thier code to prevent automated recognition of their attack. Finally, there is a branch of self-modifying code which is dedicated to creating self protecting code. This code uses conditional code execution to produce different results with the aim to protect the intelectual property of an organization.

*2.8.1 Self-Modifying Code as Attack.* Self-modifying code comes in many different forms. The general approach to malicious self-modifying code is to obfuscate the workings of the code either to prevent security researchers from reverse engineering the code to stop the attack, or to prevent automated virus detection software from creating profiles that would recognize the malware. An example of code obfuscation using cryptographic techniques is the paper which introduced the concept of Conditional Code Obfuscation [16]. The idea behind this approach is to use cryptographic techniques to create conditional statements which are logically equivalent to the unencrypted condition. The work presented in the paper "Impeding Malware Analysis Using Conditional Code

Obfuscation" uses an automated system at the compiler level to change conditional statements into these logically equivalent forms to produce obfuscated binaries.

While this approach is certainly effective, an approach to self-modifying code that is far more compelling is that taken by the virus known as W32.Simile [8]. This program checks the date to determine if it should display a political message, and if it is displayed, the binary automatically rebuilds itself, and then potentially infects other executable files. This process is capable of both shrinking and expanding the size of the binary allowing the size to change, without rapidly expanding. This kind of approach is designed to prevent the virus software from being able to develop automatic fingerprints for the virus.

*2.8.2 Self-Protecting Documents.* With the introduction of Conditional Code Obfuscation, the process for automating the obfuscation of branching statements has been understood. This work was expanded upon to produce a different kind of security than is the main focus for this project. Self-protecting documents modify many branching statements to obfuscate the conditions which would trigger the introduction of watermarks [14]. This process would allow an author to publish works in way which would make it unlikely that another person could illegally incorporate the work published by the defending author without including the watermark that the defending author wishes to incorporate.

This process was developed to help protect authors from code repackaging, the process of coping large portions, if not all, of the code of a project into another project and simply republishing. This process was believed the be threatening the health of the Android software ecosystem. With the introduction of self-protecting software, authors could enable defenses to their software apart from appealing to authorities such as the maintainers of the Andriod market.

*2.8.3 Self-Decrypting Documents for Confidentiality.* Self-decrypting documents are a potentially powerful tool for transfer of documents when it cannot be guaranteed the reciepient will have the software to decrpyt the transmission. Self-decrpyting documents for the purpose of confidentiality are a type of encrpytion distinctly differnt from other more common types of encrpytion because of the asymmetry in the implementation. Protocols like TLS and SSH require that the implemetation of the protocol exists on both sides of communication. In fact, parts of the protocol are nagotiated between the two systems to ensure that they are both only using a version they both understand. Self-decrypting documents rather are an almost entirely asymettric implementation. The sender must have the knowledge of the entire protocol and algorithm which will be used during encryption and decrpytion.

There are practical problems with self decrpyting executables though. A man-in-the-middle attack, or a trusted but not trustworthy source of a self-decrypting executable would be able to get arbitrary code to run on the recieving machine. Further, there are compatability problems. The sender would need to compile the executable for whichever machine would recieve the communication. For these reasons, it is not common to recieve this kind of communication.

While these issues are true of self-decrpyting files of all kinds, printing is exceptional in that self-decrpying files are specially suited to the limits imposed by the system. One of the biggest issues layed out for self-decrypting files is that it requires that senders of files be given freedom to run arbitrary code. While this is a major vulnerability, it is the modus operandi for all PostScript documents. PostScript, being Turing-complete can already run all arbitrary code, and printers are designed to simply interpret any PostScript files which are presented to them. For this reason, no additional vulnerability is added to the system by requiring that the self-decryping file is given aribitrary code execution privilages. Further, it seems appearant that it is beyond the realm of possibilities to request from manufacturers of printers to release new firmware for their old printers which supports modern cryptographic protocols. Even if they would release such firmware, it would not be possible for the average user to update such firmware on any given network printer they wish to print upon. This requirement enforces the constraint that any such cryptographic primitive be an asymmetric implementation. These constraints conspire to make self-decrypting files be the perfect solution to adding encytion to the network traffic between a computer which needs to print a document and a network printer which does not already support a cryptographic protocol.

## 3 THREAT MODEL

The threat that we are trying to protect against is a passive listener on the network that is listening to all communication travelling between computers controlled by people who want to print and unsecured printers. We are trying to preserve confidentiality in systems like these. Since most network printers, especially older network printers, take all communication unencrypted, there would be a significant threat that a person may try to listen in on this communication to try to scrape sensitive information that would need to be printed. This attack would be about as easy to perform as installing a listener on all wifi communication on public wifi, which is already a common technique for aquiring sensitive data.

Our threat model does not include a person with a more active attack. The reason for this is that there is no way for us to protect the key from the attacker, as our ability to save the key to a file is greater access than a person just trying to read the key file would need. While this is not the strongest guarantee for confidentiality, we believe that it addresses most of the significant concerns. It is most likely that an attacker would passively listen to the network traffic and only later do any analysis on the traffic, by which time the required key would already have been deleted. For an attacker to otbtain the key without consisitantly connecting to the printer, they would need some method to record the contents of the actual sheets of paper printed from the printer, which could be observed by the person trying to confidentially print. Furthermore, if the attacker could access the file containing the key from the key exchange, that person would then also be able to observe the plaintext of the final docuement. Therefore, the protocol could not add any security to a situation where the attacker had physical access to the printer during the print process.

## 4 PROTOCOL

The original goal was to create the entire protocol (key exchange, encryption, decryption, and printing) in PostScript because many

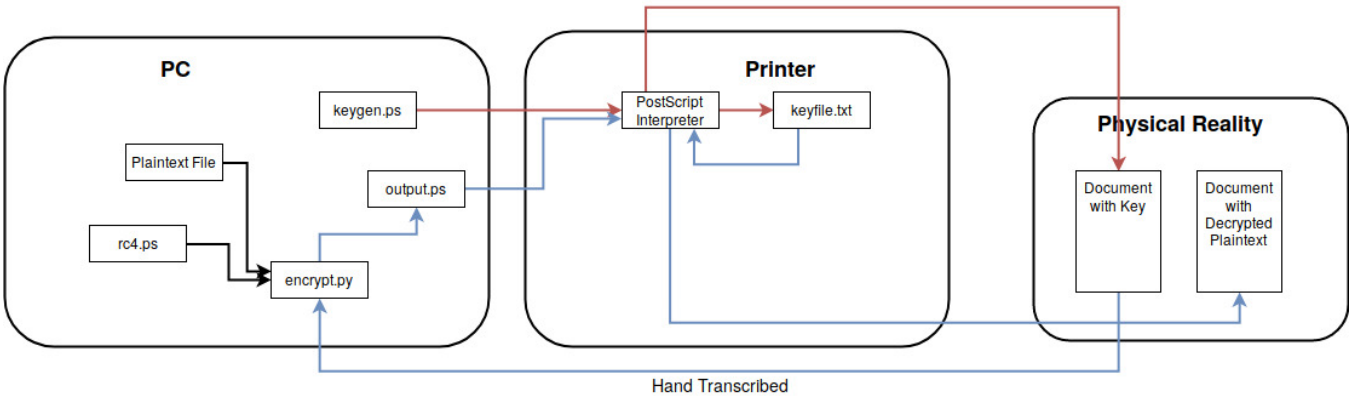


**Figure 2: Basic flow of the protocol**

printers still describe documents in this language. During implementation, we discovered that certain parts of the protocol would be easier to implement using other methods. Key exchange is notable because passing the key back to the encrypting algorithm is a very difficult problem, since printers are designed specifially for recieving messages, rather than for passing them out. For this reason, PostScript has little access in the way of controlling networking. While the encryption process can easily be performed through PostScript, since encyption and decryption are symmetric using RC4, the process of merging files and performing formmating is much easier in a language like Python. For this reason, the encryption process occurs in Python, which merges the proto-file and the desired text in the input text file to create the output file. The output file is then a self-decrpying file which contains all the information needed to retrieve the key from the printer, decrpyt the data contained in the file, and finally print the document.

For the purposes of this project, we decided to use a stream cipher. There are a few advantages to using a stream cipher for this project. The first is that the specific stream cipher can be changed without modifying the entire protocol. While we chose to use the stream cipher RC4 in our project, it would be only as difficult as implementing another stream cipher and dropping it in place to use the other stream cipher in our protocol. The second is that it allows for the ability to have more flexible encryption. While our project only encrypts the data, further improvements to the project may add encryption to the formatting of the document. As long as the encrytion of the document happens in the same order as the decrytion, slightly different implementations may pick and choose which parts of a document they feel need to be protected. The final advatage is the symetry between encryption and decryption. In other encryption schemes, such as AES, encryption and decryption are different processes. This would mean needing to implement two separate processes, which would more than double the work needed to get the process working. By implementing a stream cipher, we knew that once encyption was working, decryption would work as well.

The reason that RC4 was chosen as the cipher is its relative ease to implement. The RC4 implementation is very easy as compared to something more modern such as AES. While much easier to implement, it has a few known vulnerabilities, which is why it has fallen out of favor [19]. Knowing this, we still decided to implement RC4 due to the reasoning above. We felt that as a proof of concept, and the ease with which it is replaceable, the easier implementation was warrented to allow for us to focus on other aspects of the protocol to ensure the best overall result. Even though we knew that there were vulnerabilities in RC4, we feel like it still provides a reasonable level of security. We intetionally exclude the first $2^8$ bytes from the stream. By excluding the first $2^8$ bytes from the stream, we were able to make the biases from the short-term biases negligable. We felt that for nearly all reasonable sized documents which may be printed, the streams would not produce enough bytes to create enough corrolation for a key extraction like was performed in the paper "All of Your Biases Belong to Us" [19]. The reason for this is the immense number of identical packets they needed to collect before the biases became significant enough to extract the packets. We believe that there would need to be a truely exceptional print before RC4's biases would become much of a problem for an individual print, and according to the algorithm, the key will change between each print. This should offset most of RC4's major issues with biases.

The protocol works like this:

(1) A PostScript program is sent to the printer for execution which uses the printer itself to generate a pseudorandom key.
(2) This key is saved to a file on the printer's filesystem, and the key is printed as a hard copy which can be retrieved by the user.
(3) They key is used to encrypt the plaintext message.
(4) The cyphertext message is used as input for another PostScript program which is sent to the printer, which uses its stored key to decrypt the message and print the plaintext.
(5) The file where the key is stored is deleted, and the protocol is complete.

## 4.1 Key Exchange

The key generation and exchange protocol takes place entirely using a PostScript program. Using either GhostScript (to test locally) or CUPS to send it to a printer with the correct vulnerabilities

(i.e., file read-write). The keygen.ps script generates 16 random bytes, seeded from the system time and the program user time added together (there are obvious entropy issues here, but there are precious few sources of entropy on printers specifically and embedded systems generally).

```
% Generate Random Bytes
16 string
0 1 15 {
    1 index exch
    rand 255 mod put
} for
```

These bytes are written directly to a file on the filesystem (printer or local) and then printed as hex characters (0-9a-f) on a sheet of paper or GhostScript output. You'll notice here that *key* and *outstring* are two different variables for this purpose.

```
% Output to file Called "keyfile.txt"
/keyfile (keyfile.txt) (w) file def
keyfile key writestring
keyfile closefile

% Print to page for user to see
/Helvetica-Bold 12 selectfont
72 200 moveto
outstring show
```

The user now has a copy of the key and so does the printer.

## 4.2 Encryption/Decryption

The user next uses a text file with the desired message as the plaintext input for a python script. The user places together the encryption script, the input file, and the proto-PostScript file which contains the implementation of RC4 as well as some basic utilities. Then the user calls the encryption script, implemented in Python3. The user will then be asked to type in the key which was generated by the printer and printed out for them. This key is used to encrypt the data in the text file and the cyphertext is appended to the postscript file. During encrpytion, the encrpyting script will insert formmatting notes which will be used by the printer's interpreter to print the document to match the stucture of the input text document.

```
def PRGA(plain):
    i = 0
    j = 0
    cipher = plain
    for c in range(len(plain)):
        i = (i + 1) % 256
        j = (j + stream[i]) % 256

        swap(i,j)
        rnd = stream[(stream[i] + stream[j]) % 256]
        cipher[c] = rnd ^ plain[c]

    return cipher
```

## 4.3 Printing

The output document generated by the encryption script is then sent to the printer. The printer recieves the file and begins interpreting the file. The first instruction will be to retrieve the key from the printer. With the key in memory, it will then initialize the RC4 algorithm, and then procede to decrypt the data and print the document as if it were any other file. Ideally, it will then also delete or overwrite the keyfile. The protocol does not allow a lot of sophisticated formatting. It will be exclusively plaintext, however, linebreaks and other basic layout will remain. With this the protocol is complete.

```
% Pseudo-random generation algo
/PRGA { % array PRGA array
    /cipher exch def
    /plain cipher length array def

    /i 0 def
    /j 0 def
    0 1 cipher length 1 sub{
        /c exch def
        /i i 1 add N mod def
        /j j stream i get add N mod def
        stream i j swap
        stream i get
        stream j get
        add N mod
        /rnd_ind exch def
        /rnd stream rnd_ind get def
        /new_val rnd cipher c get xor def
        plain c new_val put
    } for

    plain %return
} def
```

Figures 3 and 4 display the results of sending each of the two print jobs to a printer. The user of our protocol would first retrieve a document which looks like that displayed in Figure 3 and use the key printed on the document during the encrpytion phase. The encryption phase of the protocol will then generate the self-decrypting file which will produce output similar to that displayed in Figure 4 matching the input text given by the input text document.

# 5 SECURITY ANALYSIS

While really any gains in the security of printed documents is an advancement within most of the network printer space, there is a serious desire to add as much security as possible. For the current version of the proof of concept, there were some known tradeoffs taken in the implementation, and there were some forms of attack that this method could not address.

The first and most obvious vulnerability in our approach is the use of RC4 as the encrpyion algorithm. While RC4 is very easy to implement, it has too many faults to be still considered a strong

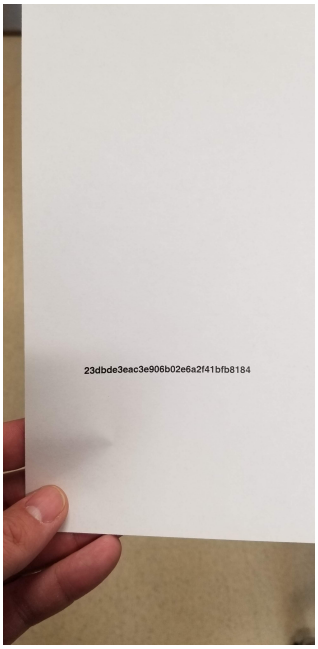


**Figure 3: Printout of key**

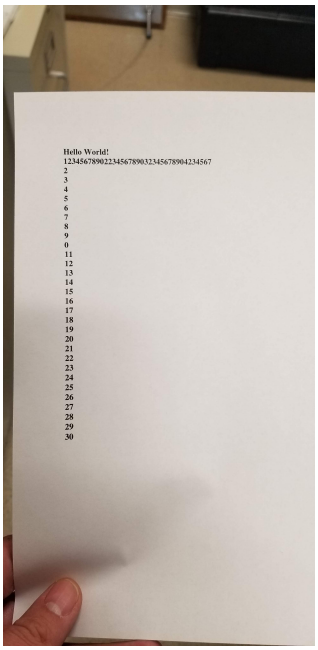


**Figure 4: Printout of final resulting plaintext**

encrytion algorithm. We intentionally made the tradeoff of implementation speed to vulnerablility by selecting RC4. Yet, it would be a relatively simple process to replace RC4 with another stream cipher, such as AES in counter mode. This would add both a significant amount of complexity to the implementation as well as a significant improvement to the security of the transmission.

The second problem with our approach is that it is not currently possbile to protect the key file from an attacker. The process of extracting the key could be made more difficult by randomizing the name of the key file, but the name of the key file would need to be transmitted along with the key. This would at least make it more difficult to isolate where the key is located for an automated process, but it cannot entirely fix the issue, as an attacker may identify the key file in another way.

We however believe that we add significant amounts of security against the threat model. We believe that our protocol restricts access to confidential information in printed docuemnts over unsecured networks well where the attacker would not have physical access to the printer to spy on the finished document.

## 6 FUTURE WORK

The work that was accomplished by this project is ripe with areas for future work to extend the capabilities of this project. As has already been talked about, there is room for improvement in the encryption algorithm used. While RC4 offers a modest amount of protection, a more modern encryption algorithm like AES could replce RC4 as the stream cipher. Beyond that, there is the opportunity to extend the scope of what is encrypted in the document. Currently, formatting, custum fonts, and other file details are not encrypted. Since PostScript contains functionality to turn a normal string and turn it into an executable string, similar to a procedure, essentially everything besides the command to retrieve the key and the implementation of RC4 could be encrypted. While we did not have the opportunity to extend the encryption to this level, it would be a significant extension on top of our work.

Another place that our work could be extended is the amount of entropy used to calculate the key. Obtaining enough entropy in an embedded system can be a challenge due to limited sources of randomness in simpler processors. Working on finding sources of entropy that could be found among printers would be an interesting problem and could make the results of the encryption process stronger.

Finally, a Diffie-Helman[7] style key exchange could both help make the key stronger because entropy could be harvested from the requesting computer rather than only from the printer, and it could reduce the overhead for the person printing.

### 6.1 3D Printing

As stated in the Müller paper, 3D printing, as a rapidly evolving device with rising adoption, but relatively immature protocols is a potential avenue for attacks [11] of all sorts. While the G-Code format is not Turing-complete, which rules it out for adapting this protocol, raising awareness of this as an issue on some level, could be the start of encouraging greater security on networked 3D printers. Indeed, many common embedded systems of this sort are woefully under-prepared to be exposed to the web at large with little or no security, with naive interfaces designed to work in ideal conditions, but are easily exploited by adversaries with all manner of malicious intent.

### 6.2 Beyond Printing - IoT

The real contribution of this work is in showing that a protocol is possible. Printers aren't the only embedded system with blatant and possibly irresponsible security flaws. Any networked device that isn't security behind a firewall is in serious danger of being compromised. This is *far* more concerning when you consider very simple networked devices that are designed to be connected to the web *all the time*, in the so-called Internet of Things. IoT devices are simple to design, implement, and deploy on the surface, but the security implications of such devices are far-reaching and profound. The whole philosophy of the Internet of Things is embedded systems on low-power devices that *log data*. Data that we may or may not be comfortable sharing with the world at large. And many, if not most, IoT devices have little in the way of encryption. Devices such as security cameras, activity trackers, and even portable video game systems might be sending information across the web with little or no encryption. Consider that there are popular social media sites that still don't use end-to-end encryption or only recently deployed it.

In order to facilitate simple, effective encryption for IoT devices, there are a couple of different avenues worthy of exploration. Our protocol might even work for any system that has a display. A key could be generated by a receiving device and then entered into the sending device and security could be enhanced. But if we consider that many embedded systems, by design, do not have displays, it might be more important or useful to work toward a more sophisticated key exchange protocol. As mentioned earlier, a Diffie-Hellman key exchange shouldn't be too terribly difficult to implement, and secure for most purposes. However, if some sort of asymmetric cryptographic primitive were to be employed (RSA, for example), perhaps some other device on the network (the router, perhaps?) could act as a trusted third party for exchanging certificates, keys, or other cryptographic elements. So long as we can be reasonably certain that this middle-man device is not compromised, then we can be reasonably confident that any communications with our devices will also be secure.

A potentially lucrative avenue for further research could focus on the creation of a local network certifying authority device which could facilitate secure communication between devices on a local network. Software libraries for embedded systems (i.e. Arduino, PIC, or other embedded microcontroller devices) could be developed to simplify the process of adding encryption to something as simple as an IoT hobby project, to something more far-reaching such as a home-automation system.

## 7 CONCLUSION

The Internet and the Web come with a built-in promise of untold capability to do things that no one in computing history has ever been able to do. It would be wonderful world if we could simply trust everyone to mind their own business and try not to step on anyone's toes. For a couple of decades, the internet worked this way, too. As it is, this is not the world we live in now and the security of networked devices is something we all need to consider. If we want the convenience of a networked device, but the security of a powerful firewall, we're most likely out of luck. However, perhaps with a little extra up-front work, even on an unsecured network, it might be possible to transmit information and generate a hard copy of some document with sensitive information securely and with only minor inconvenience.

It does require that the printer be already open to a certain set of vulnerabilities. It also requires a certian amount of upfront setup and there are a lot of moving parts. Also, the output is, to put it bluntly, not very pretty. Despite all of this, if there is a vital need to create a hard copy of some sensitive data on an unsecure network, it is, in theory, possible to do so, if only in an ad-hoc, makeshift way.

Our ad-hoc method of encrypting documents to be decrypted on a printers can potentially be expanded and adapted to address security concerns in other embedded systems, though automating its largely ad-hoc nature would be utterly essential to any implementation in a real-time embedded system.

Future work into self-decrypting files and local network key exchange protocols are an absolute must in the current security climate. Perhaps this work can convince the reader that future work in this area is critical to the future security of, not just networked printing, but embedded systems and the IoT in general. And perhaps in seeing a need, the reader is inspired to continue this research.

## REFERENCES


[1] [n. d.]. CUPS. Retrieved May 5, 2019 from https://www.cups.org

[2] [n. d.]. Hacking Printers. Retrieved May 5, 2019 from http://hacking-printers.net/

[3] Nadhem AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. 2013. On the Security of RC4 in TLS. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. USENIX, Washington, D.C., 305–320. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/alFardan

[4] C. J. Bashe, W. Buchholz, G. V. Hawkins, J. J. Ingram, and N. Rochester. 1981. The Architecture of IBM's Early Computers. *IBM Journal of Research and Development* 25, 5 (Sep. 1981), 363–376. https://doi.org/10.1147/rd.255.0363

[5] A. Crenshaw. 2005. Hacking network printers. http://www.irongeek.com/i.php?page=security/networkprinterhacking

[6] T. Dierks and E. Rescorla. 2006. *The Transport Layer Security (TLS) Protocol Version 1.1.* RFC 4346. RFC Editor. 87 pages. https://www.rfc-editor.org/info/rfc4346

[7] Whitfield Diffie and Martin Hellman. 1976. New Directions in Cryptography. In *IEEE Transactions on Information Theory*, Vol. 22. 644–654. https://doi.org/10.1109/TIT.1976.1055638

[8] Peter Ferrie. 2007. W32.Simile. Retrieved May 05, 2019 from https://www.symantec.com/security-center/writeup/2002-030617-5423-99

[9] P.R. Harrington and B. Wijinen. 2002. *An Architecture for Describing Simple Network Management Protocol Management Frameworks.* Lucent Technologies. https://tools.ietf.org/html/rfc3411

[10] Jens Müller. 2017. PRET - PRinter Exploitation Toolkit. https://github.com/RUB-NDS/PRET

[11] Jens Müller, Vladislav Mladenov, Juraj Somorovsky, and Jörg Schwenk. 2017. Exploiting Network Printers. https://www.nds.ruhr-uni-bochum.de/media/ei/arbeiten/2017/01/13/exploiting-printers.pdf

[12] Jens Müller, Vladislav Mladenov, Juraj Somorovsky, and Jörg Schwenk. 2017. SOK: Exploiting Network Printers. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Jose, CA, USA. https://doi.org/10.1109/SP.2017.47

[13] A. Popov. 2015. *Prohibiting RC4 Cipher Suites.* RFC 7465. RFC Editor. 6 pages. https://www.rfc-editor.org/info/rfc7465

[14] Chuangang Ren, Kai Chen, and Peng Liu. 2014. Droidmarking: Resilient Software Watermarking for Impeding Android Application Repackaging. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 635–646. https://doi.org/10.1145/2642937.2642977

[15] E. Rescorla. 2000. *HTTP Over TLS.* RFC 2818. RFC Editor. 7 pages. https://www.rfc-editor.org/info/rfc2818

[16] Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. 2008. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008.* http://www.isoc.org/isoc/conferences/ndss/08/papers/19_impeding_malware_analysis.pdf

[17] W. Olin Sibert. 1996. Malicious Data and Computer Security. In *Proceedings of the 19th National Information Systems Security Conference.*

[18] Ed Taft, Steve Chernicoff, and Caroline Rose. 1999. *PostScript Language Reference: third edition.* Adobe Systems Incorporated. https://www.adobe.com/content/dam/acom/en/devnet/actionscript/articles/PLRM.pdf.

[19] Mathy Vanhoef and Frank Piessens. 2015. All Your Biases Belong to Us: Breaking RC4 in WPA-TKIP and TLS. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 97–112. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/vanhoef

[20] Aaron Weaver. 2007. Corss site printing. https://www.helpnetsecurity.com/dl/articles/CrossSitePrinting.pdf

[21] T. Ylonen and C. Lonvick. 2006. *The Secure Shell (SSH) Transport Layer Protocol.* RFC 4253. RFC Editor. 32 pages. https://www.rfc-editor.org/info/rfc4253