eVerify seeks to bridge the wide gap between prospective customers acquiring a loan, and both the trusted/un-trusted institutions with the capability of providing that loan. The problem eVerify addresses, is the customers need for the protection of their privacy and confidentiality, alongside the institutions demand of client authenticity and verification. When our client seeking short/long term loans are customers across multiple banking institutions, it can be difficult to prove to a potential loaner their combined net worth, partially equating to their ability of paying back the loan in good faith. eVerify solves this problem through the use of the secure paillier cryptosystem and OCR.

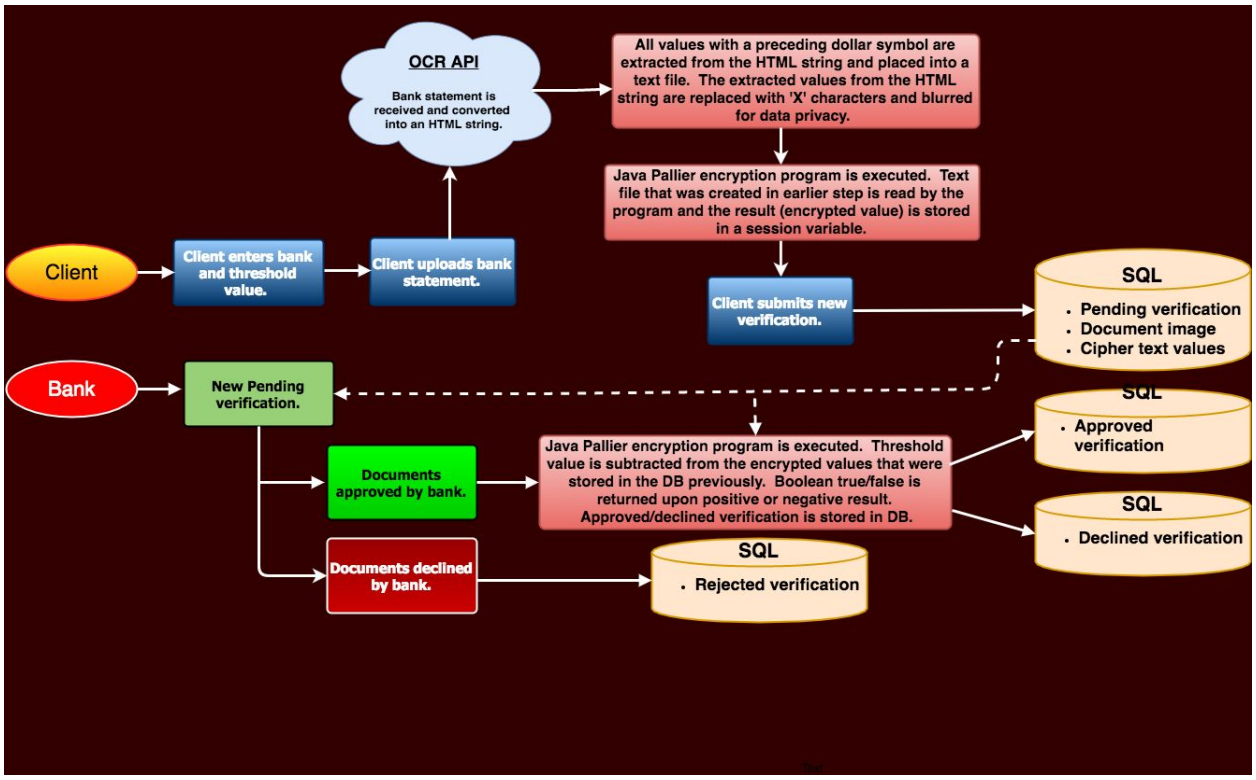
There exists two types of users of eVerify, loanees (clients) and loaners (credit/banking institutions). Clients, in today's day and age yearn for the protection of their privacy, including but not limited to banking and billing information across all of their bank entities and assets, and potentially their transactional histories. On the contrar, modern day loaners deal with a business of high risk, providing loans to clients of whom are expected to pay back their loan in good faith, a perfect example being student loans. Although these institutions bet on clients defaulting on their loans, accruing interest over time which generates a net profit for the loaning business, it is still a high risk high reward scenario. eVerify seeks to provide a compromise between these two entities in the loan process, to satisfy the needs of both of our users.

An example use case for eVerify as of the time of writing this document; a client is attempting to prove his accumulated financial assets exceed a given threshold of their choice, for which they deem a suitable partial level of trust between the loaning institution regarding their ability to pay back the requesting loan in good faith. However, this client's financial assets are placed across a multitude of banking organizations, let's say Citibank, Bank of America, Wells Fargo and Capital One. Our client trusts eVerify's ability to extract their account balances from each of these banks, and sum them to be provided to the loaning institution in question. The client logs onto our web server with valid user credentials, and uploads bank statements in the form of .pdf files straight from their online banking website, creating a new verification request. eVerify then reads through each document, utilizing an optical character recognition API call to extract account balances, then sends them to a paillier encryption java class file. The documents uploaded are then blurred in specific areas, to mask the actual account balances that were displayed, providing confidentiality for our clients. Paillier encrypts all sensitive account balances, across all uploaded bank statements provided, and queries the encrypted information alongside the blurred bank statements in our back end SQL database. This request then becomes a pending verification waiting on the approval of the loaning institution our client is requesting a loan from.

The loaning institution signs onto eVerify with separate user credentials, and can view the pending verification request, alongside the set of blurred bank statements, and the client generated threshold. At this moment, the loaning institution may reject the verification, for whatever means, or they may accept the verification, where it then becomes an approved verification. Once approved, the encrypted ciphertext sum Paillier generated during the new verification request gets multiplied by a random masking integer, and alongside the encrypted threshold, are both sent to the loaning organization through a third party means (Outside eVerify) alongside corresponding decryption parameters. The decryption keys are now in the hands of an untrusted entity, who may decrypt both the modified ciphertext sum, and threshold amount, and can perform the corresponding subtraction

operation. The purpose of the masking element is to hide the true balance difference from the threshold from the untrusted loaning institution, but still provide information regarding if it had exceeded the given threshold, or falls below it (negative value). At the end of this operation, the loaning institution now can verify that the client requesting the loan has at minimum, the threshold amount in financial assets across their banking institutions.

eVerify utilizes several algorithmic components to perform its main objective. We start with an optical character recognition API call on all client uploaded documents when creating a new verification. This API call transforms the uploaded .pdf bank statements into a long, HTML string. Once in a string form, a simple regex parser identifies relevant tokens, such as “On Deposit” “Checking” “Savings” and most importantly, “$”. Once the integer account balance information is identified, the parser removes all commas from the string, as well as any decimal value, so it rounds down the account balance information. This is due to the fact that the Paillier class utilizes BigInteger object types for encryption, instead of BigDecimal. Once both checking and savings account balance information has been extracted from all documents, they are placed in a plaintexts.txt file, to be read as input by the Paillier.java file. A shell script executes the Paillier.java file, which simply generates a set of encryption and decryption keys and encrypts all values taken as input in the plaintext.txt file, then sums them accordingly. The ciphertext sum value is then multiplied by a randomly generated masking value, so as to blind the true difference when decrypted from the given threshold amount. The output is printed to console, and then read into our backend database, while simultaneously the sensitive plaintext.txt file is deleted immediately. Concurrently, the account balance information in the HTML string that was converted from the .pdf bank statements are replaced with arbitrary XXX.XX characters, then applied a blurring css attribute. This both protects the originality of the document, and from anyone seeking to unblur the css attribute, to see the cleartext behind it. These blurred documents are stored alongside the modified ciphertext sum value in the database.



(The above diagram was created pre use of masking element for ciphertext sum, and assumes eVerify performs the final subtraction of D(ciphertextSum) - D(Threshold) = Result +/- . Since the creation of this diagram, we have decided to provide the decryption keys to the third party creditor/banking institution, as to prove there exists an untrustworthy entity of which the account information needed to be encrypted for. And despite supplying this entity with decryption keys, the masking element will hide the true balance difference from ciphertextSum from threshold, to still provide confidentiality for our loanee clients.)