Not in The Prophecies: Practical Attacks on Nostr

Hayato Kimura
NICT / The University of Osaka
Osaka, Japan
hytkimura@protonmail.com

Ryoma Ito
NICT
Tokyo, Japan
itorym@nict.go.jp

Kazuhiko Minematsu
NEC
Kanagawa, Japan
k-minematsu@nec.com

Shogo Shiraki
University of Hyogo
Hyogo, Japan
4w3tag185mpja@gmail.com

Takanori Isobe
The University of Osaka
Osaka, Japan
takanori.isobe@ist.osaka-u.ac.jp

***Abstract*—Distributed social networking services (SNSs) recently received significant attention as an alternative to traditional, centralized SNSs, which have inherent limitations on user privacy and freedom. We provide the first in-depth security analysis of Nostr, an open-source, distributed SNS protocol developed in 2019 with more than 1.1 million registered users. We investigate the specification of Nostr and the client implementations and present a number of practical attacks allowing forgeries on various objects, such as encrypted direct messages (DMs), by a malicious user or a malicious server. Even more, we show a confidentiality attack against encrypted DMs by a malicious user exploiting a flaw in the link preview mechanism and the CBC malleability. Our attacks are due to cryptographic flaws in the protocol specification and client implementation, some of which in combination elevate the forgery attack to a violation of confidentiality. We verify the practicality of our attacks via Proof-of-Concept implementations and discuss how to mitigate them.**

***Index Terms*—Nostr, plaintext recovery attack, forgery attack, key replace attack, Cache-based Forgery Attack, CBC-mode**

1. Introduction

Nostr [1] is an open-source distributed network protocol for distributed social networking services (SNSs). SNSs have grown as an essential communication tool among people. However, traditional centralized SNSs limit user privacy and freedom. In addition, as exemplified by 𝕏 (formerly Twitter), people are more aware of issues of centralized SNSs, such as sudden specification changes and unclear policies on handling user accounts. This situation attracts attention to distributed SNSs as a viable alternative. Among many existing distributed SNSs (e.g., Bluesky, Mastodon, Secure Scuttlebutt), Nostr received significant attention. For example, Twitter co-founder Jack Dorsey invested 14 Bitcoin (about 245,000 dollars) in Nostr in 2022, followed by a 5 million dollar donation in May 2023 [2]. Development of Nostr started in 2019 by an anonymous person `@fiatjaf`. As of October 2024, it has around 1.1 million registered users and 38 thousand weekly active users in [3]. Notably, Edward Snowden is considered to be a user of Nostr[1]. Since the protocol is fully open-source, a number of client implementations exist. On iOS, Damus [4] is currently the most major Nostr client application. It was released in 2023 on the App Store and garnered widespread attention. It is estimated to have 160,000 Damus users as of May 30, 2023 [5]. Additionally, among Android users, a popular client application is Amethyst [6], which has been downloaded by over 100,000 users. Other well-known popular client applications include Iris [7], FreeFrom [8], and Plebstr [9] (See Appendix B for details). Moreover, Nostr is applied to building not only a distributed SNS environment but also an e-commerce environment, and its further development is expected in the future.

The designers of Nostr aimed to design their protocol to be simple and censorship-resistant. The latter is achieved by connecting the client nodes to the relay servers that do not possess users' secrets. The protocol is specified in a series of documents called NIPs (Nostr Implementation Possibilities), which are available on GitHub. Following these NIPs, a number of implementations exist for both clients and relay servers. Nostr introduced several security features, such as message signing and encrypted direct messages (DMs) between users.

1.1. Our Contributions

In this paper, we present the first in-depth security analysis of the Nostr protocol and its popular (client) implementations. Through our investigation, we assume that their goal is end-to-end encryption (E2EE). To reflect the basic security goals of E2EE, we assume two threat models: a malicious user (MU) and a malicious server (MS). For both models, the goal is to forge or eavesdrop on the content created by an honest user (a victim), such as a DM to another user or a user profile, without knowing that user's secret key. As a result, we found a number of vulnerabilities leading to practical attacks that violate integrity and confidentiality. After our initial investigation, we also checked the latest updates in some major NIPs and evaluated their effect (NIPs for encrypted DMs, see Appendix D.5).

1. https://x.com/Snowden/status/1620790688886718466

Our findings show that Nostr has serious security issues that could be practically exploited. We give brief descriptions of our representative attacks below.

1) We present a forgery attack by a malicious server in Section 3. The attack is possible due to the lack of authenticity for public keys, as there is no official/formal means of checking them. The flaw is simple and effectively collapses the authenticity of any communication on Nostr. The attack is generic, simple, and basically undetectable.
2) We present two forgery attacks by a malicious Nostr user in Section 4. The first attack (basic attack), shown in Section 4.1, is possible due to the lack of signature verification for most communication events, which is found in some major OSS client applications, including Damus and Iris. This flaw is not in the Nostr specifications, so the attack target is limited to the users of affected client applications. The first attack does not work for encrypted DMs. To overcome this limitation, we present the second attack (advanced attack), shown in Section 4.2, that allows a malicious user to forge encrypted DMs between two users of affected client applications. This attack additionally exploits a common pitfall of encryption, namely the use of plain, unauthenticated encryption for applications needing message integrity. Nostr uses AES-CBC with a 256-bit key for the encryption of DMs. However, no MAC is employed. The attack is a variant of well-known CBC malleability attacks (see Section 1.2). A CBC malleability attack needs known plaintext/ciphertext block pairs of CBC. For that purpose, we develop a practical, dedicated method by exploiting another vulnerability of incomplete key separation in the Nostr protocol. By taking into account the nature of DMs (namely, their shortness, as their typical usage is chat), we are able to mount a universal forgery on encrypted DMs.
3) We present attacks on confidentiality against encrypted DMs in Section 5. This attack exploits the link preview mechanism in Nostr, namely a URL link contained in an encrypted DM is automatically given to the corresponding web site to create a preview. When this preview creation is done by the recipient's side, Bakry and Mysk [10] showed that the preview creation leaks an IP address of the link to outsiders for E2EE messaging applications in general. Stivala and Pellegrino at NDSS 2020 [11] showed a phishing attack based on a link preview. We found that preview creation at the recipient's side happens for the most widely-deployed client, Damus. Even more, our attack is more damaging than what the existing works [10], [11] imply. We found that, by combining with the aforementioned CBC malleability, a malicious user could either recover a non-domain part of a link (e.g., authentication information such as password) for an encrypted DM containing that link with probability one or the *whole message* in an encrypted DM (not necessarily containing URL link) with a non-negligible probability.
4) We present a forgery attack by a malicious server on profile events in Section 6. This attack targets the users of a major client application, Damus. This attack exploits an *inadequate cache search* vulnerability whereby Damus uses the sender-provided id field instead of recomputing it, allowing a previously verified id to bypass signature checks. Because the profile's content field contains Bitcoin transfer instructions in plaintext, the adversary can forge and redirect Bitcoin transfers to themselves.

For each attack, we discuss its feasibility and impacts in the corresponding section. We also found various issues and variations of our attacks. They are reported in Appendix E.

While some vulnerabilities are purely inherent to NIPs, we found that the effectiveness of our attacks also depends on the client applications, as each has unique features independent of NIPs. For this reason, we also conducted an extensive survey of available OSS/non-OSS clients to determine the applicability of our attacks. Table 1 summarizes our attacks that target legitimate users. Notably, Damus, a popular client application mentioned above, has been affected by most of our attacks. Finally, the practicality of all our attacks shown in Table 1 has been verified by our Proof-of-Concept (PoC) implementations tested in a closed environment. In addition, our ethical disclosure process and status of revisions are shown in Appendix A.1.

Relevance and Challenges of Our Work. Despite significant attention, the security of Nostr has not been thoroughly investigated to date. Although NIPs do not explicitly define clear security goals, the security features implemented in Nostr suggest that certain security goals, including E2EE, are implicitly incorporated into its design (See Section 2.3 for details). We remark that the security of distributed SNSs remains an important open problem as the adversary models and assumptions differ significantly from those of centralized SNSs. Thorough security analysis of distributed SNSs in a similar manner to analysis of existing E2EE communication systems (e.g., [12]–[19]) will also provide useful lessons to developers of distributed SNSs with E2EE features.

We faced some technical challenges in conducting our analysis which is somewhat unique to Nostr. In more detail, a client application is required to implement only the basic protocol specification (NIP-01, Section 2.1.1), and all other NIPs are optional. There is a number of clients and they are customized to fit their own unique features; thus, investigating the entire Nostr specifications alone is insufficient. We instead need to investigate all the major client implementations to clarify the gaps between the specifications and their implementations.

In fact, some vulnerabilities arise from gaps between the specifications and their implementations, which can be attributed to unclear descriptions in the NIPs specifications and the absence of implementation guidelines. For example, the NIP-01 specification does not mandate signature verification and does not specify when verification has to be done. This ambiguity has led many clients to ignore signature verification (Vulnerability 2), or allows the adversary to bypass signature verification due to inadequate cache search in Damus (see Section 6).

Nostr is composed of multiple sub-protocols, and our work reveals that Nostr's design largely overlooked interactions between its sub-protocols, and this interaction can yield vulnerabilities. For example, our forgery attack

TABLE 1. SUMMARY OF OUR ATTACKS AGAINST NOSTR. MU: MALICIOUS USER. MS: MALICIOUS SERVER. TESTING WITH IOS 17.1 AND ANDROID 11.

Attack	Sect. ref.	Model	Type: Forgery	Type: Breaking confidentiality	Target: Profile	Target: Contact List	Target: Encrypted DMs	Target: The Bitcoin transfer feature	Prerequisites (✓: required): All Event (Note (Tweet) etc...)	Prerequisites (✓: required): Vuln.1. Lack of Authenticity for Public Key	Prerequisites (✓: required): Vuln.2. Lack of Signature Verification	Prerequisites (✓: required): Vuln.3. AES-CBC without MAC	Prerequisites (✓: required): Vuln.4. Lack of Key Separation	Prerequisites (✓: required): Vuln.5. Domain Name Leakage	Prerequisites (✓: required): Vuln.6. Automatic Link Preview Generation	Prerequisites (✓: required): Vuln.7. Inadequate Cache Search	Affected clients (✓: affected, -: not affected): Damus (iOS) v1.5(8)	Damus (iOS) v1.6 (29), revised version	Amethyst (Android) v0.80.7	Iris (Web), ver.App v0.2.2	Iris (iOS) ver.2	FreeFrom (iOS) v1.2.3	FreeFrom (Android) v1.2.3	Plebstr (iOS) v0.7.6+56	Plebstr (Android) v0.7.6+56
Forgery Attack (Unauthenticated Public Key)	3	MS	✓		✓				✓								✓	–	–	–	–	–	–	✓	✓
—"—	3	MS	✓			✓			✓								✓	✓	–	–	–	–	–	✓	✓
—"—	3	MS	✓				✓		✓								✓	✓	–	–	–	–	–	✓	✓
Forgery Attack (Lack of Signature Verification)	4.1	MU	✓		✓					✓							–	–	–	–	–	✓	✓	✓	✓
—"—	4.1	MU	✓					✓		✓							✓	✓	–	–	✓	✓	✓	✓	✓
Encrypted DM Forgery	4.2	MU	✓			✓				✓	✓	✓					✓	✓	–	–	✓	✓	✓	✓	✓
URL Recovery Attack on DM	5.1	MU		✓		✓				✓	✓		✓	✓			✓	✓	–	–	–	–	–	N/A	✓
URL Recovery Attack (Variant)	5.2.2	MU		✓		✓				✓	✓		✓	✓			✓	✓	–	–	–	✓	✓	N/A	✓
Generic Message Recovery Attack	5.3	MU		✓		✓				✓	✓		✓	✓			✓	✓	–	–	–	–	–	–	–
Inadequate Cache Implementation	6	MS	✓		✓											✓	✓	✓	–	–	–	–	–	–	–
Inadequate Cache Implementation	6	MS	✓				✓									✓	✓	✓	–	–	–	–	–	–	–
Generic Replay Attack	E.2	MU	✓			✓				✓							✓	✓	–	–	✓	✓	✓	✓	✓
Truncated Replay Attack	E.3	MU	✓			✓				✓	✓						✓	✓	–	–	✓	✓	✓	✓	✓
Breaking Robustness of Damus	E.4	MS	✓		✓											✓	✓	–	–	–	–	–	–	–	–

on encrypted DMs (NIP-04, Section 2.1.2) is caused by the absence of signature verification defined in NIP-01, as well as by the Nostr connect feature (NIP-46, Section 2.1.3) being a duplicate of NIP-04 with a lack of key separation.

1.2. Related Work

Security analysis on modern communication tools with (end-to-end) encryption features is an active research topic [12]–[19].

While a large body of the research listed above is conducted on the publicly available specification documents, there are also studies based on reverse-engineering to uncover unknown/hidden specifications and vulnerabilities rooted in the implementations [14], [20]–[23].

Some of our attacks (as described in Section 4.2 and Appendix E.3) exploit the well-known malleability of CBC mode [24]–[26]. This malleability has been exploited by a number of real-world protocols/applications. Those include various Internet Protocols, e.g., [27]–[30], and encryption schemes inside XML [31], ASP.NET [32], S/MIME [33], and PDF [34]. Arbitrary code execution against CBC-encrypted binaries was also reported [35].

Some of our attacks (as described in Section 4.2) exploit the lack of key separation between sub-protocols. Similar flaws were found in Threema by Paterson, Scarlata, and Truong [17], and Matrix by Albrecht et al. [14].

Some of our attacks exploit the absence of verification of public keys. A similar flaw was found in Apple's iMessage by Garman et al, [12].

Our attacks on confidentiality (plaintext recovery) (Section 5) exploits the malleability of CBC mode and the exfiltration channels. Similar flaws were found in email messages by Poddebniak et al. [33]. We discuss the relation of our attack to their attack in Section 5.5.

Furthermore, our attacks in Section 5 exploit link previews. As mentioned earlier, Bakry and Mysk [10] and Stivala and Pellegrino [11] showed privacy/phishing attacks based on link preview in E2EE messaging applications. These attacks do not directly imply our attacks, as ours are non-trivial combinations of link preview and CBC malleability.

2. Preliminaries

2.1. Nostr

The Nostr specifications (NIPs) are publicly available on GitHub [36], enabling their use in various client applications, such as the open-source clients Damus [37], Amethyst [6], and Iris [7], as well as non-OSS such as FreeFrom [8] and Plebstr [9]. Additionally, Nostr allows anyone to set up a "relay", a server that relays posts, DMs, profiles, etc. A relay server is mostly OSS, making it possible for anyone to set it up, and the list for implementations is available[2].

NIPs have 56 documents as of December 6, 2023. Among them, this study focuses on specific aspects outlined in NIP-01: Basic Protocol Flow Description (see Section 2.1.1), NIP-04: Encrypted Direct Message (see Section 2.1.2), and NIP-46: Nostr Connect (see Section 2.1.3). Note that we investigated all the NIPs (as of December 6, 2023) and did not find any significant security issues from other NIPs. After our investigation, a new specification of encrypted standards, NIP-44 [38], [39], was published on December 20, 2023, along with an audit report by Cure53 [40]. The relationship between this specification and our work is presented in Appendix D.5. Although it is outside the scope of our investigation, we remark that no significant issues have been identified with this specification thus far. In addition, we investigate specific Nostr client implementations, especially on profile data format (see Section 2.1.4), link preview (see Section 2.1.5), Bitcoin transfer procedure (see Appendix C), cache implementation (see Section 2.1.6), and block list implementation (see Appendix. E.5).

2. https://github.com/aljazceru/awesome-nostr

2.1.1. NIP-01: Basic Protocol Flow Description. NIP-01 defines the basic data format, signature scheme, and communication procedures between clients and relay servers. Only NIP-01 is mandatory, and others are optional. This NIP describes the fundamental specification for other NIPs and establishes the core protocol that should be universally followed (see `01.md` in [36] for details). Subsequent NIPs may define new optional or mandatory fields, messages, and functionality in addition to the basic structure and protocol flow defined in this NIP.

The user holds a master key for secp256k1 locally. This master key serves as the user's identity. Events are used for posting content, updating a contact list, and sending and receiving encrypted DMs.

The process of generating an Event in NIP-01 is outlined below. The secp256k1 key generation is expressed as

$$\{k_{priv}^{\mathsf{user}}, k_{pub}^{\mathsf{user}}\} \leftarrow \mathsf{secp256k1.KeyGen}().$$

An Event is represented in a specific JSON format, including the following elements: id, pubkey, create_at, kind, tags, content, and sig. Below is an example of posting text as a microblog.

$$\begin{aligned}
\mathsf{pubkey} &\leftarrow k_{pub}^{\mathsf{user}},\\
\mathsf{create_at} &\leftarrow \text{UNIX timestamp in second},\\
\mathsf{content} &\leftarrow \text{“Hello”},\\
\mathsf{kind} &\leftarrow 1,\\
\mathsf{tags} &\leftarrow \{\},\\
\mathsf{data} &\leftarrow [0, \mathsf{pubkey}, \mathsf{create_at}, \mathsf{kind}, \mathsf{tags}, \mathsf{content}],\\
\mathsf{id} &\leftarrow \mathsf{SHA\text{-}256}(\mathsf{data}),\\
\mathsf{Event} &\leftarrow \{\mathsf{id}, \mathsf{pubkey}, \mathsf{create_at}, \mathsf{content}, \mathsf{kind}, \mathsf{tags}\},\\
\mathsf{sig} &\leftarrow \mathsf{Schnorr.Sign}(k_{priv}^{\mathsf{user}}, \mathsf{Event}).
\end{aligned}$$

The specifications for the values of id, pubkey, create_at, and sig are mandatory fields, while kind, tags, and content depend on their respective NIPs.

Here, id or Event-id is a unique identifier generated as a SHA-256 hash value of the serialized event data, represented as a 32-byte lowercase hex-encoded string; pubkey is the public key of Event creator, represented as a 32-byte lowercase hex-encoded string; create_at is a UNIX timestamp in seconds; tags and content are data related to the sender's message content, represented as arbitrary strings; and sig is a signature of the id value, represented as a 64-byte lowercase hex-encoded string. kind is a type of Event data, represented as a number.

In the context of user interactions, each user possesses a signing key for the Schnorr signature based on the secp256k1 curve to compute a sig.

NIP-01 does not require the authentication of a user for retrieving Events. Thus, any users can get Events (e.g., posted content, encrypted DMs, etc) communicated among third parties.

2.1.2. NIP-04: Encrypted DM. NIP-04 specifies how to encrypt DMs between two users. This NIP supports only one-to-one encrypted DMs. As described in Section 2.1.1, the data format of the encrypted DMs must be the same as the Event format defined in NIP-01. Then, a sender needs to embed a ciphertext into the content field of this Event in the following procedure:

1) Compute $sk \leftarrow \mathsf{secp256k1.ECDH}(k_{priv}^{\mathsf{Bob}}, k_{pub}^{\mathsf{Alice}})$
2) Generate $iv_{\mathsf{Bob}} \xleftarrow{\$} \{0,1\}^{128}$.
3) Compute $C \leftarrow \mathsf{AES\text{-}CBC.Enc}(sk, M, iv_{\mathsf{Bob}})$.
4) Generate Event according to NIP-01.
5) $\mathsf{Event.content} \leftarrow C \| iv_{\mathsf{Bob}}$.
6) $\mathsf{Event.kind} \leftarrow 4$.
7) $\mathsf{Event.tags} \leftarrow [\text{'}p\text{'} : k_{pub}^{\mathsf{Alice}}]$.
8) Compute id and Sign Event according to NIP-01.

A pair of public and private keys for the ECDH key exchange is the same as the signing key pair for the signature scheme defined in Section 2.1.1. For example, in Damus, the public key or the verification key is displayed on the user's profile screen.

The DM's integrity and the sender's authenticity rely on the signature defined in NIP-01. The scheme that combines such symmetric encryption and digital signatures is known as Encrypt-then-Sign.

2.1.3. NIP-46: Nostr Connect. NIP-46 provides an option for connecting an external web application (WebApp), which is assumed as a client application used on the user's web browser. Assuming that a user uses a WebApp temporarily, the simplest way to sign an Event on the WebApp is to incorporate the user's private (or signing) key on the daily-used client application into the WebApp. However, this implies that the WebApp gains full access to the user's account due to getting the user's private key. To avoid this problem, NIP-46 offers a feature that allows a user to compute the signature defined in NIP-01 without sharing the user's private key on the daily-used client application to the WebApp. Such a client application can hold a user's signing key; hence, technically, it can sign without asking for the user's consent. However, in the NIP-46 protocol, the signer displays a popup to the user and asks for consent to sign. In that case, a NostrApp is referred to as a signer (see `46.md` in [36] for details).

The initialization sequence for the Nostr connect is executed in the following procedure:

1) A user, Alice, obtains the public key $\mathsf{PubKey}_{\mathsf{App}}$ and URL for a WebApp from a QR code to connect the WebApp from Alice's device. To validate the signer functionality, the following URI is displayed from the QR code as an example: `nostrconnect://<PubKey_App>&relay=<URL for WebApp>&metadata=<WebAppName>`.
2) The signer uses the Alice's secret key $\mathsf{SecKey}_{\mathsf{Alice}}$ and $\mathsf{PubKey}_{\mathsf{App}}$ to generate a shared secret sk (i.e., a 256-bit key for AES-CBC encryption), which is the same procedure in NIP-04.
3) The signer sends a connect request message, which is encrypted metadata using AES-CBC with sk, to the WebApp. The following is an example of the metadata format: $\mathsf{meta} = \{\mathsf{id}, \mathsf{method}, \mathsf{params}\}$. The id is generated by converting a random floating-point number to a string and extracting the portion after the decimal point[3]; the method is a task the WebApp requires of the signer, e.g., requesting a connect (connect), getting the public key (get_public_key), and signing an Event (sign_event); params need to insert the specified parameters for each method, e.g., the public key for the connect task.

3. https://github.com/nostr-connect/connect/blob/7723e2694b03d5c435741c3682b2e65aa419e805/src/rpc.ts#L195

After completing the above sequence, the user is able to send any Event to the WebApp. The user first sends an Event that the user-generated on the WebApp to the signer. Next, the signer checks the content of the Event, which does not include its signature. The signer then approves the sending of the Event and generates its signature. Finally, the signer sends the Event with its signature to the WebApp, and the WebApp broadcasts it to the relay servers. These steps are executed every time an Event is sent to the WebApp. We remark that the NIP-46 functionality is not implemented in Damus, but the Damus users can still use it through a Nostr connect client application, such as Nostrum[4].

2.1.4. Profile Data Format in Damus. The profile in a Nostr client usually displays the username, a self-introduction, the user's public key, an icon for transferring Bitcoin (e.g., a lightning bolt icon in Damus), etc. The definition of the profile data format may vary slightly depending on the client application because NIPs do not define its details.

In the following, we detail some of the specifications unique to Damus [37]. Its profile data format is represented in a specific JSON format, including the following elements: display_name, name, website, lud06 or lud16, picture, banner, and about. These elements are embedded into the content field of an Event. The profile data is treated as the Event data; thus, this implies that the signature functionality is applied to the profile data.

The user's wallet information for the Bitcoin transfer is written in the lud06 or lud16 element. The Bitcoin transfer via profile in Damus is briefly described in Appendix C.

2.1.5. Link Preview. Link preview is a feature that automatically retrieves and displays summaries of webpages, including images and text, when a URL is embedded in a message. This allows recipients to get a glimpse of the content without having to click on the link itself.

There are two main schemes for generating link preview: server-side and client-side. Server-side generation is commonly used for public posts on social media platforms. A key advantage of this approach is that the recipient's IP address remains hidden from the website being previewed. In contrast, E2EE messengers typically rely on client-side generation because the server cannot decrypt the URL. Bakry and Mysk highlighted that client-side generation can lead to the unintentional disclosure of the user's IP address to the website being previewed [10]. To address this privacy concern, messaging applications like Signal, iMessage, and WhatsApp do not generate link previews on the recipient's side.

Link preview has not been defined as a specification in Nostr and depend on the implementation of each client. We identified that Damus, a specific Nostr client implementation, uses client-side link preview generation, and it is performed by both the sender and the recipient. In this paper, we focus on client-side link preview generation on the recipient's side, as it's a common approach that can reduce traffic on the E2EE channel.

Figure 1 illustrates the sequence of client-side link preview generation. Recipient Bob performs the link

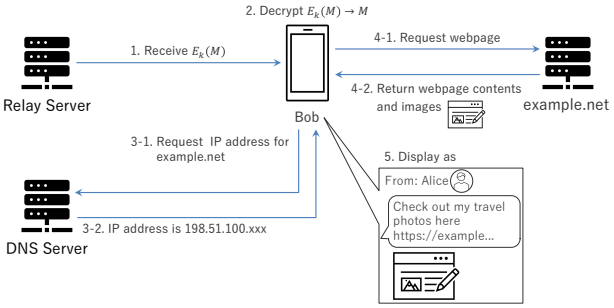

Figure 1. Client-side link preview generation for E2EE messages.

preview generation process when the message M from Alice contains a website URL (Steps 1 and 2). In the subsequent steps, the client queries the DNS to obtain the website's IP address from the domain part of the URL and retrieve the webpage's contents (HTML and images) for generating a preview.

2.1.6. Cache Implementation. In Damus, a cache implementation is deployed for the signature verification[5]. An overview of the steps in the signature verification is as follows:

1) A pair of Event-id and its signature is stored in an on-memory local database as a cache. We note that a signature of an Event data is computed from Event-id, as described in Section 2.1.1.
2) When a user receives an Event, the user checks whether the cache hits for the Event-id or not. If the cache hits, the user does not need to execute the signature verification for the received Event. Otherwise, the user must do the signature verification.

In the context of SNSs, it's natural for clients to repeatedly access mutable items such as user profiles. Besides, Nostr does not provide a mechanism for notifying updates to the Event's content. Therefore, clients can only determine if a profile has been updated by retrieving the Event. We speculate that Damus has implemented such caching strategies to bypass the verification of signatures from previously received Events to conserve battery life.

2.2. Adversary Models

To analyze the security of Nostr protocol from a cryptographic viewpoint, we adopt the adversary model of existing studies on E2EE systems, such as Isobe and Minematsu [13] and Paterson, Scarlata, and Truong [17] with minor modifications to fit into our targets.

Definition 1. (Malicious User) *A malicious user, who is a legitimate Nostr user and possesses his/her own user key pair, tries to break subsequently defined security goals by deviating from the protocol.*

Definition 2. (Malicious Server) *A malicious server, that behaves like a legitimate relay server, is able to intercept, read, and modify* Events *over the network deviating from the protocol.*

4. https://github.com/nostr-connect/nostrum

5. damus/damus/Models/HomeModel.swift – damus-io/damus

In any Nostr application, one or more relay servers are registered by default, providing the option for anyone to build a relay server and make it available to other users. For the adversary holding a (malicious) server, one way to let the target users (i.e., victims) use that malicious server is to gain the trust of the developers and have them register it as a default relay server. For instance, in a realistic scenario, the adversaries might operate a relay server seamlessly until a certain time, subsequently elevating it to the status of a default relay server for a specific Nostr application. Following this promotion, the adversaries could then replace it with a relay server program designed for malicious purposes. In essence, this implies that anyone could assume the role of a malicious server.

2.3. Security Goals

The current NIPs lack clearly defined security goals [36]. However, it is natural to expect a sort of basic (E2EE) security for Nostr as a nature of a distributed network with open relay servers. In fact, the draft NIP-44 document [36] that will substitute NIP-04 (for DMs) wrote "The main goal is to have at least some way to send messages between Nostr accounts that cannot be read by everyone." and its PoC implementation clearly mentioned E2EE[6]. Damus claims E2EE in App Store[7] as "End-to-End encrypted private messaging". Similar to the adversary models, we set the security goals based on the existing related works [13], [17] as follows:

Definition 3. (Confidentiality) *The* content *field of an* Event *issued by a legitimate Nostr user must be kept secret from any unauthorized entity.*

Definition 4. (Integrity) *Data integrity ensures that the content received as an* Event *has not been altered if its digital signature is successfully verified.*

Definition 5. (Authenticity) *Entity authenticity must be maintained. Namely, if the requested content is received as an* Event *and its digital signature is successfully verified, the content was indeed sent by a particular Nostr user.*

From the specification, a natural expectation is that confidentiality of encrypted DMs should be ensured by AES-CBC, and data integrity/entity authentication of all Events should be ensured by the Schnorr signature.

3. Forgery Attack Based on Lack of Authenticity for Public Key

This section explains how a malicious server can forge Events (e.g., profiles, contact lists, and encrypted DMs) from a legitimate user. The attack is simple but can significantly affect all Nostr-based SNS features in a realistic scenario. This attack exploits the following vulnerability caused by a specification issue in the Nostr protocol; thus, it could potentially be applied to all applications based on the Nostr protocol.

Vulnerability 1. (Lack of Authenticity for Public Key) *The authenticity of* pubkey *in received* Event*s is not verified by servers or clients. To be more precise, the public key associated with the user will be used for the signature verification. However, neither the relay server nor the receiver client verifies the authenticity of* pubkey *information embedded in the received* Event.

6. https://github.com/paulmillr/nip44/tree/main
7. https://apps.apple.com/us/app/damus/id1628663131

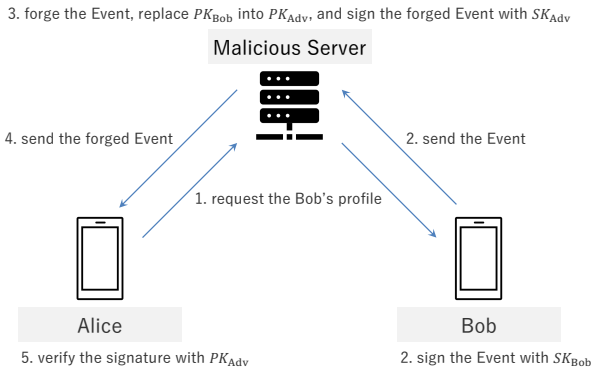

Figure 2. Concrete attack flow for executing our forgery attack on profile information.

We have confirmed that the NIPs [36] do not mention anything about implementing a process of verifying the authenticity for the pubkey embedded in the received Event, which is a crucial property for using public-key cryptography. Hence, Vulnerability 1 implies that even if the signature verification works properly for all Events, there may be a potential risk that a victim will successfully accept the forged Event. This simple flaw significantly undermines the whole authenticity of communications via Nostr.

3.1. Attack Procedure

Given that Vulnerability 1 is inherent in the Nostr protocol, our attack should be theoretically executable for all Events. As an example case, we present a generic forgery attack by a malicious server on the profile information of a victim, Bob. Figure 2 illustrates a flowchart of our attack. The detailed attack procedures are as follows:

1) A user, Alice, asks a malicious server (the adversary) for Bob's profile information.
2) The adversary requests Bob to send his profile Event. This Event is signed with Bob's secret key SK_{Bob}, and the corresponding public key PK_{Bob} is embedded into the pubkey field of the Event.
3) The adversary replaces PK_{Bob} embedded in the received Event into the adversary's public key PK_{Adv}, forges the content field of the Event (i.e., Bob's profile information), recalculates the id value and its signature with the adversary's secret key SK_{Adv}, and embeds these values into the forged Event.
4) The adversary sends the forged Event to Alice.
5) Alice verifies the signature in the received Event with the public key embedded in the Event (i.e., PK_{Adv}).

3.2. Discussions

3.2.1. Feasibility. Assuming a malicious server as an adversary, it is obvious that the adversary works within

its capabilities during Step 3 in the above procedure. The other steps also respect the Nostr protocol; hence, this attack is hard to detect by observing the server's actions on the network. At Step 5 of the attack, Alice always successfully verifies the signature in the forged the Event as it is indeed signed by SK_{Adv}, and the corresponding public key, PK_{Adv}, is embedded into Event. Considering that anyone can play the role of a malicious server, as explained in Section 2.2, our attack is highly practical and feasible in a realistic scenario.

Our attack is similar to the *key substitution attack* on iMessage by Garman et al. [12]. Following their work, we refer to our attack as a *key substitution attack* on the Nostr protocol. We remark that Garman et al.'s attack assumes a man-in-the-middle (MitM) adversary on the TLS connection between the target clients. However, this assumption seems to impose a certain limitation on practicality because a typical PKI mechanism that prevents MitM could be deployed to the TLS network. In contrast, our attack relies on a realistic malicious server assumption as a MitM adversary on the Nostr protocol without any additional assumptions. Therefore, our attack has a significant practicality as a realistic application.

3.2.2. Attack Model. In the aforementioned attack, the adversary needs to directly send a forged Event to the target user. Only the relay servers have such authority; hence, the attack model is a malicious server. A malicious user who does not own its server could not mount the attack. Still, a malicious user can perform a similar attack passively by exploiting the creation of a user ID. Due to the lack of space, we briefly describe this attack variant in Appendix E.1.

3.2.3. Impact. While most SNS features (profiles, contacts, posts) are publicly exposed, Nostr's lack of Event creator authentication (Vulnerability 1) enables a significant attack. In this attack, a malicious server can manipulate events (content) associated with a user's profile, contacts, or posts (Figure 2). This can harm user privacy (e.g., fake Bitcoin info) or reputation (e.g., inappropriate posts). Encrypted DMs are not directly affected as the adversary lacks the decryption key. The simplicity and versatility of this attack highlight the need for a fix in Nostr's event authentication.

4. Forgery Attacks Based on Lack of Signature Verification

We present a class of forgery attacks by a malicious user that affects all kinds of Events, except the profiles communicated between two users. We classify these into two types: a basic forgery on all Events excluding profiles and encrypted DMs (Section 4.1) and an advanced forgery on encrypted DMs (Section 4.2).

4.1. Basic Forgery Attack on All Events

This attack exploits the following vulnerability in some client implementations.

Vulnerability 2. (Lack of Signature Verification) *In Damus and Iris, the signature is not verified unless a profile* Event *is received.*

We investigated the source codes of open-source clients and, as stated above, found this vulnerability in Damus [37] and Iris [7]. We did not find one in Amethyst [6]. By black-box tests, we found some non-OSS clients (e.g., FreeFrom and Plebstr) also contain this vulnerability.

This vulnerability implies that all the users of the affected client skip signature verification for any Event except profiles, such as posted contents, contact lists, and encrypted DMs. Thus, a malicious user is able to forge the content field of the Event communicated between two users of the affected clients as follows:

4.1.1. Attack Procedure. A malicious user can execute the following attack procedure:

1) The adversary intercepts an arbitrary (but not a profile) Event communicated between the target users.
2) The adversary modifies some fields (e.g., content) of the intercepted Event. The sig field may be filled with an invalid signature.
3) The adversary sends the forged Event to the receiver.

4.1.2. Feasibility. The targeted user (victim) always accepts the forged Event as long as it is not a profile or an encrypted DM. When an Event content involves encrypted DMs, the adversary cannot forge it into an arbitrary message as it is encrypted by AES-CBC. The victim can notice the attack because the decryption of the forged encrypted DM would be a (at least partially) random-looking string. We provide a further discussion in Appendix D.1.

4.2. Advanced Forgery Attack on Encrypted DMs

We present a more advanced attack to overcome the aforementioned limitation for forging encrypted DMs. This attack exploits the following vulnerability in the Nostr protocol together with Vulnerability 2.

Vulnerability 3. (AES-CBC without MAC) *All DMs are encrypted by AES-CBC with a 256-bit key, but a MAC algorithm is not employed.*

Thus, signature verification is the only measure to ensure the integrity of encrypted DMs, while some client applications skip the signature verification, implying a complete loss of authenticity. The remaining problem is how to forge a CBC-encrypted message.

It is well-known that a ciphertext of CBC mode can be forged, i.e., there is no integrity of the ciphertext and its corresponding IV [24] (see also Section 1.2). To forge a plaintext block (or an IV), the adversary only needs to know a single plaintext/ciphertext (P/C) block pair. Our attack works in a similar manner.

We discuss here how to obtain P/C pairs with a high probability in a general setting. A direct approach is to guess a typical sentence, such as "Hi" or "Hello". Since a malicious user has the same authority as a legitimate user and can obtain encrypted DMs between other parties, such an approach is basically feasible. This approach generally works with a passive adversary that collects all the encrypted DMs among many users and tries to attack any one of them. However, such a guessing approach is not always successful when one wants to attack a specific user, as the number of obtained DMs would be limited.

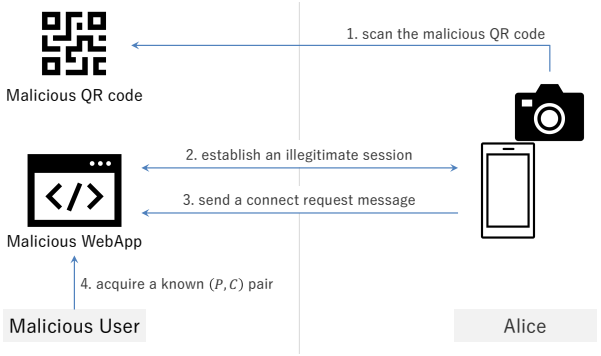

Figure 3. Concrete scenario for acquiring a known plaintext/ciphertext, (P/C), pair by exploiting Vulnerability 4 in the Nostr connect.

For this, we propose an alternative (active) attack approach that allows a malicious user to obtain a P/C pair for any target user. This exploits the following vulnerability in the Nostr protocol specification, especially on the NIP-04 (encrypted DMs) and NIP-46 (Nostr connect) described in Section 2.

Vulnerability 4. (Lack of Key Separation) *The session establishment method for encrypted DMs is also deployed in other protocols, such as Nostr connect. The public–private key pair used in this session establishment is reused by all of these protocols.*

We consider a situation where the adversary exploits Vulnerability 4 to obtain a P/C pair for the shared secret between Alice and Bob. We take the initialization sequence in the Nostr connect as an example of such a situation. See Section 2.1.3 and Figure 3 for the details of the Nostr connect and our expected scenario, respectively. The concrete scenario is as follows:

1) An adversary deviates from the legitimate protocol by embedding Bob's public key $\mathsf{PubKey}_{\mathrm{Bob}}$ into the QR code instead of the WebApp's public key $\mathsf{PubKey}_{\mathrm{App}}$, and makes Alice scan the QR code. In this case, the following URI is displayed from the QR code as an example: `nostrconnect://<PubKey_Bob>&relay=<URL for WebApp>&metadata=<WebAppName>`.
2) A signer on Alice's device uses Alice's secret key $\mathsf{SecKey}_{\mathrm{Alice}}$ and $\mathsf{PubKey}_{\mathrm{Bob}}$ to generate a shared secret sk, which is reused in encrypted DMs between Alice and Bob.
3) The signer must send a connect-request message in the beginning, which is encrypted with sk. In a plaintext of the connect-request message (i.e., metadata), the id field must be a random string, but the method and params fields are the following fixed sentences: "method : connect, params : $[\{\mathsf{PubKey}_{\mathrm{Alice}}\}]$".
4) The adversary obtains both the fixed sentences as a plaintext and the corresponding ciphertext, forming a desired P/C pair.

From the lengths of fixed sentences, the obtained P/C pair has the size of at least two blocks (32 bytes), hence it enables a forgery attack on the CBC mode in the same manner as previous studies (Section 1.2). Considering the above scenario, potential issues of Vulnerability 4 can be summarized in Appendix D.2.

The inclusion of a forged QR code in the above scenario is strictly a hypothetical assumption to facilitate the attack aimed solely at acquiring a known P/C pair. In other words, once the initialization sequence of NIP-46 is complete, it enables powerful, unlimited, and persistent DM forgery attacks, even if the malicious WebApp is disconnected from the Nostr account. Besides, an adversary can only impersonate the target user by chaining both Vulnerabilities 2 and 3, exploiting either one in isolation is insufficient to forge an encrypted DM. In contrast, a simple malicious WebApp without our methodology would require a signer (user) approval each time a post is made, thereby increasing the likelihood of suspicious messages being rejected by the user and the disconnection of the malicious web application. From this point of view, our attack is more powerful than a simple malicious WebApp.

4.3. Discussions

4.3.1. Feasibility. A major difference between ours and existing CBC malleability attacks lies in the target application and the attack goals. For example, the FIM20 attack [35] targets binary executable files, whose sizes are typically larger than one block (in the case of AES, 16 bytes), and changing the IV is often difficult for the adversary. For these reasons, the CBC malleability attack inevitably makes some decrypted blocks (more precisely, the previous block of the forged block) random unless the target is one block. While the FIM20 attack showed how to circumvent this, it is application-specific and not generally applicable.

Our attack focuses on encrypted direct messaging (DM) applications, where messages are naturally short and often fit within a single block. Our analysis of sent DMs on the Nostr platform (see Appendix D.3) confirms that many messages are indeed a single block. Our method works with single- or multi-block inputs while producing a single forged block. In other words, even if the attacker possesses a P/C pair of multiple blocks, the attack is limited to creating a single-block forgery.

In addition, an adversary can easily change the IV in some client applications if Vulnerabilities 2 and 3 could be exploited. If this is the case, a one-block plaintext can also be arbitrarily chosen by changing the values of the corresponding one-block ciphertext and the IV.

Using a similar idea to the aforementioned attacks, a (generic) replay attack on encrypted DMs, possibly with truncation, is possible. We briefly describe this attack variation in Appendices E.2 and E.3.

4.3.2. Impact. Assuming the ASCII characters are used in DMs, our attack enables us to arbitrarily create a forged message of up to 16 characters in theory. This holds for Damus; however, the effective maximum message length may be shorter due to the internal padding (PKCS#7) for other clients. See Appendix D.4. We remark that this length limitation is sufficient to mount a more powerful attack by colluding with a malicious server that can execute the forgery attack on the target user's profile described in Section 3. For example, if Alice and Bob use Damus, a malicious server performs the forgery attack on Bob's profile information (say, Bitcoin transfer information) in the

beginning, and then a malicious user performs the forgery attack on encrypted DMs. This results in a forgery such that the malicious server changes Bob's Bitcoin transfer information to the adversary's one. Then the malicious user impersonates Bob and forges an encrypted DM to promote a Bitcoin transfer from Alice to Bob (e.g., "Plz give me 1BTC"). The forgery attacks on Bitcoin transfer information and encrypted DMs are feasible under the existence of a malicious user and a malicious server, respectively; therefore, this combined attack is also feasible if a malicious user and a malicious server collude.

5. Plaintext Recovery Exploiting Link Preview

We present plaintext recovery attacks on encrypted DMs exploiting the link preview feature in E2EE by a malicious user. These attacks are categorized into URL recovery, which retrieves URLs, and generic message recovery, which targets any message content, not limited to URLs.

Link preview introduces two key vulnerabilities that can be exploited to leak information from encrypted messages.

Vulnerability 5. (Domain Name Leakage) *During link preview, the recipient's device retrieves the webpage corresponding to the URL in the encrypted message. This involves sending an unencrypted HTTP GET request, including the domain name.*

Specifically, the domain name is included in two unencrypted parts: the Server Name Indication (SNI) field of the TLS Client Hello message and the query made to the DNS during the HTTP GET request. Thus, a passive adversary monitoring network traffic can potentially intercept these unencrypted portions and recover the domain name from the URL in the encrypted message, even if the content itself is hidden by TLS encryption.

Vulnerability 6. (Automatic Link Preview Generation) *Link preview is automatically executed when the user decrypts an E2EE message or when the message is displayed on the screen.*

This automatic generation of a link preview allows an adversary to trigger an attack simply by displaying a message on the screen. In other words, it is possible to trigger an HTTP GET request to a malicious URL and leak information to an adversary, regardless of the user's security literacy.

Our attacks leverage a combination of link preview information leakage (Vulnerability 5), automatic link preview generation (Vulnerability 6), and a 1-block CBC forgery (Vulnerability 3). This demonstrates how seemingly minor vulnerabilities can be chained together to achieve a crucial attack.

5.1. URL Recovery Attack

We consider a malicious user on the network who aims to recover secret URLs contained within encrypted DMs between Alice and Bob (Victim). These secret URLs are often used in various applications like cloud storage or web conferencing. For example, it is formatted like

https://example.net/secret?q=shared-url-token.

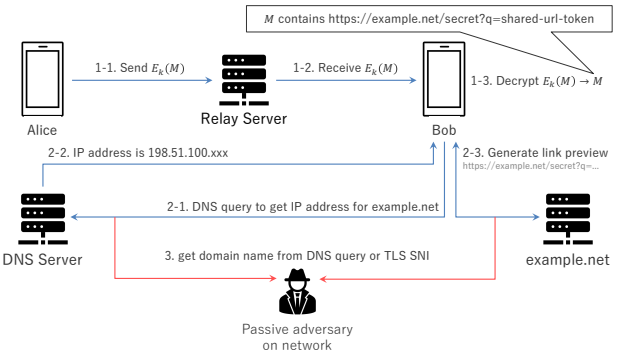

Figure 4. First phase in URL recovery: Exploiting domain name leakage via DNS or TLS SNI.

The attack consists of two phases. The adversary first tries to reveal part of the URL and then recovers the remaining part by exploiting CBC forgery.

5.1.1. First Phase. The adversary observes the DNS query automatically executed by link preview and obtains a pair of P/C of the DM being attacked.

Figure 4 illustrates a flowchart of these steps, and the detailed procedures are as follows.

1) Alice sends an encrypted DM containing a secret URL to Bob via a relay server. Bob receives and decrypts the message. Meanwhile, the adversary observes the communication between Alice and Bob.
2) The link preview feature of Bob's device automatically initiates a DNS query to resolve the domain name embedded in the URL contained in the decrypted message. Once the DNS query is completed, the link preview is generated by communicating with the target web server.
3) By monitoring network traffic, the adversary captures both the DNS query message and the TLS SNI. Since the domain name appears in a plaintext manner within these messages (e.g., https://example.net/), the adversary is able to extract the known plaintext corresponding to the DM.

Although the communication between the clients and the relay servers is protected by TLS, Nostr relay servers typically do not enforce user authentication. As a result, an adversary can issue unauthenticated queries to these servers and receive responses containing arbitrary user data, including encrypted direct messages ($E_k(M)$) compliant with the NIP-04 specification.

Finally, the adversary correlates the obtained ciphertext and plaintext by leveraging observable metadata that is retrievable from the server without user authentication (e.g., timestamps, sender information, and message size), in addition to the DNS packet capture timing. This correlation enables the adversary to associate the plaintext (e.g., the secret URL) with the corresponding ciphertext.

5.1.2. Second Phase. The adversary proceeds to obtain the secret parts other than the domain name (e.g., the path and query parameters). Figure 5 illustrates a flowchart of these steps, and the detailed procedures are as follows.

1) By leveraging a CBC forgery attack, the adversary modifies only the domain name portion. For instance,

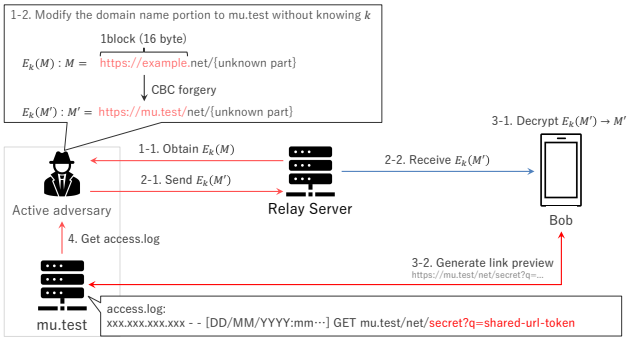

Figure 5. Second phase in URL recovery: Leveraging a CBC forgery attack.

the adversary changes the domain from example.net to mu.test, where https://mu.test/ is a website under the adversary's control. This is realized by manipulating an IV and obtaining a one-block P/C pair of the target DM.

2) The adversary injects this manipulated DM into the communication between Alice and Bob.
3) When Bob receives the manipulated DM and decrypts it, the link preview feature automatically attempts to retrieve a webpage from the adversary's website (i.e., mu.test). Crucially, since the CBC forgery attack only modified the domain name, the original path and query parameters (i.e., secret?q=shared-url-token) remain intact within the manipulated URL.
4) The adversary gains access to the path and query parameters by monitoring the access logs on their server hosting https://mu.test/. These logs record attempted visits to the website, revealing the complete secret URL even though the content itself remains encrypted.

The adversary combines the intercepted domain name with the leaked path and query parameters to reconstruct the complete secret URL, i.e., https://example.net/secret?q=shared-url-token.

5.2. Discussions for URL Recovery

5.2.1. Feasibility. For the first phase, the adversary first collects the necessary data using DNS queries or TLS SNI. This can be executed by a passive network adversary. For example, an adversary connected to the same LAN as the victim can perform packet sniffing on DNS queries, or an adversary on the network path, such as on a router, can eavesdrop on packets. This allows the adversary to obtain the P/C pair of the target DM.

For the second phase, since Nostr's relay servers usually do not require user authentication from clients, a malicious user can easily obtain the encrypted DMs between Alice and Bob. This enables the adversary to freely obtain the ciphertext that contains URLs as a message.

5.2.2. Variation. Our attack works not only for link preview of webpages but also for link preview of images on image hosting services. While limited to image content types, leaked images might contain sensitive information, potentially compromising user privacy. In addition, the domain name of the image upload server is hard-coded into FreeFrom. An adversary can use this as a known plaintext. In this case, the first phase (Section 5.1) can be omitted.

5.2.3. Impact. The secret URL contains authentication information and is widely used in cloud storage services, web conferencing tools, and private chat invitations. To the best of our knowledge, most of these invitation URLs rely solely on the URL for access control without requiring additional authentication. Given this background, our attack is not merely a simple URL compromise but is expected to lead to the compromise of more critical confidential information.

5.3. Generic Message Recovery Attack

We extend the URL recovery attack to a generic message recovery attack that can recover the plaintext of any arbitrary DM, regardless of its content. The assumptions and procedures are almost the same as those of the URL recovery attack variants. The key difference lies in the manipulation of the initial block.

In the generic message recovery attack, the adversary leverages an attacker-controlled domain to prepend an arbitrary plaintext prefix to the DM. In other words, instead of simply modifying the initial block to reflect an adversary's website (e.g., `https://mu.test/`), the attacker forces the first block to begin with the URL prefix `http://m.test?q=`. As a result, even if the original plaintext message is not a URL, it is transmitted as a query parameter value to the adversary's website, thereby leaking the plaintext.

Let $C = (C_1, C_2, \ldots, C_n)$ and IV denote an n-block CBC ciphertext and the corresponding initialization vector with a block cipher E taking the key K shared by Alice and Bob. The adversary forges the ciphertext as follows:

1) First, the adversary forges the first plaintext block so that it becomes the desired attacker-controlled prefix:
$$\tilde{P}_1 = \texttt{http://m.test?q=}.$$
2) Next, by utilizing any available one-block known P/C pair (P'_i, C'_i), where
$$C'_i = E_k(P'_i \oplus C'_{i-1}),$$
the adversary computes a forged IV, $\tilde{IV}$, such that:
$$E_k(\tilde{P}_1 \oplus \tilde{IV}) = C'_i.$$
3) Finally, the adversary composes the forged ciphertext $\tilde{C} = (\tilde{C}_1, \tilde{C}_2, \ldots, \tilde{C}_{n+2})$, where $\tilde{C}_1 = C'_i$, $\tilde{C}_2 = IV$, and $\tilde{C}_{j+2} = C_j$ for $j = 1, 2, \ldots, n$.

When the recipient decrypts $\tilde{C}$, a first block $\tilde{C}_1$ is decrypted to the attacker-controlled prefix $\tilde{P}_1$ (i.e., `http://m.test?q=`), and the subsequent blocks $\tilde{C}_2$, $\ldots$, $\tilde{C}_{n+2}$ follow the original CBC chaining, thereby preserving the remainder of the original message.

Consequently, even if the original plaintext message of the DM is not a URL, the forged ciphertext causes it to be sent with the attacker's prefix. This means that the plaintext message is transmitted as a query parameter value to the adversary's website (i.e., `http://m.test`), resulting in the entire message being leaked.

5.4. Discussions for Generic Message Recovery

5.4.1. Feasibility. $\tilde{P}_2$ will be an unpredictable value because $C_2 \neq \tilde{C}_2$ holds with a high probability. Then, the attack succeeds if $\tilde{P}_2$ can be decoded into a valid URL string. The probability of this string being valid in the URL is determined. In the case of ASCII code, 79 characters are available (alphabets, digits, and 17 types of symbols); thus, the probability that all 16 bytes of a block are valid URL characters is $(79/256)^{16} \approx 2^{-27.14}$. which is a non-negligible probability.

It is important to note a limitation of this attack. If the decrypted message contains characters invalid for URLs (like spaces), only the valid characters preceding them are leaked to the adversary's server. For example, when leaking the message "Hello World", the decryption result on the victim's device would be http://m.test?q=...Hello World. The space character is invalid in URLs, so only "Hello" is transmitted, while "World" remains undisclosed.

This potentially implies that languages (e.g., English, Hindi, Spanish, and French) that use spaces to separate words may limit the information that can be leaked. However, if the URL parser supports the interpretation of UTF-8 strings and the DM content is written in non-segmented languages (e.g., Chinese, Japanese, or Thai), more information might be treated as part of the URL and leaked. We have confirmed that this actually works on Damus. It will be shown in Appendix G.4.

5.4.2. Impact. This attack can be applied to almost all messages. Since encrypted DM exchanges are likely to contain messages that users want to keep confidential, compromising the confidentiality of such data can lead to significant privacy issues. Moreover, if the compromised data includes information related to the user's personal attributes, it could be exploited for social engineering attacks such as phishing.

5.5. Relationship with EFAIL

The EFAIL [33] attack by Poddebniak et al. has demonstrated broader plaintext recovery techniques in the context of encrypted emails. In that work, attackers have to construct complex HTML payloads using carefully forged <a> tags and other markup to trigger automatic decryption and resource retrieval by email clients. In contrast, the target of our work is messaging platforms, that typically do not depend on HTML. It generates link previews automatically whenever a message contains a valid URL starting with http/https. This means that simply including such a URL in a DM is sufficient to trigger the preview mechanism, resulting in conditions under which plaintext recovery can be achieved with minimal effort.

Furthermore, Poddebniak et al.'s CBC/CFB gadget attack generates multiple random plaintext blocks by reordering ciphertext blocks. In their email-based scenario, the diversity of supported character encodings enables clients to tolerate a significant amount of garbled text. In contrast, our approach is designed for chat applications that generally enforce a single, fixed character encoding (e.g., UTF-8). To limit the generation of unwanted random plaintext blocks, we utilize query parameters to produce only one such block.

While our experiments and evaluations are conducted within the Nostr environment, the underlying approach is generally applicable to any messaging system employing a preview feature without message integrity. Thus, our technique could be interpreted as a specific instance of the broader plaintext recovery framework of the EFAIL, tailored to the operational details of the messaging applications. In this sense, our contribution lies in tailoring existing concepts for messaging applications—leveraging the simplicity of automatic link preview triggering—to complement prior research while addressing emerging threat scenarios.

6. Forgery Attack Based on Inadequate Cache Implementation in Damus

As described in Section 4.1, the signature is verified when receiving a profile Event in Damus. Nevertheless, a malicious server connected by the Damus users can perform a forgery attack on the profile information of a legitimate Damus user. This attack exploits the following vulnerability caused by Damus's cache implementation.

Vulnerability 7. (Inadequate Cache Search) *The cache search in Damus uses the* id *value embedded into the* id *field of the target* Event *by the sender instead of the* id *value computed from the received* Event.

As described in Section 2.1.6, Damus deploys the cache implementation on the signature verification for a profile Event. Vulnerability 7 implies that the adversary can potentially make a target profile Event bypass from the signature verification by forging the id field of the Event in such a way that the id value is hit by the cache search. Since the content field of the profile Event contains the Bitcoin transfer information (i.e., the lud06 or lud16 element) in a plaintext form, bypassing the signature verification enables the adversary to forge a victim's Bitcoin transfer information into the adversary's one. That is, the malicious server (adversary) is able to "steal" Bitcoin from a Damus user.

6.1. Attack Procedure

Figure 6 shows a flowchart of our forgery attack on a profile Event, including Bitcoin transfer information. Our attack consists of preparation and attack phases. The details of the attack procedure are as follows:

Preparation Phase (Steps 1–1 to 1–3 in Figure 6):

1) A victim, Alice, requests the server to send the current Bob's profile. Here, this server does not need to be under the control of an adversary.
2) The server sends the correct Bob's profile Event to Alice. This Event includes $\mathsf{id}_{\mathsf{Bob}}$, content, sig, etc. Bob's profile data is embedded into the content field.
3) Alice checks whether the cache hits for the received $\mathsf{id}_{\mathsf{Bob}}$ or not. If the cache misses, Alice verifies the attached signature for the received Event and then stores a pair of $\mathsf{id}_{\mathsf{Bob}}$ and the corresponding verification result in the cache list on Alice's device.

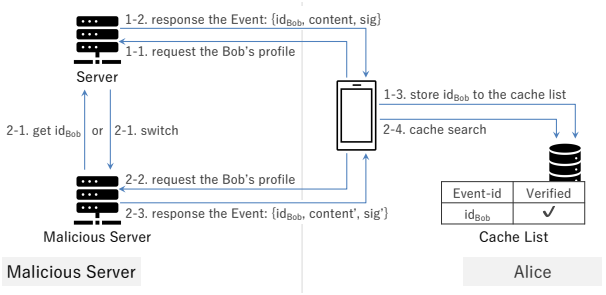

Figure 6. Concrete attack flow for bypassing the signature verification of a profile Event. Our attack consists of the preparation and attack phases. The preparation is further divided into three steps (Steps 1–1 to 1–3), whereas the attack phase is further divided into four steps (Steps 2–1 to 2–4).

Attack Phase (Steps 2–1 to 2–4 in Figure 6):

1) If the server used in the preparation phase is under the control of an adversary, after switching the server to a malicious server, the adversary can obtain $\mathsf{id}_{\mathsf{Bob}}$ stored in the server. Otherwise, the adversary having a generic user authority can obtain $\mathsf{id}_{\mathsf{Bob}}$ from the server as a legitimate user.
2) As in the preparation phase, Alice requests the malicious server to send the new Bob's profile.
3) The malicious server sends the forged profile Event to Alice. In other words, this Event contains the same $\mathsf{id}_{\mathsf{Bob}}$ value obtained in the preparation phase, but the content embedded in Bob's profile data is forged by the adversary. We note that embedding an arbitrary value into the sig field causes no problem.
4) Alice checks whether the cache hits for the received $\mathsf{id}_{\mathsf{Bob}}$ or not. The forged profile Event that Alice receives in this phase always bypasses the signature verification since a pair of $\mathsf{id}_{\mathsf{Bob}}$ and the corresponding verification result has already been stored in the cache list on Alice's device.

As described before, the content field of a profile Event contains clearly Bitcoin transfer information. Therefore, the adversary can effectively steal Bitcoins sent from Alice to Bob.

6.2. Discussions

6.2.1. Feasibility. Considering the attack scenario described in Section 2.2, a malicious server can originally behave as a legitimate (honest) server and intentionally store Event data created in response to a request from any Damus users. Even if the malicious server is different from a legitimate server used in the preparation phase, the malicious server can collect many profile Events by querying them to the legitimate server or by intercepting them over the communication between the legitimate server and the victim(s). As we mentioned in Section 2.1.6, it is expected that clients will repeatedly access mutable items such as user's profiles. From this point of view, the forgery attack on profile information is feasible by the malicious server.

We remark that a malicious user who does not own a (malicious) server cannot forge an arbitrary profile Event even with Vulnerability 7. This is because only the servers have the authority to send all the Events data. In addition, sending a repeating id value from the legitimate server is forbidden in the Nostr protocol (as id is a hash value of data containing a timestamp). For these reasons, our attack requires a malicious server.

6.2.2. Attack Variation. Vulnerability 7 also enables a malicious server to break the robustness property of communications made by Damus. Once the robustness property is broken, a victim will not be able to get the target user's profile until the target user updates his/her profile information. Appendix E.4 briefly describes this attack variant.

6.2.3. Impact. Similar to the combined attack shown in Section 4.2, assuming the existence of a malicious Damus user and a malicious Damus server, the attacks in this section would also cause a significant risk that Bitcoin may be unintentionally transferred to the adversary.

7. Mitigation

We have identified several vulnerabilities from the perspective of both specification issues in the protocol and implementation issues in the client applications. In general, it takes a lot of time to fix some issues from a specification aspect, whereas in many cases, these issues can be quickly fixed from an implementation aspect. From this fact, we recommend fixing the issues from an implementation aspect in the client applications prior to fixing the underlying specification issues in the protocol; then, we suggested countermeasures from the implementation aspect as immediate mitigation to the developers during the disclosure process. It is expected that many issues can be fixed without burdening users or impairing interoperability with relay servers and clients. In this section, we summarize three countermeasures to fix the issues from the implementation aspect in the client applications.

In Section 3, which is noteworthy, we concluded through the analysis of the PoC (Section 8) that the root cause of the vulnerabilities lies in the fact that important specifications, from the perspectives of both cryptographic systems and distributed SNSs, are not clearly defined.

From the perspective of cryptographic systems, key authenticity is lacking at the specification level. We discuss a fundamental solution to this issue in Section 7.1. Additionally, from the perspective of distributed SNSs, there is no specification for the consistency and deduplication of Events received from multiple relay servers on the client side. This is an issue that can be resolved solely through client-side implementation, as explained in detail in Appendix F.1.

In addition, we suggest countermeasures from the specification aspect as recommended mitigation, which are discussed in Appendix F due to the page limitation.

7.1. Provide Key Authentication in Cryptography Layer

To avoid Vulnerability 1 we believe it is necessary to implement a mechanism that ensures the authenticity of

public keys. We recommend the adoption of out-of-band (OOB) authentication or key transparency [41] instead of the decentralized identifiers (DIDs) [42] or public key infrastructure (PKI). While DIDs are highly useful as a decentralized authentication system, they are very expensive if their sole purpose is to provide key authentication. Moreover, constructing a PKI privately is extremely costly, and the reliability of public keys decreases with the compromise of root certification authorities.

In OOB authentication, the sender and receiver exchange key fingerprints, which are hexadecimal characters, numeric codes, or QR codes. Key fingerprints are calculated from the hash values of their public keys through an out-of-band channel. Most messaging applications (e.g.Signal, WhatsApp, Telegram, Threema) have employed key fingerprints as OOB authentication [43]. They then verify the public keys by checking the hash of the public keys used in the in-band channel key sharing against the key fingerprint. Unlike typical secure messaging systems, Nostr does not rotate the master keys. Thus, once a master key is authenticated, the user can trust it for an extended period as long as the authenticated master key is not compromised. Therefore, we consider that OOB authentication might be a mitigation in Nostr. However, prior studies have pointed to the need for user interaction as a disadvantage of OOB authentication [44]–[46]. Additionally, it is not effective in cases where one wants to use random ephemeral keys or when either the sender or receiver is not a human (e.g., scenarios like establishing end-to-end encrypted connections between web services and a user under NIP-46 and NIP-47). So, we believe that OOB is a solution for enhancing the authenticity of public keys in Nostr. However, it should be noted that there are limitations.

Key transparency is a mechanism that enables automatic public key authenticity checks by publishing public key audit logs. WhatsApp and iMessage have employed key transparency as a strategy to balance user experience and security with the authenticity of the public key [47], [48]. Regular checks of the audit logs allow the sender, receiver, and service providers (or relay servers) to verify the authenticity of public keys through key transparency. The kind of key transparency, CONIKS [41], enables each service provider to maintain a public ledger of user keys, and by auditing each other, the reliability of audits is enhanced. However, Yadav et al. have pointed out that today's secure messaging services are each operated by a single provider, and they pointed out that potentially there is no guarantee that they perform an honest audit under this assumption [49]. Meanwhile, we believe that CONIKS is suited for distributed SNSs because different service providers operate different relay servers, and the prerequisites for securely implementing CONIKS can be met.

In further discussion, it might be reasonable to implement auditors on the client. This is because, from a financial cost perspective availability, Wei and Tyson pointed out that 95% of free-to-use Nostr relays would fail to meet their operational costs through the donations [50]. Thus, we think many Nostr servers operated for non-profit purposes might be difficult to pay auditors. Therefore, we believe that implementing key transparency with anonymous client auditors [49], which allows clients to perform their own audits of key transparency logs, can mitigate this problem. However, as far as we know, there has not been sufficient discussion to apply it to distributed SNSs, and further study is needed for their implementation on Nostr.

Overall, we recommend implementing key transparency or OOB authentication. However, this may not be the best solution from the perspective of service operational costs and the federation of distributed SNSs. We believe further study is needed to balance security and implementation considerations.

7.2. Signature Verification for All Events

The primary reason for the forgery attacks described in Section 4 is the lack of consistent signature verification when receiving certain Events, such as profiles, posted contents, contact lists, and encrypted DMs. In many cases, client applications only verify signatures selectively, which opens opportunities for adversaries to exploit this inconsistency. To address this issue effectively, it is essential to implement signature verification consistently across all Events, ensuring the integrity and authenticity of the communications in line with the intended security model of the NIPs.

These vulnerabilities in the Nostr protocol stem from discrepancies between the NIP specifications and their implementations, primarily due to ambiguous descriptions and the lack of clear implementation guidelines. For example, the NIP-01 specification neither mandates signature verification nor specifies when it should be performed. This ambiguity has resulted in many clients neglecting signature verification entirely (Vulnerability 2), while others, like Damus, selectively perform signature verification, but even the implemented verification process can be bypassed due to insufficient cache search mechanisms (see Appendix 6). These gaps highlight the critical need for more rigorous implementation of signature verification across all Events in the protocol.

However, the Nostr community has expressed concerns about the potential performance impact of comprehensive signature verification. Increased processing times and resource consumption, especially on mobile devices, could negatively affect user experience[8]. Nevertheless, real-time performance is not a critical requirement for SNS, making the trade-off for increased security feasible. This is evidenced by applications like Amethyst [6] and Iris (Web) [7], which have successfully implemented full signature verification without significant user impact.

The adoption of more advanced cryptographic solutions, such as aggregate signatures, could further alleviate performance concerns by reducing the overall signature size and verification time. The draft specifications for the Ethereum consensus layer, for instance, propose BLS-based aggregate signatures [51], which could be considered as a model for optimizing signature verification in Nostr. While specific methods for implementing aggregate signatures in Nostr require further investigation, such approaches offer a promising path toward achieving a balanced solution that addresses both security and performance challenges.

In summary, mitigating the vulnerabilities caused by inconsistent signature verification requires both clear im-

8. Snort post on February 15, 2023 (UTC+0).

plementation guidelines in the NIPs and practical measures in client applications. Enforcing consistent and reliable signature verification can enhance both the security and robustness of the Nostr protocol. Further exploration of aggregate signatures in Nostr may offer a balanced solution for both security and performance.

7.3. Disabling Link Preview on the Recipient

As demonstrated in Section 5, the automatic generation of link preview exposes vulnerabilities (Vulnerability 6) that can be exploited for plaintext recovery. As a countermeasure, disabling link previews on the receiver side is effective in preventing our attacks within E2EE communication. Messaging applications like iMessage, Signal, and WhatsApp are not affected by our proposed attack, as they have already adopted a design where link previews are generated only by the sender. This design change was introduced as a mitigation against the privacy issues related to IP address leakage, as reported by Bakry and Mysk [10].

7.3.1. The id recalculation for Profile Event. The mitigation for the attack in Section 6 is the id recalculation for Profile Event. The main factor is that the id value embedded in the received profile Event is used for the cache search due to the Damus implementation issues by Vulnerability 7. To fix this issue, when a user receives a profile Event, the client application should recalculate the id value by the correct method, and its value should be checked the integrity with the id value embedded in the received Event. If the integrity check passes, the client application can use its id value to move to the cache search. Otherwise, an error message should be displayed on the user's device, and the received Event should be discarded.

8. Proof of Concept

Table 1 lists our PoC results to determine whether our attacks affect the target client applications. The details of our PoC environment are presented in Appendix G.1. Our PoC source code is available on GitHub[9]. Here, we present a brief summary of the PoC results.

We have confirmed that the following attacks work as expected based on the proposed attack procedure: the basic forgery attack (Section 4.1), the forgery attack based on an inadequate cache implementation especially in Damus (Section 6), and other attacks (Appendices E.2–E.5). Conversely, we have gained new insights that the PoC tests for the following attacks yield somewhat different results from our analysis at the protocol level even though the test perfectly simulates the proposed attack procedure: the forgery attack based on the lack of authenticity for a public key (Section 3) and the advanced forgery attack on encrypted DMs (Section 4.2). Due to the page limitation, we explain the details of these two PoC tests in Appendices G.2 and G.3.

Fortunately, our PoC tests revealed that Amethyst and Iris(Web) are resistant to all of our attacks. In addition, it has no Vulnerability 2. Hence, these prevent the attacks caused by Vulnerabilities 1 and 2. Iris(Web) has relatively complex situations. It appears that Iris has already specified the servers from which a user can request a profile or contact list. In other words, our PoC test cannot simulate a malicious server, especially in the attack caused by Vulnerability 1. Moreover, we had confirmed that the signature verification was not implemented in the source codes of Iris(iOS), but our PoC test revealed that it is implemented in Iris(Web); therefore, our attack caused by Vulnerability 2 is not executable for Iris(Web).

We discuss both mitigation that can be implemented to the client immediately (Section 7) and fundamental mitigation (Appendix F).

In Table 1, the PoC test for plaintext recovery attacks (Section 5) shows Plebstr (iOS) as 'N/A'. This is because Plebstr removed the DM function from its client before we completed the PoC test for this attack. However, we can run a PoC test on the old version of Plebstr (Android) since we have a backup of the virtual machine for the test environment. As of July 2024, Plebstr has been rebranded as Openvibe and is being run as an SNS client without the DM function.

Finally, to demonstrate the effectiveness of all mitigation proposed in Section 7, we applied each mitigation to Damus and Iris, which are the OSS applications, and conducted PoC tests in the same way as above. Among target applications, FreeFrom and Plebstr are non-OSS applications; we couldn't apply each mitigation to them in our local environment. As a result, we have confirmed that each mitigation effectively defends against the corresponding attack.

9. Conclusion

Distributed SNS is a rather new technology. Despite the significant attention and increasing number of users, security analysis on distributed SNSs is rarely done, in particular from the cryptographic perspective. We believe that ours is the first of its kind, although a large part of our results (flaws and attacks) have similarities to the existing analysis on E2EE mechanisms of real-world applications, and some attacks are embarrassingly simple. Our results on Nostr show that their use of cryptographic technologies is simple and immature, showing a sharp difference from the modern messaging applications that the research community has scrutinized. We think there is a significant lack of understanding on the secure design and analysis of distributed SNSs: what security property should be set, and what about the security of popular growing services other than Nostr, such as Mastodon and BlueSky? Considering the practical impact (say people exchange Bitcoin over Nostr), this will be an essential topic for society. We hope our work will encourage future research in that direction.

Acknowledgment

Takanori Isobe has been supported by JST AIP Acceleration Research JPMJCR24U1 Japan and JSPS KAKENHI Grant Number JP24H00696.

References

[1] Nostr, 2023, https://nostr.com/.

9. https://github.com/crypto-sec-n

[2] Forbes, "Jack dorsey donates $10 million to turn bitcoin into 'the native currency of the internet'," 2023, https://www.forbes.com/sites/digital-assets/2023/05/04/jack-dorsey-donates-10-million-to-fund-bitcoin-development/.

[3] Nostr.Band, "Nostr stats," 2024, https://stats.nostr.band/ Accessed: 2024-10-22.

[4] Damus, 2023, https://damus.io/.

[5] Forbes, "Meet @fiatjaf, the mysterious nostr creator who has lured 18 million users and $5 million from jack dorsey," 2023, https://www.forbes.com/sites/digital-assets/2023/05/30/bitcoin-social-network-nostr-creator-fiatjaf-/.

[6] Amethyst, "Amethyst: Nostr client for Android," 2023, https://github.com/vitorpamplona/amethyst.

[7] Iris, "Iris – The app for better social networks," 2023, https://github.com/irislib/iris-messenger.

[8] FreeFrom, "FreeFrom: Rewrire The Social Network!" 2023, https://freefrom.space/.

[9] Plebstr, "Plebstr: Nostr client. Reimagined." 2023, https://plebstr.com/.

[10] T. H. Bakry and T. Mysk, "Link previews: How a simple feature can have privacy and security risks," 2020, https://www.mysk.blog/2020/10/25/link-previews/.

[11] G. Stivala and G. Pellegrino, "Deceptive previews: A study of the link preview trustworthiness in social platforms," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/deceptive-previews-a-study-of-the-link-preview-trustworthiness-in-social-platforms/

[12] C. Garman, M. Green, G. Kaptchuk, I. Miers, and M. Rushanan, "Dancing on the lip of the volcano: Chosen ciphertext attacks on apple imessage," in *USENIX Security Symposium*. USENIX Association, 2016, pp. 655–672.

[13] T. Isobe and K. Minematsu, "Breaking message integrity of an end-to-end encryption scheme of LINE," in *ESORICS (2)*, ser. LNCS, vol. 11099. Springer, 2018, pp. 249–268.

[14] M. R. Albrecht, S. Celi, B. Dowling, and D. Jones, "Practically-exploitable cryptographic vulnerabilities in matrix," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022, pp. 1419–1436.

[15] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila, "A formal security analysis of the signal messaging protocol," in *EuroS&P*. IEEE, 2017, pp. 451–466.

[16] M. R. Albrecht, L. Mareková, K. G. Paterson, and I. Stepanovs, "Four attacks and a proof for telegram," in *IEEE Symposium on Security and Privacy*. IEEE, 2022, pp. 87–106.

[17] K. G. Paterson, M. Scarlata, and K. T. Truong, "Three lessons from threema: Analysis of a secure messenger," in *USENIX Security Symposium*. USENIX Association, 2023.

[18] T. Isobe, R. Ito, and K. Minematsu, "Security analysis of sframe," in *ESORICS (2)*, ser. LNCS, vol. 12973. Springer, 2021, pp. 127–146.

[19] T. Isobe and R. Ito, "Security analysis of end-to-end encryption for zoom meetings," *IEEE Access*, vol. 9, pp. 90 677–90 689, 2021.

[20] M. R. Albrecht, J. Blasco, R. B. Jensen, and L. Mareková, "Mesh messaging in large-scale protests: Breaking bridgefy," in *CT-RSA*, ser. LNCS, vol. 12704. Springer, 2021, pp. 375–398.

[21] M. R. Albrecht, R. Eikenberg, and K. G. Paterson, "Breaking bridgefy, again: Adopting libsignal is not enough," in *USENIX Security Symposium*. USENIX Association, 2022, pp. 269–286.

[22] T. von Arx and K. G. Paterson, "On the cryptographic fragility of the telegram ecosystem," in *AsiaCCS*. ACM, 2023, pp. 328–341.

[23] P. Rösler, C. Mainka, and J. Schwenk, "More is less: On the end-to-end security of group chats in signal, whatsapp, and threema," in *EuroS&P*. IEEE, 2018, pp. 415–429.

[24] S. Vaudenay, "Security flaws induced by CBC padding - applications to ssl, ipsec, WTLS ..." in *EUROCRYPT*, ser. LNCS, vol. 2332. Springer, 2002, pp. 534–546.

[25] K. G. Paterson and A. K. L. Yau, "Padding oracle attacks on the ISO CBC mode encryption standard," in *CT-RSA*, ser. Lecture Notes in Computer Science, vol. 2964. Springer, 2004, pp. 305–323.

[26] C. J. Mitchell, "Error oracle attacks on CBC mode: Is there a future for CBC mode encryption?" in *ISC*, ser. Lecture Notes in Computer Science, vol. 3650. Springer, 2005, pp. 244–258.

[27] M. R. Albrecht, J. P. Degabriele, T. B. Hansen, and K. G. Paterson, "A surfeit of SSH cipher suites," in *CCS*. ACM, 2016, pp. 1480–1491.

[28] M. R. Albrecht, K. G. Paterson, and G. J. Watson, "Plaintext recovery attacks against SSH," in *SP*. IEEE Computer Society, 2009, pp. 16–26.

[29] J. P. Degabriele and K. G. Paterson, "Attacking the ipsec standards in encryption-only configurations," in *S&P*. IEEE Computer Society, 2007, pp. 335–349.

[30] ——, "On the (in)security of ipsec in mac-then-encrypt configurations," in *CCS*. ACM, 2010, pp. 493–504.

[31] T. Jager and J. Somorovsky, "How to break XML encryption," in *CCS*. ACM, 2011, pp. 413–422.

[32] T. Duong and J. Rizzo, "Cryptography in the web: The case of cryptographic design flaws in ASP.NET," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2011, pp. 481–489.

[33] D. Poddebniak, C. Dresen, J. Müller, F. Ising, S. Schinzel, S. Friedberger, J. Somorovsky, and J. Schwenk, "Efail: Breaking S/MIME and openpgp email encryption using exfiltration channels," in *USENIX Security Symposium*. USENIX Association, 2018, pp. 549–566.

[34] J. Müller, F. Ising, V. Mladenov, C. Mainka, S. Schinzel, and J. Schwenk, "Practical decryption exfiltration: Breaking PDF encryption," in *CCS*. ACM, 2019, pp. 15–29.

[35] R. Fujita, T. Isobe, and K. Minematsu, "ACE in chains: How risky is CBC encryption of binary executable files?" in *ACNS (1)*, ser. LNCS, vol. 12146. Springer, 2020, pp. 187–207.

[36] Nostr, "Nips stand for nostr implementation possibilities," 2023, https://github.com/nostr-protocol/nips.

[37] Damus, "damus: A twitter-like nostr client for iPhone, iPad and MacOS," 2023, https://github.com/damus-io/damus.

[38] @paulmillr, "Draft: Nip44 encryption standard, rev.3," 2023, https://github.com/nostr-protocol/nips/pull/746/files.

[39] Nostr, "Nip-44 versioned encryption," 2023, https://github.com/nostr-protocol/nips/blob/95218740e2a78dbe941f09b09360541731d8f55e/44.md.

[40] M. Heiderich, S. Mazaheri, and D. Bleichenbacher, "Audit-report nip44 implementations 11.-12.2023," Cure53, 2023, https://cure53.de/audit-report_nip44-implementations.pdf.

[41] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "CONIKS: Bringing key transparency to end users," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 383–398. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/melara

[42] M. Sporny, D. Longley, M. Sabadello, D. Reed, O. Steele, and C. Allen, "Decentralized identifiers (dids) v1. 0," *W3C Recommendation*, 2022.

[43] M. Alatawi and N. Saxena, "Sok: An analysis of end-to-end encryption and authentication ceremonies in secure messaging systems," in *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 187–201. [Online]. Available: https://doi.org/10.1145/3558482.3581773

[44] A. Herzberg and H. Leibowitz, "Can johnny finally encrypt? evaluating e2e-encryption in popular im applications," in *Proceedings of the 6th Workshop on Socio-Technical Aspects in Security and Trust*, ser. STAST '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 17–28. [Online]. Available: https://doi.org/10.1145/3046055.3046059

[45] S. Schröder, M. Huber, D. Wind, and C. Rottermanner, "When signal hits the fan: On the usability and security of state-of-the-art secure mobile messaging," in *European Workshop on Usable Security. IEEE*, 2016, pp. 1–7.

[46] E. Vaziripour, J. Wu, M. O'Neill, J. Whitehead, S. Heidbrink, K. Seamons, and D. Zappala, "Is that you, alice? a usability study of the authentication ceremony of secure messaging applications," in *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, 2017, pp. 29–47.

[47] K. L. Sean Lawlor, "Deploying key transparency at whatsapp," 2023, https://engineering.fb.com/2023/04/13/security/whatsapp-key-transparency/.

[48] A. S. Engineering and Architecture, "Advancing imessage security: imessage contact key verification," 2023, https://security.apple.com/blog/imessage-contact-key-verification/.

[49] T. K. Yadav, D. Gosain, A. Herzberg, D. Zappala, and K. Seamons, "Automatic detection of fake key attacks in secure messaging," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 3019–3032. [Online]. Available: https://doi.org/10.1145/3548606.3560588

[50] Y. Wei and G. Tyson, "Exploring the nostr ecosystem: A study of decentralization and resilience," 2024.

[51] V. Buterin, Y. Weiss, D. Tirosh, S. Nacson, A. Forshtat, K. Gazso, and T. Hess, "Erc-4337: Account abstraction using alt mempool," 2021, https://eips.ethereum.org/EIPS/eip-4337.

[52] Huntr, 2023, https://huntr.dev/.

[53] MITRE, "CVE Numbering Authorities (CNAs)." 2023, https://www.cve.org/ProgramOrganization/CNAs.

[54] Nostr, "nostr - notes and other stuff transmitted by relays," 2023, https://github.com/nostr-protocol/nostr.

[55] T. Perrin and M. Marlinspike, "The double ratchet algorithm," 2016, https://www.signal.org/docs/specifications/doubleratchet/.

[56] T. Berners-Lee, R. T. Fielding, and L. M. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," RFC 3986, Jan. 2005. [Online]. Available: https://www.rfc-editor.org/info/rfc3986

Appendix A. Research Ethics & Open Science Policy

A.1. Ethical Considerations

All vulnerabilities described in this paper are due to flaws in the Nostr specification and its implementations. Also, each of our attacks exploits some of the vulnerabilities, which are flaws in the Nostr specifications, the client implementations (e.g., Damus, Iris, FreeFrom, Plebstr), or their combinations. This implies that, to avoid all of our attacks, both the specification and the affected client implementations must be fixed. It should be noteworthy here that we never reverse-engineered any of the server and client implementations; instead, we performed black-box tests for non-OSS software.

First of all, we analyzed Damus and made responsible disclosure to its developers in the following timeline: for the contents of Section 4.2 on June 30, 2023; for the contents of Section 6 on July 1, 2023; and for the contents of Appendix E.4 on July 15, 2023. We reported our findings privately through emails and the vulnerability reporting platform Huntr [52], which is authorized as a CNAs (CVE Numbering Authorities) [53].

We created patches for the above three attacks and reported them to Damus's developers. Among these patches, the one for Section 6 was merged into the Damus application in July 2023. An update of Damus, version 1.6.0, became available on GitHub and Apple TestFlight after these actions. However, subsequently, the developers removed our patch from Damus without notifying us. Therefore, the attacks remain applicable in version 1.6 (29), released on November 16, 2023.

For other responsible disclosures, we have completed the process of notifying developers of our findings on Sections 3, 4, and 6 and Appendix E under the typical 90-day disclosure period. As for the timeline, in December 2023, we contacted a third-party organization for coordination of vulnerability disclosure. Then, through this third-party organization or Huntr, we sent each product's developer an independent vulnerability report, together with a PoC trial video and PoC codes. The dates on which the third-party organization or Huntr successfully contacted the developers and provided them with detailed vulnerability information are listed below: for Damus v1.5(8) on June 30, 2023; for Plebstr on January 23, 2024; for FreeFrom on January 24, 2024; for Damus v1.6(29) on January 24, 2024; and for Iris(iOS) on February 2, 2024.

The developers of FreeFrom and Plebstr acknowledge all the vulnerabilities we reported. The patched version of FreeFrom, v1.3.6, was released in February 2024. However, as of October 24, 2024, we still have not received a clear response from the developers of client applications other than FreeFrom and Plebstr. We will continue to monitor the status of these client applications for any updates or fixes.

TABLE 2. Downloaded numbers for each Nostr client application (confirmation on October 24, 2024 (UTC+0)).

App	Downloads
Damus (iOS)	160,000+††
Amethyst (Android)	100,000+
Openvibe(previously Plebstr) (Android)	10,000+
Iris (Android)†	10,000+
FreeFrom (Android)	10,000+

† We did not conduct any PoC tests on Iris (Android) as it could not launch in our environment. However, to understand the popularity of Iris, we present the download numbers for the Android version.
†† The statistics for Damus are sourced from this article [5] (as of May 2023), as download statistics for the iOS app are not publicly available.

A.2. Open Science Policy

Detailed instructions for replicating the PoC tests are provided in Appendix G.1. This includes the environment setup, dependencies, and steps to re-execute the tests on local machines. The versioned dependencies for the tools used, including the Nostr clients, relay server implementations are explicitly documented to facilitate reproducibility.

Appendix B. Statistics of Nostr

Table 2 lists the download numbers to gauge the popularity of Nostr client applications. The number of downloaded Android applications can be verified from

Google Play[10,11,12]. However, the number of downloaded iOS applications and the usage of web applications have not been disclosed. Exceptionally, for Damus, an estimated user count of 160,000 as of May 30, 2023, has been revealed through interviews and research conducted by Forbes [5].

Appendix C. Bitcoin Transfer for Damus

In Damus, the Bitcoin transfer via profile is executed in the following procedure:

1) A sender taps the lightning bolt button on a receiver's profile screen. Then, the sender's device screen moves to the screen for transferring Bitcoin.
2) The sender specifies the amount of Bitcoin and taps the "Zap User" button.
3) The sender's device screen moves to the wallet app, and the sender taps the "Send" button.

After completing these steps, the specified amount of Bitcoin is transferred to the receiver's wallet. The detail of the receiver's wallet is written by the lud06 or lud16 element in the profile Event data.

Appendix D. Further Discussion of Our Attacks

D.1. Lack of Signature Verification on Client Side

As described in Table 1, some client applications, such as Damus, Iris, FreeFrom, and Plebstar, are affected by the attacks due to Vulnerability 2. Each of these client applications employs its own implementation, but the signature verification is not executed for almost all the received Events in them. Then, we investigate a main factor that prevents the signature verification from being executed to get a new insight from such a factor.

For example, we focus on Damus here. Our investigation revealed that the Damus developers decided not to implement the signature verification to mitigate its adverse impact on the iPhone's battery life[13]. Instead, the default relay server executes the signature verification.

A trusted server seems rational and works well in centralized SNSs like 𝕏, but it is not necessarily a relevant case in the Nostr-based distributed SNSs. This is because no NIP document mentions executing the signature verification on the server side. In addition, the Nostr GitHub page [54] explicitly states that "Anyone can run a relay [...] Relays don't have to be trusted. Signatures are verified on the client side", emphasizing the protocol's design so that all users do not have to rely on relay servers. This requires all users to verify signatures to detect dishonest behavior by a malicious server.

D.2. Potential Issues of Vulnerability 4

As described in Section 4.2, we assume that the adversary exploits Vulnerability 4 to try to acquire a plaintext/ciphertext (P/C) pair encrypted using the shared secret between Alice and Bob. Considering such a situation as an example of the initialization sequence in the Nostr connect, potential issues of Vulnerability 4 can be listed as follows:

- The use of the same public key pair, the same session establishment method, and the same encryption mechanism are deployed in both protocols described in the NIP-04 and NIP-46 documents. This issue enables the adversary to reuse the acquired P/C pair in encrypted DMs between the target users.
- In the initialization sequence of the Nostr connect, the known plaintext specified in the NIP-46 document is encrypted. This issue potentially enables the adversary to acquire the known P/C pair.
- The authenticity of $\mathrm{PubKey}_{\mathrm{App}}$ is not ensured during the initialization sequence of the Nostr connect. This issue enables the adversary to embed an arbitrary public key into a QR code. In other words, as in the above scenario, the adversary can establish an illegitimate session for the encrypted communication between Alice and Bob.
- The adversary can embed a URL for his/her own server into a QR code. Then, the adversary can trick a victim who scans such a QR code into sending a connect request message to the adversary. This is a specification issue in the NIP-46 document. Even if a server is not registered for the Nostr account, it is approved on the system to connect to the URL for the server embedded into the QR code. This issue enables the adversary to easily acquire the known P/C pair without intercepting such a pair on the communication between Alice's signer and the connected WebApp.

D.3. Practicality of 1-block Message Forgery

As described in Section 4.2, we provided the advanced forgery attack on encrypted DMs, in which a malicious user can manipulate a 1-block short message to achieve any desired decryption result. Then, we demonstrate here that forging even a single message block (16 characters) can lead to a practical attack on DMs in a chat manner.

We believe that forging a 1-block message is highly practical because, in a chat, a long message tends to be divided into multiple short messages before being sent. We have confirmed that this expectation is correct by analyzing the size of encrypted DMs in the Nostr protocol. To demonstrate this, we collected $10,000$ of the most recent messages each from three popular relay servers, which were used for Damus on December 5, 2023 (UTC+0) by defaults[14], where DM metrics were available. After removing duplicated or invalid messages, we used a total of $25,577$ encrypted DMs for our analysis.

Figure 7 illustrates a distribution in the number of posted DMs for each ciphertext size. In this figure, the vertical axis represents the number of posted DMs, and the horizontal axis represents the size of the ciphertext in 16-byte units. It can be seen from this figure that 1-block

10. Amethyst - Google Play
11. Plebstr – Nostr client – Google Play
12. Iris –- The Nostr client – Google Play
13. Snort post on February 14, 2023 (UTC+0).
14. wss://relay.damus.io, wss://yabu.me, wss://nos.lol

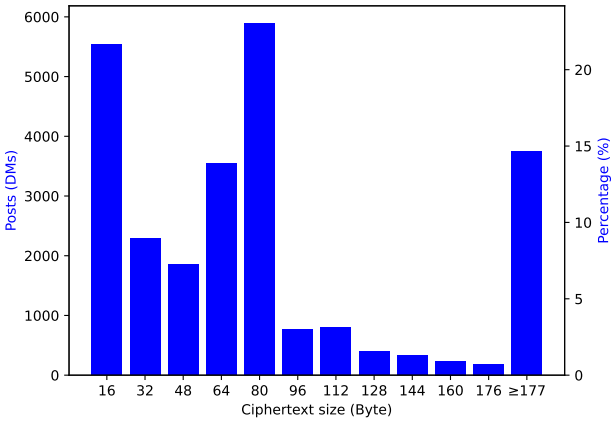

Figure 7. Distribution in the number of posted DMs for each ciphertext size. The total number of DMs we aggregated is 25,577.

(16-byte) messages were sent in 21.6% of all cases. This is the second most frequently sent message size, only 1.4% less than the most common 5-block (80-byte) messages, i.e., 23.0%.

To sum up, based on our considerations on the use case of chat messages, we conclude that our forgery attack on encrypted DMs (Section 4.2) has significant practicality.

D.4. Lack of Padding Check on Damus

In general, the well-known PKCS#7 padding mechanism is used for the AES-CBC encryption, but the NIP-04 makes no mention of the padding mechanism. This causes a slight difference in the implementation among Nostr client applications. Specifically, we have confirmed that Damus does not employ the padding mechanism, while other client applications do it. This difference slightly affects the feasibility of the forgery attack on encrypted DMs (Section 4.2).

A malicious Damus user does not need to consider the padding mechanism when trying to forge the encrypted DMs. Assuming that the ASCII character codes are used for the encrypted DMs, the adversary can forge a message up to 16 characters long since the block size of AES is 16 bytes. In contrast, the other client applications follow the PKCS#7 padding mechanism. This implies that when a malicious user tries to send a 16-byte message to the target user, in fact, the size of the padded message becomes 32 bytes. Given that our attack is valid for a 1-byte padded message, the adversary should not send a message with a size of 16 bytes or more. Therefore, a malicious user in the other client applications can forge a message up to 15 characters long.

D.5. On NIP-44

A new specification for encrypted messaging, NIP-44, [38], [39] was published on December 20, 2023, accompanied by an audit report by Cure53. As the successor to NIP-04, NIP-44 replaces the conventional CBC mode with an authenticated encryption scheme, ChaCha20 combined with HMAC-SHA256, to guarantee ciphertext integrity and mitigate the malleability issues inherent to AES-CBC encryption (as exploited in Sections 4.2 and 5).

While NIP-44 incorporates several improvements over NIP-04, it does not fully address certain underlying protocol design concerns. In particular, design vulnerabilities such as twist attacks on secp256k1, the lack of forward secrecy, and the reuse of signing keys for encryption are inherent issues that were likely present in NIP-04 as well. It should be noted that the Cure53 audit focuses solely on NIP-44 and does not provide an evaluation of NIP-04.

We do not know if our private contact with the developer affected NIP-44. Nonetheless, we assume that the transition to NIP-44 has adequately remedied the core design issues we reported, such as CBC malleability and the separation of session keys.

However, due to backward compatibility requirements, the migration from NIP-04 to NIP-44 is expected to be gradual. As of July 10, 2024, major Nostr clients still predominantly support NIP-04, and our targets, except Amethyst, have not yet adopted NIP-44. Therefore, the vulnerabilities addressed in this work still have significant practical implications. Furthermore, even if NIP-44 eventually replaces NIP-04, our attacks on components other than encrypted direct messages, as described in Sections 3, 4.1, 6, and Appendix E.4, will remain applicable.

Future work may extend our analysis to other emerging NIP specifications, thereby broadening the overall understanding of their security impacts on the Nostr ecosystem. For example, in the context of private direct messaging, the encryption specification is provided by NIP-44, while NIP-17 and NIP-59 define the message format. We have not analyzed the message format in this work. In the event that future applications extend the use of NIP-44 encryption beyond private DMs, ensuring careful design and conducting rigorous analysis of their message formats will be imperative.

Appendix E. Other Attacks

We briefly summarize five additional attacks caused by some vulnerabilities presented in Sections 3–6. Moreover, we mention a new issue from the Damus implementation aspect.

E.1. Impersonation Attack Using Similar User ID

E.1.1. Overview. In the Nostr protocol, any user cannot use an arbitrary string as a user ID. However, users can embed an arbitrary partial string into their user ID by using publicly available tools like nostrogen[15] and rana[16]. In fact, rana successfully found a proper private key that corresponds to the public key whose first five characters match the specified string. This can be done within 8 minutes on our MacBook Pro with i7-8569U CPU and 16GB RAM. This fact suggests that the adversary may be able to intentionally create a similar account of the target user by generating a similar user ID of the target user; thus, there exists a potential risk that the adversary can impersonate the target user.

15. https://github.com/tonyinit/nostrogen
16. https://github.com/grunch/rana

As an application of the impersonation attack using a similar user ID, a malicious user may be able to execute a man-in-the-middle (MitM) attack on encrypted DMs between two users (e.g., Alice and Bob). If Alice accidentally sends an Event of an encrypted DM to a similar account to Bob's (i.e., the adversary's account), the adversary can forge the received Event, impersonate Alice, and send the forged Event to Bob. Due to Vulnerability 1, Bob successfully verifies the signature in the forged Event; thus, he properly accepts the forged Event.

This attack is a direct application of the forgery attack explained in Section 3. The only difference between these attacks is whether the adversary creates a similar user ID or not. This implies that if the adversary can easily create a similar user ID of the target user, the PoC tests for both attacks can be considered to yield the same result. In addition, the issue of creating a similar user ID is solved with a generic social engineering technique, and in fact, it is known to be relatively easy to obtain. Therefore, we do not list the PoC result of the impersonation attack in Table 1.

E.1.2. Mitigation. This attack is based on Vulnerability 1. Therefore, this can be made infeasible by taking the countermeasures suggested in Section 7.1 and Appendix F.1.

E.2. Generic Replay Attack

E.2.1. Overview. By exploiting Vulnerability 2, a malicious user can execute a generic replay attack on encrypted DMs between two users (e.g., Alice and Bob) in some client applications, such as Damus [37] and Iris [7]. Specifically, the adversary runs the following steps:

1) the adversary intercepts an Event of an encrypted DM sent from Alice to Bob.
2) The adversary forges the create_at and id fields of the Event in such a way that a new timestamp is embedded into the create_at field, and the recalculated id value is embedded into the id field.
3) The adversary impersonates Alice and sends the forged Event of the encrypted DM to Bob.

Normally, the signature verification for the forged Event fails with a very high probability, but Bob properly accepts the forged Event due to Vulnerability 2. Therefore, the generic replay attack is feasible by a malicious user in some client applications affected by Vulnerability 2.

E.2.2. Mitigation. This attack is based on Vulnerability 2. Therefore, this can be made infeasible by taking the countermeasure suggested in Section 7.2.

E.3. Truncated Replay Attack

E.3.1. Overview. In the generic replay attack, a malicious user uses the same content in the intercepted Event of an encrypted DM. We briefly demonstrate here that a different type of replay attack is feasible even if a part of the content is truncated. This type of attack is referred to as a truncated replay attack.

In the CBC mode, the i-th plaintext block, P_i, depends only on the $(i-1)$-th and the i-th ciphertext blocks, C_{i-1} and C_i. This property can be exploited in the truncated replay attack. For example, a malicious Damus user runs the following procedure:

1) The adversary truncates up to the $(i-1)$-th ciphertext block and embeds the i-th ciphertext block onwards (i.e., $C_i, \ldots, C_n$) into the content field of the target Event. This means that P_i of the original Event is the first plaintext block of the forged Event.
2) The adversary replaces the IV value with C_{i-1}.

In the forged Event, the plaintext blocks are properly decrypted from the ciphertext blocks and the forged IV. Therefore, by incorporating the above procedure into the generic replay attack, the truncated replay attack is feasible by a malicious user in some client applications affected by Vulnerability 2.

E.3.2. Mitigation. For the same reason as the generic replay attack, this can be made infeasible by taking the countermeasure suggested in Section 7.2.

E.4. Breaking Robustness of Damus

E.4.1. Overview. Receiving the target Event from multiple relay servers allows all users to ensure censorship resistance through the distributed robustness property that the Nostr protocol aims to achieve. However, a malicious server on the Damus network can disrupt the robustness by exploiting Vulnerability 7. Specifically, once the robustness is broken, a victim, Bob, will not be able to get the target user's (Alice's) profile until Alice updates her profile information.

A malicious server attempts to break the robustness of Damus as follows:

1) The adversary gets the current Alice's profile Event from the other legitimate relay server. This can be normally done with a generic user authority.
2) The adversary forges the sig value of the profile Event into an invalid sig value. Now, the id value of the forged Event is the same as that of the original one.
3) When Bob requests Alice's profile Event, the adversary must send the forged Event earlier than any other legitimate relay servers on the Damus network do the proper Event. If not, the adversary always fails to break the robustness.
4) After Bob receives the forged Event, he always fails to verify the sig value in the Event. Then, he stores the id value and the corresponding verification result (i.e., 'fail') in the cache on his device.

After completing the above scenario, all the legitimate relay servers possess the proper Alice's profile Event, but Bob cannot get the current Alice's profile until Alice updates her profile information. This is because the id value of the Event is not changed until the Event data is updated; thus, the received Event is always rejected due to the cache search.

E.4.2. Mitigation. The main factor in this issue is that failed signature verification results are stored in the cache list on the user's device. To avoid breaking the robustness property, we recommend not storing the failed results in the cache list if client applications employ a cache implementation like Damus.

E.5. Bypassing Block List of Damus

E.5.1. Overview. We discovered an attack that allows a malicious user to bypass the block list implementation for encrypted DM. The block list is a well-known feature for some SNSs, but its specification is not part of the NIP standard specifications. Damus uniquely implements its feature; then, we analyzed its details.

Damus manages its block list by listing the hexadecimal string of a user's pubkey value. If the pubkey string in the received DM is listed in the block list on the user's device, Damus does not display the content of its DM.

We found an issue in the verification process of the pubkey string listed in the block list is case-sensitive. Normally, the alphabetic characters in the pubkey string are written in lowercase. Then, this issue enables a malicious user to bypass the target user's block list by converting a part or all of the pubkey string to uppercase.

E.5.2. Mitigation. The simplest way to avoid bypassing the block list is to detect during the verification process that the hexadecimal uppercase and lowercase pubkey strings represent the same value.

Appendix F. Recommended Mitigation

We summarize three countermeasures to fix the specification issues in the Nostr protocol. As far as we can see the NIPs, there are almost no security claims from the cryptographic perspective, such as the importance of signature verification. For this reason, we recommend that the Nostr developers prepare the NIP describing some security considerations, including functionalities that must be implemented in the client applications.

F.1. Authenticity Check for Public Keys on distributed SNS

We discuss in Section 7.1 how to compose key authenticity using OOB and key transparency. In this section, we propose a novel technique called search query as another scheme for mitigating Vulnerability 1. This technique is based on the fact that in a distributed SNS there are multiple relays with different owners. This defines the specification for consistency checks and deduplication when receiving Event data from multiple servers. The core idea of the search query is to use one or more trusted primary servers to verify Event data received from untrusted servers. This allows users to avoid negative impacts even when connected to malicious servers as long as trustworthy servers are available within Nostr (e.g., user-selected servers or servers operated by the same organization that provides the client application). A concrete example of the search query is as follows.

1) Alice requests the sender's pubkey from multiple connected servers as background queries.
2) she checks the consistency of all the pubkeys obtained by the queries and the pubkey embedded in the Event data received from the primary connected server.
3) If there is an inconsistency, she discards the received Event.

The performance impact of the above procedure would be small because our mitigation does not increase the number of queries for Event retrieval compared to the current specifications of Nostr. For instance, under the current Nostr design, when a user connects to N relays, Events are redundantly retrieved from all connected servers. Our proposal clarifies this behavior, ensuring that a user queries no more than N trusted servers as background queries, thus not increasing the number of communication queries compared to the existing specifications. Moreover, it is practically hard for the adversary to return all the same pubkey values to Alice, as it needs to take control of all the connected servers to Alice. Although we do not claim the search query as a perfect solution, it is expected to thwart the forgery attack explained in Section 3.

F.2. Switching to Other Encryption Scheme

The forgery attack described in Section 4.2 is based on the existing attack on the CBC mode [35]. This attack can be made infeasible by simply switching to the other encryption scheme, but the Nostr developer will have to modify the NIP-04 document since Vulnerability 3 is caused by the specification issue.

While writing this manuscript, we found out that the Nostr developers are considering publishing a new specification for encrypted DMs as a draft version of NIP-44 document [38], and they published it on the end of December 2023 [39]. They employed a generic composition of ChaCha20 and HMAC-SHA256 as an authenticated encryption scheme and HKDF as a key derivation function. It is expected that the security of the Nostr client will be improved by deploying these schemes.

However, to effectively address the risk of downgrade attacks, it is essential to implement mechanisms that prevent fallback to weaker encryption schemes. As of October 24, 2024, our target clients continue to use NIP-04, which lacks the improved security properties of NIP-44. Therefore, validating the negotiated encryption scheme and rejecting connections that attempt to use deprecated or less secure methods remains a critical step in securing the Nostr protocol.

F.3. Key Separation

Vulnerability 4 can be avoided by separating public key pairs used in each protocol, such as encrypted DMs, the Nostr connect, the Nostr marketplace, etc. Specifically, a user generates a private key for each protocol from the user's master key using a key derivation function like HKDF. Then, by calculating the corresponding public key from the generated private key, the user can possess public key pairs for each protocol. However, as far as the AES-CBC encryption is employed for encrypted DMs, it should be noted that this countermeasure cannot avoid the forgery attacks by exploiting the guessing approach, as described in Section 4.2.

Additionally, it is worth considering modern cryptographic solutions like the Double Ratchet algorithm, which is implemented in messaging applications such as Signal, WhatsApp, and iMessage PQ3 [55]. The Double Ratchet algorithm not only facilitates key separation but also

enables forward secrecy, offering a more robust solution for securing encrypted communications.

Appendix G. More Detailed Explanation of Our PoC Tests

G.1. PoC Environment

We tested our Proof of Concept (PoC) tests with two versions of Damus, Amethyst, Iris, FreeFrom, and Plebstr. The detailed version of the target client applications is shown in Table 1. These client applications were the latest versions as of November 30, 2023, the date we tested them. All our PoC tests were executed in a local network environment.

For simulating the Nostr clients, we used two iOS devices with version 17.1 to test our PoCs for the target iOS applications, which were conducted on iPhone 12 mini physical phone and iPhone 15 simulator. Also, we used an Android device to test our PoCs for the target Android applications, which were conducted on Android 11 with its emulator.

For simulating the Nostr relay server, we used Nostream[17] version 1.22.6 as a relay server application and installed it into MacBook Pro 2019 13-inch with macOS version 13.6.1.

For building the Nostr connect (NIP-46) environments, we used Nostrum[18] (version 592e125) as a signer application and Nostr Connect SDK[19] (version 6ae464b), which is a library that allows you to easily integrate the Nostr Connect into a web application.

All communications between the client and the server were protected using TLS 1.2. We set up a TLS environment in the local network using a self-signed certificate, which were installed on the testing devices. It is important to note the adversaries would not know the root key in real-world settings, so they cannot break the TLS in our PoC tests.

G.2. Conditions for Successful Key Substitution Attack (Section 3)

In the theoretical aspect, our key substitution attack (Section 3) is feasible for all Events. However, almost all the NIPs are provided as optional features except for the NIP-01 specification; thus, not all options are necessarily configured into the target client applications. Then, we limited targets to a profile and contact lists, which are the fundamental SNS features, and implemented the key substitution attack on them under the malicious server assumption.

Our PoC tests have clarified that in Damus v1.6(29) and Plebstr (iOS/Android), the key substitution attack on a profile is invalid, while the attack on contact lists is valid. Also, in other target client applications, Amethyst, Iris, and FreeFrom, the attacks on both targets are invalid. Amethyst and Iris are resistant to such attacks, as described in Section 8. It is unclear why FreeFrom was able to prevent the attacks due to its non-OSS application, but we guess that it is resistant to the attack owing to the same reason as the cases of Amethyst.

Looking into the details of our PoC tests for the attack, especially on a profile, we have found a condition for making our attack feasible in Damus v1.5(8) and Plebstr (iOS/Android). Specifically, the condition depends on the order of screen transitions for displaying the victim's profile. For example, they are roughly divided into the following two patterns:

1) A user directly displays the victim's profile from the screen of contact lists, timelines, or DMs.
2) Assuming that a user knows the victim's pubkey value. Then, the user uses it and searches the victim's account from the search screen to display the victim's profile.

Here, the timeline means a screen in which the contents posted by users are listed. By carefully observing our PoC tests, we have confirmed that forging the victim's profile succeeds only in the above second pattern. The reason why the forgery fails in the first pattern is that the attack does not involve replacing the victim's public key in the previous screen (i.e., an Event of contact lists, timelines, or DMs), which still contains the correct public key as a reference for the profile. As a result, the client can identify the correct public key, preventing the attack from succeeding.

Since our PoC tests aim to verify the pure key substitution attack on a profile, we did not assume forging the previous screen. However, if we test PoC for the attack by taking such a forgery into account, the feasibility of the attack is expected to increase (but it is out of scope in our PoC tests).

G.3. Success Probability for Advanced Forgery Attack on Encrypted DMs (Section 4.2)

We conducted a PoC test to verify the advanced forgery attack on encrypted DMs described in Section 4.2, exploiting Vulnerability 4 to obtain a known plaintext/ciphertext pair. Additionally, our attack can be prevented by implementing the mitigation measures proposed in Section 7.

We discuss here the success probability of our attack. As described in Section 4.2, once the adversary acquires a 1-block P/C pair, the adversary can easily execute our attack (i.e., with a probability of 100%). In other words, it can be considered that the success probability of our attack depends on the probability that the adversary acquires a 1-block P/C pair. After looking deeper into the details of acquiring a 1-block P/C pair, we found that its success probability is affected by the string length of the id value in the connect request message sent from the signer on the victim's device. As described in Section 2.1.3, the string length of the id value is not limited in the protocol for the Nostr connect, and the string of the id value is generated by the `Math.random().toString().slice(2)` function of the typescript language. In addition, the string of the id value is always set sequentially, starting from the first plaintext block for the AES-CBC encryption. This implies that varying the string length of the id value is to vary the bit position of the first string of the known plaintext. Given that the id value is generated on the victim's device, the

17. nostream
18. Nostrum
19. Nostr Connect SDK

TABLE 3. Distribution of the Length in Values Obtained from `Math.random()` in React Native (ver.0.70.5) on iOS 16.7

Length	Probability	Count (Try: 100000)
11	0.00001	1
12	0.00004	4
13	0.00064	64
14	0.00712	712
15	0.06661	6661
16	0.61408	61408
17	0.27791	27791
18	0.03018	3018
19	0.00305	305
20	0.00031	31
21	0.00004	4
22	0.00001	1

adversary cannot take a method of acquiring a known P/C pair while dynamically adapting the change of the string length of the id value. Therefore, the adversary needs to focus on the most frequently generated string length of the id value to improve the success probability of the attack.

We executed an additional experiment to check the most frequently generated string length of the id value. In this experiment, we ran the `Math.random().toString().slice(2)` function 100,000 times and obtained the distribution of the string length of the id value. As a result, it can be seen from this experiment that the probability that the string length of the id value is 16 characters is around 61% (see Table 3). To summarize, the adversary can execute the forgery attack on encrypted DMs with a probability of around 61% by limiting the string length of the id value to 16 characters.

G.4. Link Preview and URL Parser for Each Client (Section 5)

All attacks in Section 5 were successful against Damus. On the other hand, Plebstr's URL parser did not treat query parameters as URLs. Consequently, URL recovery attacks and their variants are valid against Plebstr, but the generic message recovery attack is invalid. In addition, FreeFrom does not implement link preview for webpages, but it does for images. Therefore, it is vulnerable to a variant of the URL recovery attack discussed in Section 5.2.2.

Furthermore, we demonstrated that the concerns discussed in Section 5.4.1 actually occur in Damus. The root cause of this issue is that Damus separates the boundaries between URLs and regular text with spaces when a URL appears in the decrypted string. In other words, any characters following a URL are treated as part of the URL until a space appears. Additionally, if the characters included in "that URL" are multi-byte characters that conform to UTF-8 rather than ASCII, the URL parser on Damus encodes the message in a format compliant with RFC 3986's Percent-Encoding [56]. For example, the Emoji `grinning face` (Unicode: `U+1F600`) is converted to `%F0%9F%98%8A`. Moreover, messages written solely in non-segmented languages (e.g., Chinese, Japanese, and Thai) are expected to rarely contain spaces, and more messages are converted into URL-valid query parameters and leaked to the adversary's server.

G.5. Permanent Cache Violation on Damus v1.6(29)

We investigated the conditions under which the cache of Damus v1.5(8) and v1.6(29) persist to understand the impact of Vulnerability 7 on the actual environment.

Our findings indicated that the cache implementation in Damus v1.5(8) was a simple key-value store implemented in memory. Thanks to this implementation, victims could simply restore Damus to a normal state by restarting the application even if victims are under the influence of the attack as described in Section 6.

In contrast, the cache implementation in Damus v1.6(29) is as a local persistent database. It was found that not only the signature verification results but also the profile contents are persistently cached. This leads to two major issues.

G.5.1. Issue 1: victims will be exposed to attacks for a long time. Our investigation showed that the cache is only deleted upon uninstallation. This means that once a victim is attacked, restarting the application may not resolve them susceptible to attacks, including the forgery attack mentioned in Section 6. That is, Alice remains under attack until she receives a new profile from the legitimate Bob with the correct signature. However, the timing of the profile update depends on Bob, and Alice has no control over it. The only viable solution for Alice is to reinstall Damus for this case.

G.5.2. Issue 2: do not sign in to a device that has received suspicious operations. Our investigation also found that the cache is not deleted even when a user signs out of Damus, and it continues to be referenced when signing in as another user. By exploiting this issue, an adversary with offline access to the victim's device (or a shared device where the victim is likely to sign into Damus) can execute the forgery attack mentioned in Section 6, even if the victim only accesses non-malicious relay servers.

To execute this attack, the adversary initially signs into Damus on Alice's device (or a shared device like an iPad) with their own account and performs the attack procedure described in Section 6 on Bob's profile. This results in the forged Bob's profile being saved in Damus's cache. Afterward, the adversary signs out of Damus. When Alice accesses Bob's profile on Damus from her device (or a shared device) that has undergone such malicious manipulation, she may see the forged Bob's profile despite only accessing non-malicious relay servers. The viable solution in this case is also to reinstall Damus if she finds any evidence of suspicious activity on her device (or a shared device).