An Exploration of the Effectiveness of Quantum Key Distribution for Secure Communication in Modern Environments

To what extent is the BB84 quantum key distribution protocol effective for secure communication in modern environments?

A Computer Science Extended Essay

Word count: 3,690

Contents

1. Introduction

The rapid development of modern technologies and the equally rapid adoption of these technologies into nearly all facets of human life have proven digital cryptography to be an essential tool for virtually all human functions. With the goal of protecting sensitive information from the viewing and modification of bad actors, traditional digital cryptography has been implemented in many forms, including the encrypted storing of passwords; secure web browsing via the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols; cryptocurrencies like Bitcoin and Ethereum; and electronic signatures that are often enforceable by law. However, traditional encryption techniques continue to face mounting challenges from the rapid advancement of the very technologies they're supposed to protect.

Enter the BB84 quantum key distribution (QKD) protocol. Unlike traditional encryption techniques, which rely on the complexity of mathematical algorithms, BB84 uses the principles of quantum mechanics to establish secure communication channels. BB84 relies on the transmission of individual photons, which can exist in multiple states simultaneously, to encode information. Through a process of quantum key generation and exchange, BB84 enables two parties to securely establish a shared encryption key – known only to them – without a risk of interception or eavesdropping. Their key can then be used to encrypt messages to be transmitted along a public channel.

In essence, BB84 represents an evolution in encryption technology as it offers a level of security that should be resilient against future technological advancements. As

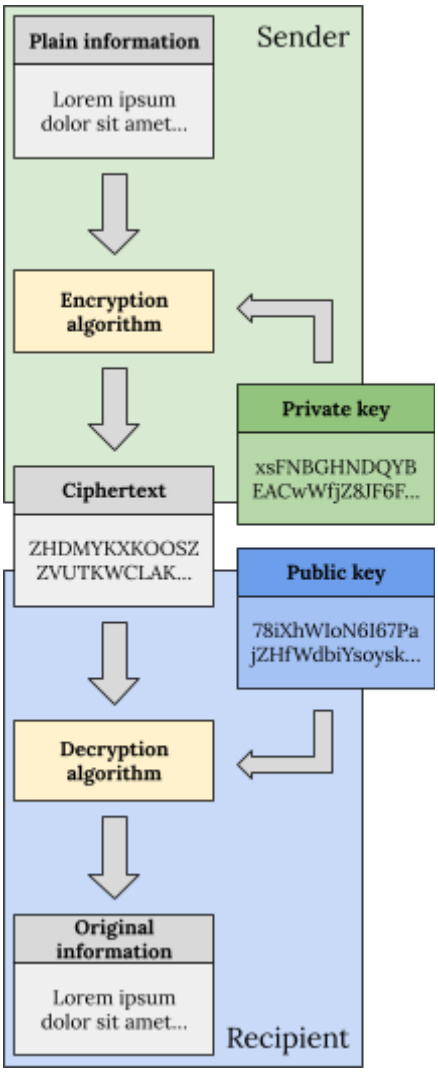
such, it holds promise for standardised use in modernised environments where data security is of the utmost importance.

The intent of this paper is to determine to what extent the BB84 quantum key distribution protocol is effective for standardised use in modern environments. The paper will investigate the resilience of quantum key distribution technology to ensure the resilience against potential future threats in computing technologies, and discuss the security implications of implementing the BB84 protocol in data-critical environments. Finally, a technical demonstration is conducted, using a program to emulate a BB84 system, demonstrating the system's effectiveness for secure communication. According to the research and technical demonstration, I find that the BB84 quantum key distribution protocol is indeed effective for standardised use in modern environments due to its exceedingly secure communication features that provide necessary protections in environments where the performance cost of implementing the system can be neglected.

1.1 Digital cryptography and cryptosystems

Digital cryptography is built around the four tenets of confidentiality, integrity, non-repudiation, and authentication, such that the sent information should only be accessed by the person for whom it is intended; the information should not be modified or altered during transmission; the sender of the information should not have the ability to deny their intention to send the information; and the sender's and recipient's identities must be provable (IBM). This section will explore the security of current modern cryptographic systems ("cryptosystems").

Figure 1. A typical asymmetrical cryptosystem



A cryptosystem encompasses all those concerned with the sending and receiving of encrypted data. There exist two categories of cryptosystems: symmetrical and asymmetrical; a cryptosystem is considered “symmetrical” when both the sender and the receiver use the same key to encrypt and decrypt information, and “asymmetrical” when the system involves the use of two separate keys: a private key, which must be kept private, and a public key, which should be shared publicly (Berlove and Laroche).

1.1.1 Common cryptosystems

The Caesar cipher is a common example of a symmetrical cryptosystem. The cipher works by transposing any phrase p by any n amount of letters, so, for instance, a Caesar cipher of $n = 3$ would, given the input string $p =$ "$computer$", produce an output string of "$frpsxwhu$". Since the receiving party would be aware of the value of n, they would be able to decipher the message by simply performing onto it a Caesar cipher of $-\ n$, producing the original p. With only twenty-six possible values of n in the standard English alphabet, this particular cipher can be easily broken by a process of repeated guessing and checking. Symmetrical cryptosystems are characterized by their speed and efficiency, though they lack the far more thorough security provided by asymmetrical cryptosystems because of the shared-key approach as the use of a single key for both

encryption and decryption puts all present and past communications at risk for compromise if this sole key is insecurely shared or stored.

On the other hand, public-key cryptography is an example of an asymmetrical cryptosystem (as depicted in Fig. 1) that may be implemented with modern technology. Two keys (long strings of letters and numbers) are generated via complicated one-way mathematical functions, of which one is made public and the other is kept private. Anyone in possession of the public key may use it to encrypt text, producing a "ciphertext", but the private key is required in addition to the public key to decrypt this ciphertext back to the original text. This system of cryptography introduces a layer of security that, in the event of an interceptor or unwanted party snooping on the message during transmission, will prevent said unwanted party from producing the original message. It will also prevent this party from decrypting previously transmitted messages, for instance, if encrypted chat logs were to be exposed.

RSA (Rivest–Shamir–Adleman) is an example of a public-key cryptosystem implementation that provides security to a significant number of communications on the modern web (Matthews). RSA has been implemented through RSA public-key encryption and the RSA digital signature scheme, and this cryptosystem relies on the assumption that factoring large numbers is computationally hard. While this assumption has not been disproven through the use of classical computers, it can be countered with the use of quantum computing (as will be discussed in more detail later in this essay).

1.2 Quantum key distribution and the BB84 protocol

Quantum key distribution (QKD) is a lesser-adopted technology that leverages the principles of quantum mechanics to provide enhanced security in communication systems. The BB84 protocol, proposed by Charles Bennett and Gilles Brassard in 1984, represents one of the first and perhaps one of the most well-known QKD protocols. BB84 utilizes quantum bits, or qubits, which can exist in superpositions of states, to enable fundamentally new methods of secure communication. It has the potential to alleviate the potential vulnerabilities of public-key cryptography by enabling far more secure communication between two parties.

A photon is the smallest unit of light and it cannot be divided further. It behaves like a particle but it also exhibits wave-like properties; for instance, a photon can have its electromagnetic field oriented or polarized in a specific direction, i.e. horizontally or vertically (Rogers 31). Through this, we can achieve binary encoding (i.e., “0” could be encoded as horizontal polarization and “1” as vertical polarization). Furthermore, photons can be created in a state of superposition where they simultaneously exhibit multiple polarization states. We can measure these states in different ways, allowing us to encode quantum bits (or qubits) into the polarization of a photon.

QKD is simply the usage of this technology to establish the public-key symmetric cryptosystem. Though symmetric cryptosystems tend to be demonstrably less secure than asymmetric ones, QKD is demonstrably far more secure than classic public-key cryptography as:

- the key itself is long enough to be secure; and
- the key is shared in an eavesdropper-blocking manner;

but mainly due to two rules introduced by the quantum technology itself:

- Wootters, Zurek, and Dieks' *No-Cloning Theorem*, which states that "classical information can in principle always be copied and quantum information cannot" (Farhi and Harrow). According to this theorem, while a beam of light can be split into two beams, a photon, as the smallest unit of light, cannot be divided further in a similar manner. As such, there is no way to copy the photon's state without destroying the original state; and
- Werner Heisenberg's *Uncertainty Principle*, which states that "the more precisely one [tries] to measure the position, the more uncertain the momentum [will] become, and vice versa" (APS125). According to this principle, it is impossible to know for certain everything about a quantum state because when we measure it we disturb it.

Since there is no way to copy the photon's state without destroying the original, and there is no way to measure the photon's state without disturbing it, we cannot learn anything about the system without disturbing it in some way. These rules form the basis of BB84's intensely secure communication.

2. Quantum resilience for future-proof security

BB84 provides an advantage over traditional encryption methods by offering resilience against potential future threats from the development of quantum computing technology.

One example of a threat to traditional encryption introduced by quantum computing is Shor's algorithm. Introduced by mathematician Peter Shor in 1994, Shor's algorithm is a quantum algorithm designed for integer factorization, or decomposing a composite number into its prime factors. This task is computationally intensive for classical computers, especially as the size of the number increases. However, Shor's algorithm leverages the principles of quantum mechanics to perform factorization exponentially faster than classical computers. While classical algorithms run in sub-exponential time, Shor's algorithm runs in polynomial time, making it significantly faster (Shor). Polynomial time is significantly faster than sub-exponential time for large inputs. For example, an algorithm that runs in $O(n^2)$ time (a polynomial time) will be much faster than one that runs in $O(2^n)$ time (an exponential time) as n (the input size) gets larger. Shor's algorithm allows quantum computers to factor large integers exponentially faster than classical algorithms, which is significant because many widely used encryption systems such as RSA depend on the assumption that factoring is computationally hard. It is fundamentally a threat to the encryption of today.

Traditional encryption algorithms like RSA are entirely based on the presumed difficulty of applying the integer factorization process to very large numbers, a task that is

computationally expensive and thus practically infeasible for classical computers (Lenstra). However, as quantum computing continues to develop, quantum algorithms, such as Shor's, threaten to undermine this security by leveraging the advantages in mathematical efficiency that quantum computers inherently possess over classical computers. Encryption techniques that depend on the complexity of mathematical problems, such as RSA, are becoming more vulnerable.

In fact, researchers from Google and KTH Royal Institute of Technology, Sweden demonstrated in 2019 that a quantum computer with 20 million qubits could theoretically break a 2,048-bit RSA encryption in just eight hours (Gidney and Ekerå), a significant reduction in resources and time that were previously thought to be required to do so. Furthermore, a researcher from Courant Institute of Mathematical Sciences, New York University demonstrated in 2024 a newfound quantum factoring algorithm that may be even faster than Shor's algorithm (Regev). These algorithms expose a newfound threat to the presumed security of the integer factorization process, as if the equations are to be reversed, the encryption algorithm is broken and messages can be decrypted by an unauthorized party with no way to detect this, and no way to prevent it besides purging the message for all parties.

By contrast however, BB84 is uniquely positioned to counter these threats by leveraging quantum mechanics rather than mathematical complexity. In the BB84 protocol, any attempt to intercept or even measure the quantum key would disturb the quantum states of the photons used to transmit the key, immediately alerting both parties to any potential eavesdropping in addition to altering the message itself for both the

intended recipient and the eavesdropper. This ensures that the key exchange is secure, any interference is detectable, and in the case of interference, no party can receive the message, which is the safest option. Traditional encryption methods on the other hand can be broken without detection, leaving the data vulnerable without the knowledge of the communicating parties.

BB84's inherent resilience against the threats of today and tomorrow makes it a strong candidate for future secure communication. As quantum computing evolves and threatens to break traditional encryption, BB84 would allow us to use those same technologies to safeguard messages from interceptors, not only alerting the recipient of any interception but also preventing the original message from being read by an unintended party in the event of an interception. BB84 offers a secure, forward-thinking solution that aligns with the increasing need for robust data protection in the face of advancing threats.

3. Secure communication in data-critical environments

In data-critical environments, the secure communication of information is incredibly important. The BB84 quantum key distribution protocol offers significant advantages over traditional cryptographic methods. One such advantage is its ability to detect eavesdropping attempts in real-time. This is made possible by quantum mechanics—when a third party attempts to intercept the key, the no-cloning theorem ensures that the quantum state of the transmitted photons is irreversibly altered, which creates anomalies in the data. These anomalies alert the intended recipient to the breach,

which would ideally lead to the destruction of the compromised key. By contrast, traditional cryptographic methods offer no methods to detect security breaches, which has led to data leaks and vulnerabilities. One such example is the Heartbleed bug, a vulnerability in the OpenSSL library which allowed attackers to exploit a flaw in the heartbeat extension and access sensitive data like encryption keys and passwords without detection. Since traditional encryption methods lacked real-time breach detection, the vulnerability went unnoticed for years and compromised millions of users, with researchers stating all internet users are “likely to be affected either directly or indirectly” (Synopsys, Inc). Such a breach would not have been possible had the BB84 protocol been in use.

Beyond the immediate detection of threats, implementing BB84 has the advantage of preemptive data protection, which reduces the need for retroactive measures. Traditional cryptographic systems can only reveal vulnerabilities after a breach has occurred, forcing companies to implement retroactive mitigation strategies such as re-encrypting data, or monitoring for suspicious activities. These are costly and time-consuming and would be entirely circumvented with the use of BB84’s real-time breach prevention.

Furthermore, BB84’s is forward secure. This provides another advantage over traditional encryption, with the BB84 protocol ensuring that even if one key exchange is compromised, future keys remain secure. This is because each key exchange is independent of previous ones (Rogers), so an attacker cannot use past intercepted keys to predict future keys or communications. This makes BB84 more valuable in long-term

communication systems, where the continuous exchange of sensitive information would require ongoing protection, without the risk of cumulative vulnerabilities.

There are many practical implications for the instant detection of interception, especially in environments where the integrity of data is vital, in sectors such as finance, healthcare, and government. Where sensitive information is constantly exchanged, the assurance that eavesdropping can be immediately detected significantly reduces the risk of data breaches which could have untold consequences.

4. Technical demonstration

In this section, a technical demonstration of the BB84 quantum key distribution cryptosystem is conducted to demonstrate the system's effectiveness for secure communication in modern environments. The emulator, written in the Lua programming language, was carefully designed to emulate a sender and receiver to calculate the security of an information exchange with and without eavesdropper(s) (i.e. how many bits are lost in transit and how similar the received information is to the initial sent information for both the intended recipient and an eavesdropper). To achieve this goal, the emulator system consists of four parts:

1. the **dictionary**, which generates randomized strings of English dictionary words, emulating the sending of strings that might be sent in a real-world information exchange. The core functionality of this component is visible in Figure 2.

a. This component makes use of a list of common US English words, provided by GitHub user David Norman (Norman).

2. the **encoder-decoder**, which encodes ASCII strings to binary and decodes binary back to ASCII strings. The core functionality of this component is visible in Figure 3.

Figure 2. Core functionality of the dictionary component

```
random = function(length)
    length = length or 10
    local ret = "" -- return value

    for i = 1, len do
        ret = ret .. words[math.random(1, #words)]
        if i ~= len then
            ret = ret .. " "
        end
    end

    return ret
end
```

Figure 3. Core functionality of the encoder-decoder component

```
-- encode ASCII string to binary
encode = function(ascii_str)
    local binary_str = ""

    for i = 1, #ascii_str do
        local byte = ascii_str:sub(i, i):byte()  -- get the ASCII value of the character
        local binary = ""

        for j = 7, 0, -1 do  -- loop from 7th to 0th bit
            local bit = math.floor(byte / 2^j) % 2  -- extract the bit
            binary = binary .. tostring(bit)  -- append the bit to the binary string
        end

        binary_str = binary_str .. binary  -- append the 8-bit binary string
    end

    return binary_str
end,

-- decode binary string to ASCII
decode = function(binary_str)
    local ascii_str = ""

    for i = 1, #binary_str, 8 do
        local byte = binary_str:sub(i, i + 7)  -- get 8-bit chunk
        local char = string.char(tonumber(byte, 2))  -- Convert binary to ASCII
        ascii_str = ascii_str .. char
    end

    return ascii_str
end
```

3. the **automator**, which continually runs the emulation with a set number of interceptors for a set number of repetitions. The complete code for this component is available in Appendix C.

4. the **emulator** itself, which simulates the sending and receiving of strings along quantum bases. The emulator does not run on its own and simply

exports a function to be run by the automator, returning important statistical information such as the number of successfully transmitted bits, which greatly reduces the time and effort necessary to collect data using the emulator. The complete code for this component is available in Appendix B.

4.1 Use of the automator to demonstrate changes introduced by interceptors

The automator was used to run 100 trials of the emulator with one, zero, and two interceptors each, for a total of 300 trials, and the raw data extracted from these trials is viewable in Appendix A, where an "unsuccessful" bit is defined as any bit where the sender basis and recipient basis do not match and the bit must thus be discarded and a "successful" bit is any bit that is successfully transmitted (i.e., any bit that is not unsuccessful). Through the analysis of the data, several trends are revealed:

1. For any trial with zero interceptors, the string is always transmitted successfully (i.e., the sent and received strings match). However, the inverse is also true; for any trial with at least one interceptor, the string will never be transmitted successfully since, as a result of some individual bits failing to send, the entire message will be corrupted and unreadable after being decoded to ASCII. This is clearly demonstrated in Figure 4 and Figure 5. Note how, in Figure 5, not only is the received message unreadable, it is of an entirely different length, signifying the gravity of the data corruption. This confirms that there is no way to copy or observe the photon's state without destroying the original state (the no-cloning theorem).

Figure 4. Sample results after complete run with zero interceptors

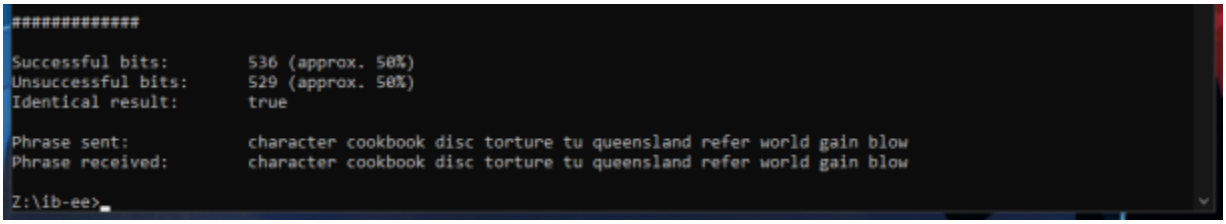
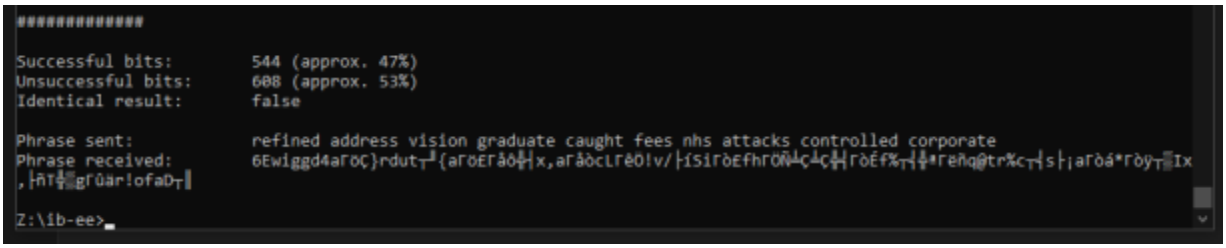


Figure 5. Sample results after complete run with one interceptor

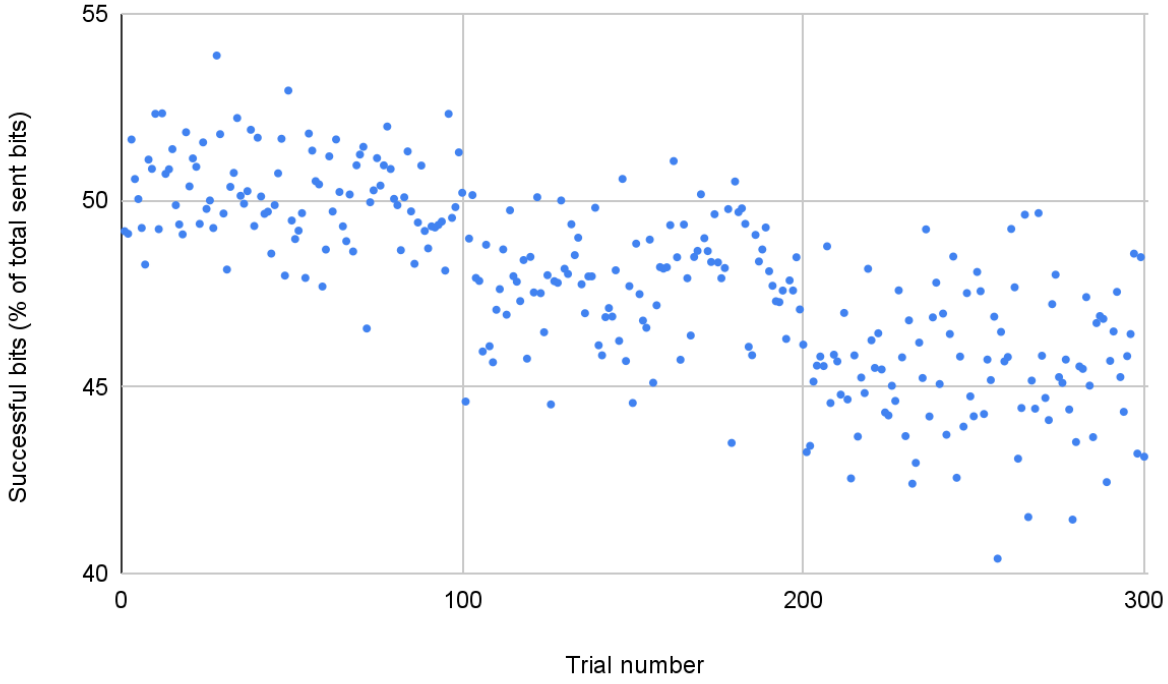


2. The percentage of successful bits decreases as more interceptors are added, meaning more bits are discarded with the introduction of more interceptors. This is also demonstrated in Figure 4 and Figure 5.
    a. The average percentage of successful bits for zero interceptor runs is 50.166%. For one interceptor runs it is 47.833%, with a 2.333% decrease. For two interceptor runs it is 45.517%, with a further 2.316% decrease.
    b. Figure 6 contains a scatter plot constructed using the raw data extracted from these trials. This visual interpretation of the data demonstrates that:

i. The percentage of total sent bits that are successful is directly affected by the number of interceptors used, with a flat, approximately 2.3% decrease per interceptor added.

ii. An increase in interceptors decreases the density of the data points; thus, the use of more interceptors introduces more variability in the number of bits that will be discarded.

Figure 6. Trial number compared to successful bits (% of total sent bits)



The percentage of successful bits could be useful as a measure to determine the validity of the decoded string. For instance, as the lowest value in the first 100 trials is 46.567%, this number could be used as an approximate guideline; if the system were to produce a value lower than this, this could serve as another indicator that there is an interceptor present. However, this is not a surefire method to determine the presence of

an interceptor and should only prompt a human investigator to take a closer look at the system. This also may not be the most useful metric as an interceptor would be unable to read the intended string either way due to the laws of quantum mechanics (as previously discussed).

4.2 Analysis and limitations of the method

Overall, the technical demonstration in its current form managed to aid me in building a definitive answer to the research question of this essay. However, after further thought, it is clear to me that there exist some limitations in this method that could potentially affect the validity and usefulness of the results. They are as follows:

1. The simplified nature of the emulator itself. The emulator accurately simulates the sending and receiving of information but does not account for real-world factors such as noise, imperfect quantum states, or other hardware limitations that might influence the key distribution process in an actual quantum system. In real-world settings, quantum key distribution is subject to physical constraints like photon loss and the probabilistic nature of quantum measurements (measurement disturbs the state of a system), none of which were modelled in this demonstration. As such, the results obtained from the emulator may be somewhat optimistic compared to what might be observed in a practical demonstration.
2. The small number of trials run. A small number of trials may not provide enough statistical data to fully assess the security of the system in

environments where the system is run more often. For instance, running the system more times may produce rarer outliers that should be noted as they may affect the reliability of the system. Large sample sizes would help draw more definitive conclusions about the system's robustness.

5. Conclusion

Through the research and technical demonstration, I find that the BB84 quantum key distribution protocol is indeed effective for standardised use in modern environments due to its exceedingly secure communication features that provide necessary protections in environments where the performance cost of implementing the system can be neglected. The BB84 system offers a secure, forward-thinking solution that matches the increasing need for robust data protection in the face of advancing threats. Furthermore, quantum key distribution's ability to immediately detect interceptors based on the received data and the percentage of successfully transmitted bits allows for further protection in all modern environments.

These capabilities make a BB84 system not only effective, but extremely valuable, for secure communication in modern environments where the benefits of real-time eavesdropping detection outweigh the cost. By leveraging the principles of quantum mechanics – specifically, the no-cloning theorem and the uncertainty principle, the BB84 introduces incredibly useful security features.

However, the implementation of BB84 is not without its challenges. The system requires quantum equipment where certain factors such as noise, imperfect quantum

states, or other hardware limitations might influence the key distribution process in an actual quantum system, introducing physical constraints and making the system susceptible to the probabilistic nature of quantum measurements. Advances in hardware and error-correction techniques will be necessary to overcome these obstacles and ensure successful adoption.

The BB84 protocol represents a groundbreaking step forward in the field of cryptography, offering a forward-thinking solution to evolving security threats — however, its implementation requires specialized equipment and must be correctly set up and used. As such, it is most suitable for organizations that can justify offsetting these expenses, such as governments or other industries handling highly sensitive data.

Works Cited

APS125. “This Month in Physics History: February 1927: Heisenberg's Uncertainty Principle.” *APS125*, 2024, https://www.aps.org/archives/publications/apsnews/200802/physicshistory.cfm. Accessed 6 August 2024.

Berlove, Orlee, and Gregg Laroche. “Public and private encryption keys.” *PreVeil*, 11 July 2024, https://www.preveil.com/blog/public-and-private-key/. Accessed 27 July 2024.

Farhi, Edward, and Aram Harrow. “Quantum Cloning, Quantum Money, and Quantum Monogamy.” *Physics at MIT*, MIT, 2013, https://physics.mit.edu/wp-content/uploads/2021/01/physicsatmit_13_farhiharrow.pdf. Accessed 6 August 2024.

Gidney, Craig, and Martin Ekerå. “[1905.09749] How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits.” *arXiv*, 23 May 2019, https://arxiv.org/abs/1905.09749. Accessed 6 September 2024.

IBM. “What Is Cryptography?” *IBM*, https://www.ibm.com/topics/cryptography. Accessed 15 November 2024.

Lenstra, Arjen K. “Integer Factoring.” *Springer Link*, Encyclopedia of Cryptography and Security, 2011, https://link.springer.com/referenceworkentry/10.1007/978-1-4419-5906-5_455. Accessed 10 September 2024.

Matthews, Tim. “How RSA and the Diffie-Hellman Key Exchange Became the Most Popular Cryptosystems.” *Exabeam*, 3 September 2019, https://www.exabeam.com/blog/infosec-trends/how-rsa-and-the-diffie-hellman-key-exchange-became-the-most-popular-cryptosystems/. Accessed 19 November 2024.

Norman, David. “1-1000.txt.” *1,000 most common US English words*, GitHub, 26 November 2012, https://gist.github.com/deekayen/4148741. Accessed 19 November 2024.

Regev, Oded. “[2308.06572] An Efficient Quantum Factoring Algorithm.” *arXiv*, 12 August 2023, https://arxiv.org/abs/2308.06572. Accessed 6 September 2024.

Rogers, Daniel J. *Broadband Quantum Cryptography*. Morgan & Claypool Publishers, 2010.

Shor, Peter W. “Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer.” *arXiv*, 30 August 1995, https://arxiv.org/abs/quant-ph/9508027. Accessed 14 October 2024.

Synopsys, Inc. *Heartbleed Bug*, Synopsys, Inc, 3 Jule 2020, https://heartbleed.com. Accessed 27 July 2024.

Appendix

Appendix A. Data collected using the automator

As stated in the body of the essay, the following chart displays the processed data for three hundred trials of the BB84 system emulator. It displays the number of successful bits (bits that did not need to be discarded) and unsuccessful bits as raw numbers and as percentages of the total number of sent bits. It also displays whether the sent and received strings match.

Trial number	Interceptors used	Total number of sent bits	Successful bits		Unsuccessful bits		String successfully transmitted? (do sent and received strings match?)
			number	percent of total (%)	number	percent of total (%)	
1	0	1334	656	49.17541229	678	50.82458771	yes
2	0	1401	688	49.10778016	713	50.89221984	yes
3	0	1193	616	51.63453479	577	48.36546521	yes
4	0	1218	616	50.57471264	602	49.42528736	yes
5	0	1327	664	50.03767898	663	49.96232102	yes
6	0	1153	568	49.26279271	585	50.73720729	yes
7	0	1077	520	48.28226555	557	51.71773445	yes
8	0	1049	536	51.09628217	513	48.90371783	yes
9	0	1180	600	50.84745763	580	49.15254237	yes
10	0	1162	608	52.32358003	554	47.67641997	yes

11	0	1105	544	49.23076923	561	50.76923077	yes
12	0	1177	616	52.3364486	561	47.6635514	yes
13	0	1262	640	50.71315372	622	49.28684628	yes
14	0	1196	608	50.8361204	588	49.1638796	yes
15	0	1199	616	51.37614679	583	48.62385321	yes
16	0	1187	592	49.873631	595	50.126369	yes
17	0	1167	576	49.35732648	591	50.64267352	yes
18	0	1320	648	49.09090909	672	50.90909091	yes
19	0	1312	680	51.82926829	632	48.17073171	yes
20	0	1191	600	50.37783375	591	49.62216625	yes
21	0	1064	544	51.12781955	520	48.87218045	yes
22	0	1163	592	50.90283749	571	49.09716251	yes
23	0	956	472	49.37238494	484	50.62761506	yes
24	0	931	480	51.55746509	451	48.44253491	yes
25	0	884	440	49.77375566	444	50.22624434	yes
26	0	1136	568	50	568	50	yes
27	0	1218	600	49.26108374	618	50.73891626	yes
28	0	965	520	53.88601036	445	46.11398964	yes
29	0	1097	568	51.77757521	529	48.22242479	yes
30	0	1144	568	49.65034965	576	50.34965035	yes

31	0	1296	624	48.14814815	672	51.85185185	yes
32	0	1096	552	50.3649635	544	49.6350365	yes
33	0	1419	720	50.73995772	699	49.26004228	yes
34	0	950	496	52.21052632	454	47.78947368	yes
35	0	1197	600	50.12531328	597	49.87468672	yes
36	0	1138	568	49.91212654	570	50.08787346	yes
37	0	1401	704	50.24982156	697	49.75017844	yes
38	0	1187	616	51.89553496	571	48.10446504	yes
39	0	1314	648	49.31506849	666	50.68493151	yes
40	0	1099	568	51.6833485	531	48.3166515	yes
41	0	1405	704	50.10676157	701	49.89323843	yes
42	0	1257	624	49.64200477	633	50.35799523	yes
43	0	998	496	49.6993988	502	50.3006012	yes
44	0	1334	648	48.57571214	686	51.42428786	yes
45	0	1203	600	49.87531172	603	50.12468828	yes
46	0	1167	592	50.72836332	575	49.27163668	yes
47	0	1239	640	51.65456013	599	48.34543987	yes
48	0	1117	536	47.98567592	581	52.01432408	yes
49	0	967	512	52.94725957	455	47.05274043	yes
50	0	1116	552	49.46236559	564	50.53763441	yes

51	0	1209	592	48.96608768	617	51.03391232	yes
52	0	1236	608	49.19093851	628	50.80906149	yes
53	0	1176	584	49.65986395	592	50.34013605	yes
54	0	1252	600	47.92332268	652	52.07667732	yes
55	0	1282	664	51.79407176	618	48.20592824	yes
56	0	1309	672	51.3368984	637	48.6631016	yes
57	0	966	488	50.51759834	478	49.48240166	yes
58	0	1158	584	50.43177893	574	49.56822107	yes
59	0	1040	496	47.69230769	544	52.30769231	yes
60	0	1216	592	48.68421053	624	51.31578947	yes
61	0	1141	584	51.18317266	557	48.81682734	yes
62	0	1014	504	49.70414201	510	50.29585799	yes
63	0	1100	568	51.63636364	532	48.36363636	yes
64	0	1099	552	50.22747953	547	49.77252047	yes
65	0	1298	640	49.30662558	658	50.69337442	yes
66	0	1145	560	48.90829694	585	51.09170306	yes
67	0	1244	624	50.1607717	620	49.8392283	yes
68	0	987	480	48.63221884	507	51.36778116	yes
69	0	1272	648	50.94339623	624	49.05660377	yes
70	0	1218	624	51.23152709	594	48.76847291	yes

71	0	1182	608	51.43824027	574	48.56175973	yes
72	0	1340	624	46.56716418	716	53.43283582	yes
73	0	1041	520	49.95196926	521	50.04803074	yes
74	0	1289	648	50.27152832	641	49.72847168	yes
75	0	1189	608	51.13540791	581	48.86459209	yes
76	0	1127	568	50.39929015	559	49.60070985	yes
77	0	1115	568	50.94170404	547	49.05829596	yes
78	0	908	472	51.98237885	436	48.01762115	yes
79	0	1007	512	50.84409136	495	49.15590864	yes
80	0	1167	584	50.0428449	583	49.9571551	yes
81	0	1187	592	49.873631	595	50.126369	yes
82	0	1200	584	48.66666667	616	51.33333333	yes
83	0	1166	584	50.08576329	582	49.91423671	yes
84	0	1216	624	51.31578947	592	48.68421053	yes
85	0	1368	680	49.70760234	688	50.29239766	yes
86	0	1060	512	48.30188679	548	51.69811321	yes
87	0	1101	544	49.40962761	557	50.59037239	yes
88	0	1335	680	50.93632959	655	49.06367041	yes
89	0	1285	632	49.18287938	653	50.81712062	yes
90	0	1051	512	48.71550904	539	51.28449096	yes

91	0	1152	568	49.30555556	584	50.69444444	yes
92	0	1104	544	49.27536232	560	50.72463768	yes
93	0	1297	640	49.34464148	657	50.65535852	yes
94	0	1149	568	49.43429069	581	50.56570931	yes
95	0	1330	640	48.12030075	690	51.87969925	yes
96	0	1055	552	52.32227488	503	47.67772512	yes
97	0	1179	584	49.53350297	595	50.46649703	yes
98	0	1124	560	49.82206406	564	50.17793594	yes
99	0	1279	656	51.29007037	623	48.70992963	yes
100	0	1211	608	50.20644096	603	49.79355904	yes
101	1	1150	513	44.60869565	637	55.39130435	no
102	1	1370	671	48.97810219	699	51.02189781	no
103	1	1041	522	50.14409222	519	49.85590778	no
104	1	1540	738	47.92207792	802	52.07792208	no
105	1	1298	621	47.84283513	677	52.15716487	no
106	1	1136	522	45.95070423	614	54.04929577	no
107	1	1221	596	48.81244881	625	51.18755119	no
108	1	1241	572	46.0918614	669	53.9081386	no
109	1	1233	563	45.66098946	670	54.33901054	no
110	1	1143	538	47.06911636	605	52.93088364	no

111	1	1304	621	47.62269939	683	52.37730061	no
112	1	1105	538	48.68778281	567	51.31221719	no
113	1	1306	613	46.93721286	693	53.06278714	no
114	1	949	472	49.73656481	477	50.26343519	no
115	1	1157	555	47.96888505	602	52.03111495	no
116	1	1125	538	47.82222222	587	52.17777778	no
117	1	945	447	47.3015873	498	52.6984127	no
118	1	1283	621	48.40218239	662	51.59781761	no
119	1	1285	588	45.75875486	697	54.24124514	no
120	1	1058	513	48.48771267	545	51.51228733	no
121	1	1115	530	47.53363229	585	52.46636771	no
122	1	1142	572	50.08756567	570	49.91243433	no
123	1	1046	497	47.51434034	549	52.48565966	no
124	1	1033	480	46.46660213	553	53.53339787	no
125	1	1173	563	47.99658994	610	52.00341006	no
126	1	1152	513	44.53125	639	55.46875	no
127	1	1386	663	47.83549784	723	52.16450216	no
128	1	1404	671	47.79202279	733	52.20797721	no
129	1	1376	688	50	688	50	no
130	1	1256	605	48.16878981	651	51.83121019	no
131	1	1345	646	48.02973978	699	51.97026022	no

132	1	1175	580	49.36170213	595	50.63829787	no
133	1	1160	563	48.53448276	597	51.46551724	no
134	1	1302	638	49.0015361	664	50.9984639	no
135	1	1179	563	47.75233249	616	52.24766751	no
136	1	1075	505	46.97674419	570	53.02325581	no
137	1	1105	530	47.9638009	575	52.0361991	no
138	1	1278	613	47.96557121	665	52.03442879	no
139	1	1281	638	49.80483997	643	50.19516003	no
140	1	1455	671	46.11683849	784	53.88316151	no
141	1	1156	530	45.84775087	626	54.15224913	no
142	1	1007	472	46.87189672	535	53.12810328	no
143	1	1248	588	47.11538462	660	52.88461538	no
144	1	1060	497	46.88679245	563	53.11320755	no
145	1	1309	630	48.12834225	679	51.87165775	no
146	1	1129	522	46.23560673	607	53.76439327	no
147	1	1212	613	50.57755776	599	49.42244224	no
148	1	1324	605	45.69486405	719	54.30513595	no
149	1	1023	488	47.7028348	535	52.2971652	no
150	1	1059	472	44.57034939	587	55.42965061	no
151	1	1425	696	48.84210526	729	51.15789474	no

152	1	1274	605	47.48822606	669	52.51177394	no
153	1	1009	472	46.7789891	537	53.2210109	no
154	1	1333	621	46.58664666	712	53.41335334	no
155	1	1048	513	48.95038168	535	51.04961832	no
156	1	1064	480	45.11278195	584	54.88721805	no
157	1	1352	638	47.18934911	714	52.81065089	no
158	1	1427	688	48.21303434	739	51.78696566	no
159	1	1204	580	48.17275748	624	51.82724252	no
160	1	1116	538	48.2078853	578	51.7921147	no
161	1	1058	522	49.33837429	536	50.66162571	no
162	1	1136	580	51.05633803	556	48.94366197	no
163	1	1281	621	48.47775176	660	51.52224824	no
164	1	1159	530	45.72907679	629	54.27092321	no
165	1	1394	688	49.3543759	706	50.6456241	no
166	1	1367	655	47.91514265	712	52.08485735	no
167	1	1160	538	46.37931034	622	53.62068966	no
168	1	1419	688	48.48484848	731	51.51515152	no
169	1	1295	630	48.64864865	665	51.35135135	no
170	1	1206	605	50.16583748	601	49.83416252	no
171	1	1235	605	48.98785425	630	51.01214575	no

172	1	1363	663	48.64269993	700	51.35730007	no
173	1	1061	513	48.35061263	548	51.64938737	no
174	1	1084	538	49.63099631	546	50.36900369	no
175	1	1355	655	48.33948339	700	51.66051661	no
176	1	1367	655	47.91514265	712	52.08485735	no
177	1	1187	572	48.18871104	615	51.81128896	no
178	1	1081	538	49.76873265	543	50.23126735	no
179	1	1046	455	43.49904398	591	56.50095602	no
180	1	984	497	50.50813008	487	49.49186992	no
181	1	1117	555	49.6866607	562	50.3133393	no
182	1	1181	588	49.78831499	593	50.21168501	no
183	1	1276	630	49.37304075	646	50.62695925	no
184	1	1096	505	46.07664234	591	53.92335766	no
185	1	1193	547	45.85079631	646	54.14920369	no
186	1	1300	638	49.07692308	662	50.92307692	no
187	1	1131	547	48.3642794	584	51.6357206	no
188	1	1294	630	48.6862442	664	51.3137558	no
189	1	1380	680	49.27536232	700	50.72463768	no
190	1	1291	621	48.10224632	670	51.89775368	no
191	1	1268	605	47.71293375	663	52.28706625	no

192	1	1332	630	47.2972973	702	52.7027027	no
193	1	1174	555	47.27427598	619	52.72572402	no
194	1	1097	522	47.58432088	575	52.41567912	no
195	1	1145	530	46.28820961	615	53.71179039	no
196	1	1003	480	47.85643071	523	52.14356929	no
197	1	1305	621	47.5862069	684	52.4137931	no
198	1	1248	605	48.4775641	643	51.5224359	no
199	1	1215	572	47.0781893	643	52.9218107	no
200	1	1203	555	46.13466334	648	53.86533666	no
201	2	1334	577	43.25337331	757	56.74662669	no
202	2	1117	485	43.41987466	632	56.58012534	no
203	2	1318	595	45.14415781	723	54.85584219	no
204	2	1084	494	45.57195572	590	54.42804428	no
205	2	1159	531	45.81535807	628	54.18464193	no
206	2	1205	549	45.56016598	656	54.43983402	no
207	2	1220	595	48.7704918	625	51.2295082	no
208	2	985	439	44.56852792	546	55.43147208	no
209	2	1197	549	45.86466165	648	54.13533835	no
210	2	1182	540	45.68527919	642	54.31472081	no
211	2	1143	512	44.7944007	631	55.2055993	no

212	2	1111	522	46.98469847	589	53.01530153	no
213	2	1106	494	44.66546112	612	55.33453888	no
214	2	1248	531	42.54807692	717	57.45192308	no
215	2	999	458	45.84584585	541	54.15415415	no
216	2	1090	476	43.66972477	614	56.33027523	no
217	2	1032	467	45.25193798	565	54.74806202	no
218	2	1307	586	44.83550115	721	55.16449885	no
219	2	1501	723	48.16788807	778	51.83211193	no
220	2	1267	586	46.25098658	681	53.74901342	no
221	2	1147	522	45.51002616	625	54.48997384	no
222	2	1025	476	46.43902439	549	53.56097561	no
223	2	1247	567	45.4691259	680	54.5308741	no
224	2	950	421	44.31578947	529	55.68421053	no
225	2	911	403	44.23710209	508	55.76289791	no
226	2	1097	494	45.0319052	603	54.9680948	no
227	2	1107	494	44.62511292	613	55.37488708	no
228	2	1597	760	47.58922981	837	52.41077019	no
229	2	1260	577	45.79365079	683	54.20634921	no
230	2	1195	522	43.68200837	673	56.31799163	no
231	2	1291	604	46.78543765	687	53.21456235	no

232	2	1014	430	42.40631164	584	57.59368836	no
233	2	1066	458	42.96435272	608	57.03564728	no
234	2	1089	503	46.18916437	586	53.81083563	no
235	2	1397	632	45.23979957	765	54.76020043	no
236	2	1227	604	49.22575387	623	50.77424613	no
237	2	1036	458	44.20849421	578	55.79150579	no
238	2	1308	613	46.86544343	695	53.13455657	no
239	2	1475	705	47.79661017	770	52.20338983	no
240	2	1280	577	45.078125	703	54.921875	no
241	2	1286	604	46.96734059	682	53.03265941	no
242	2	1194	522	43.71859296	672	56.28140704	no
243	2	1480	687	46.41891892	793	53.58108108	no
244	2	1264	613	48.49683544	651	51.50316456	no
245	2	1076	458	42.56505576	618	57.43494424	no
246	2	1338	613	45.81464873	725	54.18535127	no
247	2	1270	558	43.93700787	712	56.06299213	no
248	2	1349	641	47.51667902	708	52.48332098	no
249	2	1247	558	44.74739374	689	55.25260626	no
250	2	1201	531	44.2131557	670	55.7868443	no
251	2	1200	577	48.08333333	623	51.91666667	no

252	2	1192	567	47.56711409	625	52.43288591	no
253	2	1240	549	44.27419355	691	55.72580645	no
254	2	1301	595	45.73405073	706	54.26594927	no
255	2	1277	577	45.18402506	700	54.81597494	no
256	2	1348	632	46.884273	716	53.115727	no
257	2	1109	448	40.39675383	661	59.60324617	no
258	2	1220	567	46.47540984	653	53.52459016	no
259	2	1182	540	45.68527919	642	54.31472081	no
260	2	1358	622	45.80265096	736	54.19734904	no
261	2	1637	806	49.23640806	831	50.76359194	no
262	2	1267	604	47.67166535	663	52.32833465	no
263	2	1105	476	43.07692308	629	56.92307692	no
264	2	1195	531	44.43514644	664	55.56485356	no
265	2	1181	586	49.61896698	595	50.38103302	no
266	2	1125	467	41.51111111	658	58.48888889	no
267	2	1439	650	45.17025712	789	54.82974288	no
268	2	1236	549	44.41747573	687	55.58252427	no
269	2	1327	659	49.66088922	668	50.33911078	no
270	2	1237	567	45.8367017	670	54.1632983	no
271	2	982	439	44.70468432	543	55.29531568	no

272	2	934	412	44.11134904	522	55.88865096	no
273	2	1241	586	47.21998388	655	52.78001612	no
274	2	1181	567	48.01016088	614	51.98983912	no
275	2	1193	540	45.26404023	653	54.73595977	no
276	2	1237	558	45.109135	679	54.890865	no
277	2	1301	595	45.73405073	706	54.26594927	no
278	2	1133	503	44.39541041	630	55.60458959	no
279	2	1192	494	41.44295302	698	58.55704698	no
280	2	1073	467	43.52283318	606	56.47716682	no
281	2	1045	476	45.55023923	569	54.44976077	no
282	2	1207	549	45.48467274	658	54.51532726	no
283	2	1177	558	47.4086661	619	52.5913339	no
284	2	1259	567	45.03574265	692	54.96425735	no
285	2	1111	485	43.65436544	626	56.34563456	no
286	2	1156	540	46.71280277	616	53.28719723	no
287	2	1307	613	46.90130069	694	53.09869931	no
288	2	1134	531	46.82539683	603	53.17460317	no
289	2	1013	430	42.44817374	583	57.55182626	no
290	2	1302	595	45.69892473	707	54.30107527	no
291	2	1338	622	46.48729447	716	53.51270553	no

292	2	1020	485	47.54901961	535	52.45098039	no
293	2	1436	650	45.26462396	786	54.73537604	no
294	2	1094	485	44.33272395	609	55.66727605	no
295	2	1198	549	45.8263773	649	54.1736227	no
296	2	1340	622	46.41791045	718	53.58208955	no
297	2	1017	494	48.57423795	523	51.42576205	no
298	2	1164	503	43.21305842	661	56.78694158	no
299	2	1283	622	48.48012471	661	51.51987529	no
300	2	997	430	43.12938816	567	56.87061184	no

Appendix B. Full emulator code

As stated in the body of the essay, the following code (emulator.lua) was used to run three hundred trials of a simulation of the BB84 system emulator; this specific software is the emulator itself. It returns a function to be called multiple times by the automator, returning statistical data each time, which is collected and processed by the automator.

```
-- emulator.lua
-- BB84 system emulator
require("moduleroot")


-- import modules
local encoder_decoder = require("encoder-decoder")
local dictionary = require("dictionary")

-- get a random basis (either "horizontal-vertical" or "angled")
function getRandomBasis()
   return randomFromArray({"horizontal-vertical", "angled"})
end

-- given a bit and basis, return its photon state
function bitToPhotonState(bit, basis)
```

```
    if basis == "horizontal-vertical" then
        return ({"H", "V"})[bit+1]
    elseif basis == "angled" then
        return ({"+45", "-45"})[bit+1]
    end
    return nil
end

-- given a photon state, return its bit
function photonStateToBit(state)
    return ({ ["H"] = 0, ["V"] = 1, ["+45"] = 0, ["-45"] = 1 })[state] or nil
end

-- print a table of bits
function printBitTable(t)
    local s = ""
    for i, bit in pairs(t) do
        s = (s .. bit)
        if (i % 8 == 0) then
            s = (s .. " ")
        end
    end
    print(s)
    return s
end

-- turn a fraction (a/b) to a rounded percent string
function roundPercent(a, b)
    return "approx. " .. math.floor(((a / b) * 100) + 0.5) .. "%"
end

-- return a random item from this array
function randomFromArray(array)
    return array[ math.random(1, #array) ]
end



-- initialize the random number generator with a differing seed
math.randomseed(os.time() * 1000 + rand(0, 9999))

-- return function
return { run = function (numberInterceptors)

    local SenderSentBit             -- the value that the pervious person sent

    local SenderBasis               -- the sender's basis
    local RecipientBasis            -- the recipient's basis
    local PreviousBasis             -- previous person's basis

    local CurrentReadBit            -- the value that the current person reads
     local PreviousSentBit            -- the value that the current person sends to the
next
```

```
    local SentBits = ""                -- every bit that is sent, even if it is later
discarded
   local DiscardedBitCount = 0     -- the number of discarded bits
   local SenderResult = ""         -- the result formed by the sender
   local ReceiverResult = ""       -- the result formed by the receiver
    local InterceptorResult = {}     -- result formed by all interceptors;   i = #, v =
result

   -- set a fallback value for the interceptor result
   setmetatable(InterceptorResult, {
       __index = function(t, k)
            warn("Error: This index (" .. tostring(k) or "non-string index" .. ") of
the InterceptorResult table does not exist!!!")
           return ""
       end
   })

   function run_Character(SenderSentBit)
       -- sender
       SenderBasis = getRandomBasis()
       SentBits = (SentBits .. SenderSentBit)

            print(string.format("Sending  a  %s  (%s)  on  basis  %s",  SenderSentBit,
bitToPhotonState(SenderSentBit, SenderBasis), SenderBasis))
       PreviousSentBit = SenderSentBit
       PreviousBasis = SenderBasis

       for j = 1, numberInterceptors do
           RecipientBasis = getRandomBasis()
           print(string.format("\nStart Interceptor #%s (%s)", j, RecipientBasis))

           -- recieves a 0 or 1 without knowing if it's right or random
           if (RecipientBasis == PreviousBasis) then
               CurrentReadBit = PreviousSentBit
           else
               CurrentReadBit = math.random(0, 1)
           end
                   print(string.format("Interceptor  read  %s  (%s)",  CurrentReadBit,
bitToPhotonState(CurrentReadBit, RecipientBasis)))
           InterceptorResult[j] = (InterceptorResult[j] .. CurrentReadBit)

           -- sends a 0 or 1 pretending they're Alice

           print("End Interceptor #" .. j)
           PreviousSentBit = CurrentReadBit
       end

       CurrentReadBit = PreviousSentBit
       RecipientBasis = getRandomBasis()

       print("\n\nRecipient basis: " .. RecipientBasis)
       if (RecipientBasis == SenderBasis) then
                   print("Recipient  reads:  "  ..  (CurrentReadBit  or  "")  ..  "  ("  ..
(bitToPhotonState(CurrentReadBit, RecipientBasis) or "") .. ")")
```

```
            SenderResult = (SenderResult .. SenderSentBit)
            ReceiverResult = (ReceiverResult .. CurrentReadBit)
        else
              print("Sender basis and recipient basis do not match; discarding current
bit")
            run_Character(SenderSentBit)
            DiscardedBitCount = (DiscardedBitCount + 1)
        end
    end


    local toSend = dictionary.random(10)
    local toSendBits = encoder_decoder.encode(toSend)
    for i = 1, toSendBits:len() do
        print("\n##############")
        print("Bit #" .. i .. "/" .. toSendBits:len())
        run_Character(toSendBits:sub(i, i))
    end

    print("\n##############")
    --print("Initial str:\t", (SentBits or ""))
    --print("Sender has:\t", (SenderResult or ""))
    --print("Receiver has:\t", (ReceiverResult or ""))
    --for i, v in pairs(InterceptorResult) do
        --print("Interceptor " .. i .. " has:", v)
    --end

    local succBits = string.len(SenderResult)
        print("\nSuccessful  bits:",  succBits  ..  "  ("  ..  roundPercent(succBits,
string.len(SentBits)) .. ")")
    local unsuccBits = DiscardedBitCount
       print("Unsuccessful  bits:",  unsuccBits  ..  "  ("  ..  roundPercent(unsuccBits,
string.len(SentBits)) .. ")")
    print("Identical result:", (SenderResult == ReceiverResult))

    print("\nPhrase sent:\t", toSend)
    print("Phrase received:\t", (encoder_decoder.decode(ReceiverResult) or ""))

    -- return logging info
    return string.len(SentBits), succBits, unsuccBits

end }
```

Appendix C. Full automator code

As stated in the body of the essay, the following code (automator.lua) was used to run three hundred trials of a simulation of the BB84 system emulator; this specific software is the automator. It calls the emulator function multiple times, and collects and processes the statistical data returned by that function.

```lua
-- automator.lua
-- Automatically runs emulator with set conditions
require("moduleroot")


-- import modules
local emulator = require("emulator")


-- configuration
local Repetitions = 1
local NumInterceptors = 0


-- statistics
local Stats = {}
Stats["Repetitions"] = Repetitions
Stats["Number of interceptors"] = NumInterceptors
Stats["Total bits"] = 0
Stats["Total successful bits"] = 0
Stats["Total unsuccessful bits"] = 0
Stats["All data"] = {}


-- run the automations
for i = 1, Repetitions do
   print("Running repetition " .. i .. "...")

   -- run the rep
   local a, b, c = emulator.run(NumInterceptors)

   -- count stats!
   Stats["Total bits"]                 = Stats["Total bits"]               + a
   Stats["Total successful bits"]      = Stats["Total successful bits"]    + b
   Stats["Total unsuccessful bits"]    = Stats["Total unsuccessful bits"]  + c
   table.insert(Stats["All data"], { a, b, c })

   print("Complete repetition " .. i)
end


-- big divider
print([[



   ###
   AUTOMATION COMPLETE
   ###
]])


-- print all stats
print([[
```

```
   # Stats:

]])
for i, v in pairs(Stats) do
   print(i, v)
end


-- then, print the numerical data for each rep!
print([[

   # Numerical data:

]])
print("Repetition number / Total bits / Total successful bits / Total unsuccessful
bits")
for i, v in pairs(Stats["All data"]) do
   print(i, v[1], v[2], v[3])
end
```