Journal of Management & Entrepreneurship
ISSN 2229-5348
UGC Care Group I Journal
Vol-13 Issue-01 April 2024

Is Two Factor Authentication Enough to Secure your System: A Study

[1] Monica Lamba, [2]Rajesh Kumar Jaiswal, [3] Udit Kashyap, [4] Sudhir Kumar Lakhan

[1,]Associate Professor, Department of ECE, Arya College of Engineering, Jaipur, Rajasthan, India

[2]Associate Professor, Department of MBA, Arya College of Engineering, Jaipur, Rajasthan, India

[3,4]B.Tech Student, Department of CSE, Arya College of Engineering, Jaipur, Rajasthan, India

lambamonica346@gmail.com, raj84jaiswal@rediffmail.com, uditkashyap007@gmail.com, skumar63002@gmail.com

Abstract: In this study, we explore the significance of constantly enhancing authentication methods to combatthe increasing cyber risks. As cybercrimes continue to soar, safeguarding online accounts becomes essential. We specifically analyze the effectiveness of two-factor authentication (2FA) in bolstering account security. By conducting a thorough review of relevant literature and examining real-life examples, we assess the strengths and limitations of 2FA. Our research also delves into the changing cyber environment and factors that impact the adoption of 2FA. Our findings shed light on how 2FA can effectively mitigate common threats such as phishing and credential breaches. Additionally, we explore new authentication technologies that are on the horizon.

1. Introduction

In today's digital age, protecting our personal and sensitive information is more important than ever. With traditional methods like passwords becoming more vulnerable to cyber-attacks, it's crucial to implement stronger security measures. Two-factor authentication (2FA) has become a popular solution to enhance account security by adding an extra layer of verification before granting access. In this overview, we will discuss 2FA (two-factor authentication), including its principles, mechanisms, benefits, and challenges. We will also explore how 2FA helps address modern cybersecurity issues and prevents unauthorized access to online accounts. Nowadays, more applications are incorporating various forms of two-factor authentication (2FA) or multi-factor authentication (MFA) into their testing processes. This paper offers a comprehensive look at the common methods used in 2FA and provides detailed instructions on how to assess them during testing.

2. How does Two-Factor Authentication Function Step by Step?

Two-factor authentication is the idea of using two different types of "factors" for security. Typically, these factors are categorized as "something you know," "something you have," "something you are," or "somewhere you are."For example, a password is “something you know”: just having two passwords doesn’t add significant security in general, but requiring you to be in your office or have your phone with you can add security: someone who has phished your password is less likely to be in your office or have also stolen your phone. Most 2FA implementations expect the second factor to be “something you have”, and that’s what we’ll focus on here. These fall into a few common implementations:

2.1 A device that generates a code

These were among the earliest implementations of two-factor authentication (such as via S/KEY or a physical token with a numeric display). In these, some offline mechanism generates a code, usually a six-digit number, and you type it in to your computer to log in. The idea is that that code could only be generated by the physical device you’re using (and the server) because of a shared secret, so if they match, you know that the person logging in has access to

the physical device. This is also among the most common mechanisms of two-factor support these days, due to the ubiquity of TOTP implementations (available in software such as Google Authenticator, as well as many associated applications such as Authy, Microsoft Authenticator, FreeOTP, and so on). In general, if you've had to point your phone at a QR code to set up 2FA, you're probably using TOTP. TOTP uses a shared secret that both your device and the server know, and combine with the current time to generate a number that both sides can agree on, but would be difficult for someone without the secret to guess. This is effectively the same principle behind RSA SecurID tokens. HOTP, a slight variant, simply uses a counter for the number of codes you've generated instead of the time, and S/KEY is also usage-based, but uses a different mechanism to generate the codes. In all of these cases, most servers expect that your clock and theirs – or your count of how many codes you've used and theirs – aren't exactly in sync, so they'll accept a small number of codes "around" the right one to provide some error recovery.

2.2 A device that does a cryptographic handshake

These are a bit more modern, because they rely on much more integration with your browser or operating system. Typically, these are using WebAuthn as part of FIDO2, and are along the lines of a YubiKey or other USB device you tap to authenticate, although WebAuthn is now supported natively on iOS and macOS devices. These "security keys" use a private key that is intended to never leave the device to sign a challenge from the server, proving that the device is present. As an added benefit, WebAuthn challenges specify the domain name you're logging into, so, for example, example1.com and example2.com can't impersonate one another. This helps alleviate phishing concerns, because even if an attacker can convince you to type in your password and use your security key, they still can't use the security key's response to log in to another website.

As a result of its phishing resistance and general ease of use, WebAuthn is generally the best 2FA implementation available for most applications. (Push-based confirmations on another device can provide additional features, such as authenticating specific transactions, but generally require a specific app be installed on a mobile device, which is a high bar for many users.)

2.3 A code sent via another medium

Some two-factor services use an SMS or emailed code to verify that you have access to a phone number or email address. These are riskier because it is relatively easy to divert SMS messages (by social-engineering your phone company, by using attacks on SS7, or any one of several other methods), and email is often used for password resets (meaning that an attacker who has access to your email may be able to both reset your password and bypass the 2FA request).

2.4 Push confirmations

Some systems use an enrolled phone (or other mobile device) to push a confirmation prompt that you interact with on your phone. These are almost always set up by a third-party authentication provider, as they require additional infrastructure. These prompts can include additional information (such as where you're logging in to, or what transaction you're confirming), which can help confirm those details "out of band" rather than simply trusting what a website says. These prompts can be convenient in some cases, especially if your users interact with your application extremely frequently. However, they almost always require users to download a specific application, which isn't desirable for many uses: this makes them better for cases where out of band confirmations are important (such as banking transactions) or where prompts will be frequent (such as confirming actions by employees). Because they are almost exclusively set up only by authentication-specific providers rather than for a specific application you'd be developing or testing, this post largely doesn't discuss them, but check the "General 2FA Issues" section for issues that can apply to them as well.

3. Testing Two-Factor Systems

There are a lot of somewhat subtle things here that are worth getting right, given the intent of two-factor authentication as a strong security feature. Code-based systems have quite a few

potential pitfalls, but even applications using U2F / WebAuthn tokens have some of these concerns as we'll see below.

For developers of applications considering using two-factor authentication: WebAuthn (or CTAP2 for non-web applications) is the best current option. It's difficult to phish users using this mechanism, and it is resistant to brute-force attacks. If you use this, be sure to allow users to enrol multiple devices as a backup. If you can't require users to have a FIDO2 device, TOTP is the most common alternative, and supported by many applications on various mobile devices. (Third parties also offer mobile applications, often with push mechanisms for triggering 2FA responses, which may be worth a look.) Avoid using SMS as a two-factor solution, as it is often vulnerable to hijacking or social-engineering. Regardless of what you use for your 2FA method, the list below is worth running through. For people evaluating applications with two-factor authentication: here are some specific areas of problems to check for, how to check for them, why they're problematic, and suggested fixes. A number of these assume the application is using TOTP or a very similar protocol, but thanks to Google Authenticator, it's pretty common. (Some companies may roll their own, like Twitter's S/KEY-inspired backup codes, but that's less common.)

Nowadays lots of applications will implement multiple methods for performing two-factor authentication. If you're looking at an application that does, make sure you test all of them: sometimes developers make a mistake in one that they've avoided in another (especially if they were added at different times, or by different teams). For ease of navigating, the checks below are broken up into three categories:

a. General 2FA Issues: these can apply to all 2FA implementations.

b. Authentication Code-Based Issues: these apply to TOTP, SMS, or other mechanisms that require you to type in a code.

c. WebAuthn Security Key Issues: these apply only to WebAuthn security keys.

3.1 General 2FA Issues

These issues and checks can apply to nearly any two-factor authentication system, regardless of whether they use WebAuthn, TOTP, or even something else.

Session State Confusion

Once you've validated your password, can you do anything else in the app with whatever state your session is in before you complete the 2FA authentication?

What? Many applications have been retrofitted to add 2FA support, rather than designed with it from the ground up. If a user authenticates with just a password, before finishing 2FA, the application must track their partial login in some fashion. In some cases, this may allow an attacker to authenticate with just a password, and perform actions as if they were logged in. Alternatively, some sort of cookie or token given to the user for temporary access to the 2FA screen may also be useful for something. (For example, one application I tested gave the same data for "you are temporarily authenticated with just a password" and "this computer is 'remembered' and does not require two-factor authentication", so renaming the cookie would completely bypass two-factor authentication.)

How to test it: This is likely to be somewhat application specific. If you have source code and the application consistently uses a specific authentication mechanism, it may be sufficient to ensure that the authentication mechanism works properly in the "in-between" state. Otherwise, a tool such as wuntee'sAuthz Burp plugin may help test all endpoints with a session that has had a password entered but hasn't yet had its second factor verified How to fix it: Ideally, implement a site-wide authentication handler, and correctly make it only allow access to the 2FA authentication page when a user is half-logged-in. Otherwise, ensure that checks on each action only allow fully authenticated users.

Does waiting on the two-factor page let you ignore password changes?

What? When allowing a user to start signing in with their password and prompting for 2FA, some applications will simply store that the user

is partially logged in. As Luke Berner noticed, this makes it difficult to recover from a stolen password: even changing your password may not keep an attacker from logging in if they had opened the 2FA login flow before your password was changed. (In some cases, this is exploitable even if the victim doesn't intentionally enable 2FA – see Luke's post for more details.)

How to test it:

1. Open two different browsers and go to log in.

2. Get to the 2FA screen in one browser, entering the user's password.

3. In the other browser, log in fully and change the user's password.

4. In the first browser, correctly finish 2FA authentication and try to log in without re-entering the user's password.

5. If you can log in with the first browser without entering the new password, the application is vulnerable.

How to fix it: When a user's password changes, log the timestamp of this change. When starting the 2FA login process with a correct password, also store the timestamp of the 2FA process beginning. Then, upon completing a 2FA login, reject the login attempt if the 2FA process was started before the last password change. (If the application provides a "log off all sessions" feature, this should also be checked when finishing the 2FA flow.)

Remember This Computer

Is it possible to forge the "remember this computer" token? (This is likely different than the "remember that I'm logged in" token.)

What? Not all 2FA applications will support such a feature, but if they do, this is obviously something you should look at more closely. This is generally just a long-lived cookie with a randomly-generated value stored in a database, but it is important to verify that the value is indeed securely randomly generated, and different for each user. (Ideally, the token would be different for each computer, too.

How to test it: You may be able to obtain a number of "remember this computer" tokens and compare them to see how they're constructed (if there's any obvious structure). The token doesn't technically have to change from one "remembered computer" to the next for the same user, so you may need multiple accounts. (Another implementation would be to randomly generate a token for each computer, and store a set of associated computers for each account. This would allow multiple users to "remember" the same computer if desirable.) The easiest way to test this is often by looking at source code.

How to fix it: Any "remember this computer" tokens should be generated using a cryptographically secure random number generator, and should probably be treated as long-lived secrets. 128 bits of entropy should be a decent minimum.

Can you revoke remembered computers?

What? If somebody accidentally sets the "remember this computer" flag on a shared computer, can they remove it? Can they do so remotely – from a different computer? If not, anyone who can copy that token can bypass 2FA for the life of the token.

How to test it: In this case, the user interface pretty much needs to support revoking other computers. It doesn't have to identify them, though: some services, for example, simply let you revoke all previously remembered computers. Dig around for some option for invalidating old remembered computers, and make sure it actually works. (Creating tokens for two old remembered computers and ensuring that they're invalidated should be sufficient.)

How to fix it: If this option doesn't exist, it should be added. Revoking other computers should be done by wiping any stored "remembered device" tokens from the account, possibly replacing them with a newly generated token if a value is necessary.

Re-prompting

Are you prompted for your password and 2FA again before disabling 2FA?

What? 2FA provides additional security for an account. If an attacker is able to gain access to a victim's account (such as via XSS or someone

leaving a computer unlocked), they may be able to simply disable 2FA without re-authenticating. It is also worth mentioning that a user should enter their password before being able to enable 2FA as well, so an attacker can't simply enable 2FA on a victim's account and effectively lock out the victim.

How to test it: Try to disable 2FA. Do you have to enter your password and use two-factor authentication again? The two-factor prompt is more important than the password. (If you disable the two-factor authentication, the password is still required to log in.)

How to fix it: The application should require a password and a two-factor prompt before allowing a user to disable 2FA on their own account. Support staff may want or need the ability to circumvent this restriction, in case somebody loses their second factor, but normal users shouldn't be able to do so.

3.2 Authentication Code-Based Issues

These checks apply to 2FA implementations that use a code you have to type in manually. Where possible, it really can be worth migrating to Web Authn security keys: these keys are resistant to phishing and have far fewer potential pitfalls. However, that isn't always an option for applications (hardware keys are sometimes unsuitable, or users may not have devices that support WebAuthn), so code-based authentication is still a must in many cases.

Secret Communications

Is the secret secure?

What? Some systems perform 2FA checks via generating a code that is sent via a SMS text message or an emailed code. While this is arguably better than doing nothing, it is worth considering whether the security of the way the code is sent matches the threat model of the application. SMS authentication is generally considered insecure, and should be avoided in most cases. Email as a second factor can be appropriate in some situations, but many applications also allow password reset via email, reducing the security of an account to the security of the associated email address.

How to test it: This is simple: if you get a 2FA code via SMS, it isn't being sent securely. If you get a code via email, it likely isn't secure either – or is at least not likely to be more secure than the password reset mechanism, making 2FA pointless.

How to fix it: This is harder: you should carefully consider whether the authentication is sufficient for the purposes of the application, and whether there are any mitigating controls. For example, perhaps you can see your bank balance with an SMS 2FA code, but can only make a withdrawal in person, in which case a SMS may be sufficient for some clients (but perhaps not others). In general, it is worth advising your clients to default to more secure options, even if SMS or email is required by some customers, and then push to deprecate those options over time.

Shared Secret

Is the secret any good?

What? Most code-based 2FA systems rely on a shared secret. If this secret isn't actually cryptographically secure, or is very short, an attacker may be able to guess it. If you have source code access, this should be easy to check, but otherwise, try checking the secret (if you get a QR code, scan it with a barcode scanner, not just the Google Authenticator app) for length and apparent randomness – generate a few, if you can. Testing more than that in a blackbox assessment is pretty difficult, though. How to test it: If you can review the source code, ensure that the secret is being generated from a secure source of randomness, not merely a default random number generator. (Most programming languages' default random number generators are fast but predictable, meaning attackers might be able to figure out which secrets were assigned to which users.) In general, these should be 20 bytes long according to the TOTP spec: longer is fine (but may cause compatibility issues); shorter secrets should be avoided for security purposes. If you can't get access to the source code, try generating several secrets for the same or different accounts. Is each one different? Does any part of the secret appear to be sequential? Based on the account information? Otherwise insecure? (Getting enough secrets to apply proper entropy estimation is likely to be difficult, or at least extremely tedious, and may still not completely answer the question of the source of the secrets.)

How to fix it: Generate secrets from a secure source of entropy. /dev/urandom is nice, but language-specific CSPRNGs are sometimes fine too. Check with a friendly cryptographer if you're not sure.

Reuse

Can you reuse a 2FA code? (Log in to the same account in two browsers with the same code.)

What? The impact here is that someone looking over your shoulder (or potentially an attacker who intercepts your request) could reuse a 2FA code, circumventing the 2FA protection. This can obviously only happen when an attacker gets access to a code around the same time you're using it, but that's still a plausible scenario for many users.

How to test it:

1. Open two different browsers and go to log in. (Alternatively, two profiles in the same browser works as well.)

2. Get to the 2FA screen in both browsers.

3. Obtain a valid 2FA code, and use it to log in with both browsers.

4. If the same code works to log in to both sessions, the implementation is vulnerable.

How to fix it: When checking to see which code was generated by the user, store the code's timestamp in the user record. When logging the user in, allow only code that are newer than the recorded timestamp for the user's most recent 2FA code. Additionally, an error message to the user saying that the code has already been used (rather than simply a generic error) would be useful, as it gives them the opportunity to notice the attack.

Timing

How long is a given code valid? How early can you submit a code?

What? Assuming your codes are time-based, applications generally support a span of time for which the codes are valid, to allow people to type them in slowly, account for clock drift on phones, and so on. Because new codes are almost always generated in steps of 30 seconds (although some systems use 60 seconds), this means that multiple codes are valid at the same time. However, each code in the allowable window is a valid code for the user's account, so allowing codes to be valid for an hour means that lots of codes are valid, and it is consequently easier to guess a valid code. A common validity range here is about five minutes in either direction from the time as set on the server (or 20 codes total), although tightening this up to two or three minutes is nice, at the expense of users with terrible clocks on their phones. Regardless, the main concern here is how many codes are valid at a time: the validity window divided by the time between new codes. How to test it: How long an old code is valid: make sure your 2FA generator's time is roughly correct, generate a code, and try it after several minutes, to determine how long it takes for a code to become unavailable. How early a code is valid: generally, this window is symmetric, but if you have source code, you may be able to determine an actual value. Otherwise, you will have to try generating future codes and testing them. The OATH TOTP generation mechanism is a bit painful to calculate by hand, but there are JavaScript pages (see this one for example, but note that your secret will be leaked to the Google Chart API if you don't comment out line 38) that can help.

How to fix it: It may be appropriate to reduce the validity window for codes, particularly if more than 20 codes are valid at any given time. This can be done by simply adjusting the number of codes that are tested on the server side.

Can you save an old code, wait for a new code, use the new code, and then use the old code?

What? This is a variant of the replay issue noted above. If the service just bans the current code (rather than storing the most recent code's timestamp and rejecting codes that are that old or older), this can be an issue. This attack requires some explanation: if an attacker sees a user's 2FA code generation device, they may see a code that's older than the one that the user chooses to enter. (For instance, the user may think they only have a few seconds to enter the code, and decide to not use it.) In that case, the attacker could use the older code.

How to test it:

1. Open two different browsers and go to log in.

2. Get to the 2FA screen in both browsers.

3. Obtain a valid 2FA code, and write it down. Do not use it yet.

4. Wait for the next valid 2FA code, and log in using it in one browser.

5. In the other browser, try to log in with the old 2FA code that was written down.

6. If you can log in using both sessions, the application is vulnerable.

How to fix it: The correct fix for this is the same as that in the Reuse section above.

Lockout

(Test this with a sacrificial account or at the end of a day – if not the end of a test, lest you find yourself locked out of testing entirely.)

Does the website lock you out of guessing codes frequently?

What? There should be some mechanism for rate-limiting 2FA code guesses. Assuming a standard six-digit code, each guess generally has a (validity window size / 1000000) chance of being correct. Assuming eight codes are valid at any given time, an attacker would only have to guess log(0.5)/log(1-(8/1000000)) = 86644 codes to have a 50% chance of guessing a valid code. While this seems like a large number, it's easy to automate. A CAPTCHA may be worthwhile to limit guesses, but a user should still probably be notified in some way if many guesses are being made, as this probably means that their password has been compromised. (Most 2FA implementations require a correct password before prompting for the second factor.)

How to test it: Attempt to log in with the wrong 2FA code. Do this repeatedly in a short period of time. If the application starts prompting you for a CAPTCHA, or locking you out entirely, it's not vulnerable.

How to fix it: Track 2FA failures per-user on the server side. After several consecutive failures, require a CAPTCHA with the 2FA code. Some applications will instead implement this as a "if there have been more than N attempts in M minutes" requirement as well, which may be acceptable. Particularly sensitive applications should strongly consider alerting the user for several days after any failed 2FA attempt, as an incorrect attempt still implies an attacker may have gotten the user's password.

Can you bypass the lockout by clearing your cookies? Changing your IP address?

What? Sometimes lockout data is tracked in a session cookie (a la BAD_ATTEMPTS=2), or other bad ways of handling this. Sometimes it is tracked by IP address, by developers who haven't considered the existence of botnets or proxies. Either way, if an attacker can easily circumvent the lockout mechanism, it's broken

How to test it: Try clearing your cookies if you get locked out or are prompted for CAPTCHAs. If they go away, the application is vulnerable. Checking for how the lockout information is tracked on the server is best, and will probably require source code review.

How to fix it: Any sort of lockout tracking data should be associated with the account in the database, not handed to the client, or tracked based on any identification of the client, as any of that could be changed.

3.3 WebAuthn Security Key Issues

There's relatively little that can go wrong with WebAuthn that is worth testing on the application side: devices may have vulnerabilities, but that's typically out of the scope of what a web application is defending against. Instead, there is really only one thing worth considering that is specific to WebAuthn implementations:

Multiple Security Key Support

Can the user add multiple security keys to their account?

What? With code-based 2FA, a user may be able to back up their secrets (or use an application that does so automatically), but WebAuthn devices are deliberately difficult – ideally, impossible – to back up. As a result, your users should be able to register multiple security keys on their account, so that one is available as a backup in case another gets destroyed or lost. This might not seem like a

security issue as such, but it can certainly impact the availability of your service for users who lose their key, and it can even dissuade others from setting up 2FA in the first place if they worry about being able to recover from a lost key.

How to test it: This is relatively simple: in the account settings, can you add another security key once one has been added? (Ideally there would be the ability to add several, not just two, for more thorough disaster recovery or usage scenarios.) If not, it's probably a good idea to let people add more.

How to fix it: As above, let users add multiple keys to their account.

4. Conclusions

Two-factor authentication is often a good additional step for security of systems and applications. It isn't a silver bullet, and there are lots of things that can go wrong. We've seen a high-level overview of common 2FA mechanisms, and a variety of potential implementation flaws, as well as how to test for them and how to fix them. If you're writing or testing a two-factor implementation, hopefully they will be of use to you.

- If you're designing a two-factor implementation, the key takeaways are:
- Use WebAuthn if possible (but allow people to add multiple devices): it's resistant to phishing and removes a lot of the pitfalls in other methods.
- Double-check the list of issues above when you're done to make sure you haven't missed something.
- Don't use SMS for two-factor authentication.

References

[1]. http://searchsecurity.techtarget.com/definition/multifactor-authentication-MFA.
[2]. McAfee Case Study "Securing the Cloud with Strong Two-Factor Authentication through McAfee One Time Password" http://www.mcafee.com/in/casestudies/cs-cloudalize.aspx.
[3]. Sharma NA, Farik M. Security gaps in authentication factor credentials. Int J Sci Technol Res 2016; 5: 116–120. [Google Scholar]
[4]. Boyd C, Mathuria A, Stebila D. Protocols for authentication and key establishment. 2nd ed.Berlin, Heidelberg: Springer, 2020, p.521. [Google Scholar]
[5]. Wagenen J, V.The benefits of multifactor authentication in healthcare, https://healthtechmagazine.net/article/2018/12/benefits-multifactor-authentication-healthcare-perfcon (2018, accessed 4 September 2021).
[6]. Alizai ZA, Tareen NF, Jadoon I. Improved IoT device authentication scheme using device capability and digital signatures. In: 2018 international conference on applied and engineering mathematics (ICAEM), Taxila, Pakistan, 04-05 September 2018, pp.1–5. IEEE.
[7]. Newaz AI, Sikder AK, Rahman MA, et al. A survey on security and privacy issues in modern healthcare systems: attacks and defenses. ACM Trans ComputHealthc 2021; 2: 1–44.
[8]. Himanshu Aora, Kiran Ahuja, Himanshu Sharma, Kartik Goyal and Gyanendra Kumar, "Artificial Intelligence and Machine Learning in Game Development", Turkish Online Journal of Qualitative Inquiry (TOJQI), vol. 12, no. 8, pp. 1153-1158, 2021.
[9]. H. Arora, G. K. Soni, R. K. Kushwaha and P. Prasoon, "Digital Image Security Based on the Hybrid Model of Image Hiding and Encryption", 2021 6th International Conference on Communication and Electronics Systems (ICCES), pp. 1153-1157, 2021.
[10]. K. Ahuja, H. Sekhawat, S. Mishra and P. Jha, "Machine Learning in Artificial Intelligence: Towards a Common Understanding", Turkish Online Journal of Qualitative Inquiry (TOJQI), vol. 12, no. 8, pp. 1143-1152, July 2021.
[11]. G. K. Soni, H. Arora, B. Jain, "A Novel Image Encryption Technique Using Arnold Transform and Asymmetric RSA Algorithm", International Conference on Artificial Intelligence:

Advances and Applications 2019. Algorithms for Intelligent Systems, Springer, pp. 83-90, 2020.

[12]. Vipin Singh, Manish Choubisa and Gaurav Kumar Soni, "Enhanced Image Steganography Technique for Hiding Multiple Images in an Image Using LSB Technique", TEST Engineering Management, vol. 83, pp. 30561-30565, May-June 2020.

[13]. Jha, P., Dembla, D. & Dubey, W. Deep learning models for enhancing potato leaf disease prediction: Implementation of transfer learning based stacking ensemble model. Multimed Tools Appl 83, pp. 37839–37858, 2024.

[14]. G. K. Soni, A. Rawat, S. Jain and S. K. Sharma, "A Pixel-Based Digital Medical Images Protection Using Genetic Algorithm with LSB Watermark Technique", Springer Smart Systems and IoT: Innovations in Computing, pp. 483-492, 2020.

[15]. G. Shankar, V. Gupta, G. K. Soni, B. B. Jain, & P. K. Jangid, “OTA for WLAN WiFi Application Using CMOS 90nm Technology”, International Journal of Intelligent Systems and Applications in Engineering, 10(1s), pp. 230-233, 2022.

[16]. Babita Jain, Gaurav Soni, Shruti Thapar, M Rao, "A Review on Routing Protocol of MANET with its Characteristics, Applications and Issues", International Journal of Early Childhood Special Education, Vol. 14, Issue. 5, 2022.

[17]. Jha, P., Dembla, D., Dubey, W. (2023). Crop Disease Detection and Classification Using Deep Learning-Based Classifier Algorithm. In: Rathore, V.S., Piuri, V., Babo, R., Ferreira, M.C. (eds) Emerging Trends in Expert Applications and Security. ICETEAS 2023. Lecture Notes in Networks and Systems, vol 682. Springer, Singapore.

[18]. P. Jha, T. Biswas, U. Sagar and K. Ahuja, "Prediction with ML paradigm in Healthcare System," 2021 Second International Conference on Electronics and Sustainable Communication Systems (ICESC), pp. 1334-1342, 2021.

[19]. P. Upadhyay, K. K. Sharma, R. Dwivedi and P. Jha, "A Statistical Machine Learning Approach to Optimize Workload in Cloud Data Centre," 2023 7th International Conference on Computing Methodologies and Communication (ICCMC), pp. 276-280, 2023.

[20]. H. Arora, M. Kumar, T. Rasool and P. Panchal, "Facial and Emotional Identification using Artificial Intelligence", IEEE 6th International Conference on Trends in Electronics and Informatics (ICOEI), pp. 1025-1030, 2022.

[21]. Himanshu Aora, Kiran Ahuja, Himanshu Sharma, Kartik Goyal and Gyanendra Kumar, "Artificial Intelligence and Machine Learning in Game Development", Turkish Online Journal of Qualitative Inquiry (TOJQI), vol. 12, no. 8, pp. 1153-1158, 2021.

[22]. T. A and U. A, "Generative AI: A Transformative Force in Business Intelligence," 2024 2nd International Conference on Intelligent Data Communication Technologies and Internet of Things (IDCIoT), pp. 1234-1240, 2024.