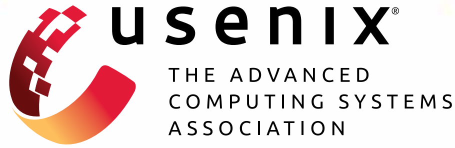


Boomerang: Metadata-Private Messaging under Hardware Trust

Peipei Jiang, *Wuhan University and City University of Hong Kong;* Qian Wang and Jianhao Cheng, *Wuhan University;* Cong Wang, *City University of Hong Kong;* Lei Xu, *Nanjing University of Science and Technology;* Xinyu Wang, *Tencent Inc.;* Yihao Wu and Xiaoyuan Li, *Wuhan University;* Kui Ren, *Zhejiang University*

https://www.usenix.org/conference/nsdi23/presentation/jiang

This paper is included in the Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation.

April 17–19, 2023 • Boston, MA, USA

978-1-939133-33-5

Open access to the Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation is sponsored by

Boomerang: Metadata-Private Messaging under Hardware Trust

Peipei Jiang[1,2] Qian Wang[1,*] Jianhao Cheng[1] Cong Wang[2] Lei Xu[3]
Xinyu Wang[4] Yihao Wu[1] Xiaoyuan Li[1] Kui Ren[5]
[1] *School of Cyber Science and Engineering, Wuhan University* [2] *City University of Hong Kong*
[3] *Nanjing University of Science and Technology* [4] *Tencent Inc.* [5] *Zhejiang University*

Abstract

In end-to-end encrypted (E2EE) messaging systems, protecting communication metadata, such as who is communicating with whom, at what time, etc., remains a challenging problem. Existing designs mostly fall into the balancing act among security, performance, and trust assumptions: 1) designs with cryptographic security often use hefty operations, incurring performance roadblocks and expensive operational costs for large-scale deployment; 2) more performant systems often follow a weaker security guarantee, like differential privacy, and generally demand more trust from the involved servers. So far, there has been no dominant solution. In this paper, we take a different technical route from prior art, and propose Boomerang, an alternative metadata-private messaging system leveraging the readily available trust assumption on secure enclaves (as those emerging in the cloud). Through a number of carefully tailored oblivious techniques on message shuffling, workload distribution, and proactive patching of the communication pattern, Boomerang brings together low latency, horizontal scalability, and cryptographic security, without prohibitive extra cost. With 32 machines, Boomerang achieves 99th percentile latency of 7.76 seconds for 2^{20} clients. We hope Boomerang offers attractive alternative options to the current landscape of metadata-private messaging designs.

1 Introduction

In E2EE messaging systems [59], the exposure of privacy-revealing communication metadata, including the identities of the communicating parties, and the timing and volume of the traffic, remains a big concern [26, 39, 61]. Communication metadata can not only be used to target whistleblowers and journalists [37, 70], but also serve as a surveillance means to reveal intimate details of a person's life [38]. Designing a metadata-private messaging system is a challenging problem, given the powerful attackers that can monitor and actively interfere with the network traffic [31, 39]. The popular system in practice today that hides communication metadata, namely Tor [34], is not resilient against even passive traffic analysis attacks (e.g., through timing, volume patterns, etc.) [39]. Because of this, academic research systems have been developed recently with well-defined security guarantees for improved metadata privacy [3–6, 8, 11, 15, 16, 27–29, 36, 37, 52, 54, 56–58, 70, 72, 88, 91]. Ideally, such a system must ensure that any pair of connected clients might be communicating from the view of powerful attackers. This implies two necessary requirements: 1) unlinking senders and receivers during any message exchange; and 2) maintaining the same communication pattern across all connected clients, to resist traffic analysis. Roughly speaking, most prior art on metadata-private messaging falls into the balancing act among security, performance, and trust assumptions, while trying to meet these two requirements (more discussions in §7).

Among the diverse landscape of metadata-private messaging designs, there are two commonalities of state-of-the-art systems: 1) leveraging an intermediate "virtual address" to facilitate obfuscated message "drop" and "fetch" between the communicating pairs; and 2) operating in rounds. Designs with cryptographic security follow technical routes of sophisticated cryptographic operations that either obliviously "write to (drop)" [5, 27, 37, 52, 54, 70] or obliviously "read from (fetch)" the virtual address [4, 8]. The hefty operations, however, often present performance roadblocks and unfavorable operational dollar costs, hindering large-scale voluntary adoptions in practice. More performant systems [11, 56, 87, 88] choose to relax the security guarantee and use random noise to disguise the observable "drop" / "fetch" at the virtual address in the framework of differential privacy. Generally, these systems need to trust a fraction of servers for the claimed security, where the latency would be increased if trusting fewer servers. So far there has yet to be any dominant solution.

Motivations. In this paper we propose Boomerang, an alternative metadata-private messaging system leveraging secure enclaves. Boomerang takes a different technical route from prior art, and is partially motivated by the readily available

*Qian Wang is the corresponding author.

trust assumption on hardware enclaves. As emerging in the cloud [78, 81], secure enclaves provide a convenient and native implementation choice to build secure yet sophisticated systems in practical settings.

Like much prior art [24, 32, 35, 42, 49, 50, 74], Boomerang leverages hardware enclaves for both performance and security. Performance is arguably one of the key reasons for Tor's wide adoption in practice [39]. Yet, this is what most prior art on metadata-private messaging is still lacking. We believe that using hardware enclaves to improve performance could be the key missing ingredient for wider adoption of metadata-private messaging systems in practice. Under hardware trust, Boomerang hides users' online communication behaviors with a strong guarantee of metadata privacy and resilience against powerful attackers, while retaining good enough performance.

Besides, we envision that there could be broader options of technical routes for metadata-private messaging, with different trust assumptions, performance, and security guarantees. For whistleblowers [37, 70], their most reasonable choice is to go for designs with a cryptographic guarantee and zero trust in the servers [8, 39], where performance might be much less concerning. For the mass general users who want more than E2EE messaging (e.g., due to concerns about metadata tracking [38, 47]), we hope Boomerang could offer a better option with lowered operational cost and improved performance. These benefits can potentially attract a large user base, which is crucial in private communication systems [33].

Challenges. While trusting the enclaves brings easy-to-see benefits, such as using fewer servers for traffic mixing than prior art for reduced latency, it does not entirely solve the problems in building a scalable metadata-private messaging system. This is because: 1) secure enclaves exhibit their own threat models and attack surfaces, especially the leakage of memory access patterns [32, 90], which demand tailored system structure and oblivious algorithm designs; 2) as acknowledged by prior art [39, 54, 57, 88], even with just one trusted server, dealing with the thorny problems of active attackers that may selectively interfere with traffic, e.g., disconnecting selected clients to gain advantages in identifying targeted communicating pairs, remains hard to address; and 3) as privacy loves company [33], the need to support more clients suggests the necessity of pushing for horizontal scaling designs, which is also a hot topic in recent years [11, 52, 54, 56, 57, 87]. Notably, scalability is not only a usability requirement but also a security demand [33, 39, 54, 56, 87].

Technical overview. For metadata-private messaging, Boomerang draws many insights from the prior art, and is designed to be a performant system with cryptographic security under hardware trust. It operates in rounds for bi-directional conversations, and centers around the paradigm of adopting a private "virtual address" to facilitate obfuscated message exchange that unlinks the sender and receiver. As we overview below, its technical instantiation involves tailored oblivious algorithms for message shuffling, proactive resistance against active attacks, and horizontal scaling. For ease of presentation, we present a basic single-server Boomerang (§3) and a scalable multi-server Boomerang+ (§4).

1) Basic single-server Boomerang. From a high level, the system operates as follows: Upon proper setup, in each round, each connected client sends a message, tagged with a "private label", which is randomly derived from a pairwise shared secret with his communicating buddy, to the Boomerang server. The Boomerang server obliviously checks all the messages and swaps any pair of messages sharing the same labels for the relevant communicating pair. In this regular case, each label shows up twice each round. But active attackers might block selected clients or control a subset of clients to disrupt this regular pattern, causing each label to show up once or more than twice each round. The problem is quite subtle, because we need to fix these irregular patterns, without giving attackers any advantage in linking the remaining clients.

Boomerang's key insight is to preserve the same observable receiving pattern for each connected client, no matter how the sending pattern changes. We build oblivious algorithms for enclaves to detect messages with irregular label patterns and proactively patch them (§3.3.2) by returning those messages back to the corresponding senders, like a "boomerang". In this way, we can contain the influence attempts within the problematic clients themselves, isolated from the remaining clients. Based on a library of basic general-purpose oblivious primitives [2, 71], we design specialized oblivious algorithms (§3.3.2, §3.3.3, §4.2) for all enclave operations in Boomerang, including proactive resistance designs against active attacks and horizontal scaling in Boomerang+.

2) Scalable multi-server Boomerang+. For horizontal scalability, directly replicating the basic single-server Boomerang and letting each server process a subset of communicating pairs would not work, because pairwise clients connecting to one server have a higher possibility to communicate than those across different servers. Introducing a load balancer, known as an entry node in Boomerang+, to obliviously distribute batches of messages to a group of Boomerang nodes for message exchange would fulfill the need for "global mixing", where any pair of connected clients at the entry node might be communicating. But as pointed out by recent art [32, 89], two requirements remain: 1) a centralized proxy [82] can be error-prone and a scalability bottleneck of the underlying system; and 2) any batch structure from a load balancer must be generated using public information, so as to ensure that no sensitive information can be observable from the load balancing.

Note that these requirements are generic for any security-aware scalable system design. Answering them can be quite design specific, especially on setting the bound on batch size without triggering an overflow. Inspired by the balls-into-bins analysis [32], we model the problem of setting the maxi-

mum batch size in Boomerang+ as a weighted balls-into-bins game, where we partition the messages by their private labels and each message's weight is determined by its label pattern (showing once, twice, or more than twice). We prove an upper bound on the maximum batch size and show its marginal overhead on our horizontal scaling design. Following the oblivious load balancer design [32], we derive tailored oblivious algorithms to generate sub-batches whose distribution across rounds is indistinguishable (§4.2).

2 Threat Model and Security Goals

2.1 Threat Model and Assumptions

We consider the following threats. Attackers can: 1) observe the global network states (including the timing, volume, and link states); 2) actively tamper with the network, such as selectively dropping messages, blocking selected connections, and controlling a subset of clients; and 3) access the server-side components beyond hardware enclaves, such as the memory, files, and networks, as well as the operating system.

Boomerang is built on servers equipped with secure enclaves, where memory access patterns can be observed [32, 49, 62, 66]. Boomerang is designed to work with generic enclaves [18, 30], and we adopt Intel SGX for our implementation. Our oblivious algorithms can deal with side-channel attacks against SGX leveraging memory access patterns to extract secrets, such as the cache attacks [19, 41, 67] and paging-based attacks [20]. We assume a public key infrastructure (PKI) to help manage the public keys of clients. Communications among clients and enclaves are securely established via TLS integrated with remote attestation [1, 46, 51]. In Appendix A, we elaborate in more detail on how a client can establish the trust on a multi-enclave system like Boomerang through remote attestation, following common practices suggested from the prior art [9, 23, 35, 74, 79].

Communication model. Like many previous metadata-private messaging systems [4, 8, 54, 56, 57, 87, 88], Boomerang operates in rounds to ensure the uniformity of communication pattern among clients, and focuses on pairwise messaging among online clients who have coordinated their conversations. Processing message exchanges on round boundaries implies that clients of Boomerang send encrypted messages with a fixed rate and size, independent of their true communication activities, which allows the dealing against traffic analysis attacks. This can be done at Boomerang clients by generating "blank" messages if a user types nothing or too slow, and queuing/splitting messages if a user types too fast or sends a message of large size [88].

Under this communication model, Boomerang does not hide the fact that clients are using the system. Boomerang offers online anonymity by supporting a large scale of clients. The anonymity set includes both active clients in real conversations with their buddies and online idle clients who do not have a conversation buddy but can voluntarily send "blank" messages to themselves as cover traffic to further enlarge this anonymity set [56, 88]. We suggest the clients always keep online to disguise the real communication actions [33, 39]. Because all clients connected in the system behave the same at each communication round, we can hide the communication metadata with cryptographic security against powerful attackers. We formalize this security notion in Definition 2.1.

Bootstrapping Boomerang. Besides operating in rounds, Boomerang adopts the existing practices of a bootstrapping phase for clients to start conversations [11, 37, 52, 58]. Particularly, clients should run an "add-friend" protocol (where clients can verify each other's identity and share their secrets) and a "dialing" protocol (where pairwise clients coordinate the time to have the conversation and exchange session keys) [7]. To add a friend, the common practice is to: 1) exchange secrets in person (e.g., by showing a QR code at a coffee shop [7, 37]); or 2) use an online metadata-private add-friend protocol [58]. To dial a friend, the common practice is to adopt an out-of-band metadata-private dialing system, e.g., Alpenhorn [58]. Dialing brings additional costs to clients, which will be amortized over multiple conversation rounds. Note that similar bootstrapping phases have been adopted by prior art [54, 56, 57, 87]. Thus, throughout the paper, we keep our focus on the conversation protocol design, which is where much prior art is differentiated [7], and make simplified assumptions that the bootstrapping phase is properly done. Specifically, upon proper bootstrapping, we will narrow down the problem of metadata-private messaging to how to design an obfuscated message exchange (aka conversation) protocol among pairwise clients under hardware trust.

Attacks out of scope. Similar to other enclave-based systems [32, 35, 49, 75], we don't consider denial-of-service attacks. The enclave code is assumed to be correct and faithfully fulfilling our oblivious algorithms. Orthogonal countermeasures to recent noteworthy leakage attacks through power consumption channels [25, 68] and transient executions [48, 77] are also beyond the scope of this work. One generic limitation to all metadata-private messaging systems is the long-term intersection threat [14]. Recall that we do not hide whether clients are using the systems or not. Thus, an attacker, who observes the anonymity set of online clients changing across rounds, might infer the linkage of communicating pairs. For example, two clients that simultaneously get online or offline are more likely to be communicating. The common practice to mitigate this concern is to let the clients always stay online [56, 88] or keep the same communication pattern [11] and send enough cover traffic. We also suggest adopting orthogonal mitigation techniques [60, 92] and using more cover traffic, as also noted by Clarion [36].

2.2 Security Goals

Like much prior art, our security goal follows the unobservability concept in anonymous communication [43]. Specifi-

cally, for any two online clients, Alice and Bob, we want to ensure that an attacker cannot distinguish whether they are real buddies in a conversation or not. We want to guarantee this is true even in a malicious setting, where an attacker can control up to $m-2$ clients for m as the total number of clients connected to the system. We formalize this property below as communication pattern indistinguishability.

Definition 2.1 *Let λ be the security parameter and m be the number of connected clients in the system. Define an experiment EXP with an attacker $\mathcal{A}$ who controls $m-2$ clients:*

- *The attacker creates a pair of clients in a conversation as $P_0 = (c_0, c_1)$, and a pair of clients not in a conversation as $P_1 = (c'_0, c'_1)$ (e.g., being idle or missing buddies).*
- *The challenger randomly chooses $b \in \{0,1\}$ and plays the role of the pair of clients P_b to simulate interactions between them and the server. During this procedure, the challenger needs to simulate the payload (aka contents that can be represented by a random string) for the chosen pair of clients.*
- *The attacker observes their transcriptions and outputs a bit $b' \in \{0,1\}$ to guess the challenger's choice.*

We define the attacker $\mathcal{A}$'s advantage with respect to EXP as

$$\mathsf{Adv}_{\mathsf{EXP},\mathcal{A}} = \left|\Pr[b = b'] - \Pr[b \neq b']\right|.$$

We say that a system is communication pattern indistinguishable if the advantage $\mathsf{Adv}_{\mathsf{EXP},\mathcal{A}}$ is negligible in λ for all probabilistic polynomial time (PPT) attackers.

This definition implies that seemingly any pair of connected clients might be communicating with equal chance. We will show this is indeed the case in Boomerang and Boomerang+.

3 Boomerang: Basic Instantiation

3.1 Overview

Boomerang runs in rounds with a single enclave-based server. Following the same practice in previous round-based designs [54, 56, 57, 87, 88], Boomerang requires a coordinator to announce round numbers across the server and clients. In each round, each client sends and receives one message respectively to and from the Boomerang server. To facilitate oblivious message swapping, every message is tagged with a "private label", which is a pseudorandom string generated from a pairwise session secret between a communicating pair. (See §2 for our system setup assumptions.) Any pair of messages sharing the same private label will be swapped, which indicates a regular case when each private label shows up exactly twice each round. Messages with irregular private label patterns will be detected and looped back (§3.3.2).

Figure 1 shows an overview of our Boomerang design. The Boomerang server first (❶) sorts the packets and detects the irregular pattern of the private labels. Then, (❷) it proactively fixes the irregular pattern via the proactive pattern patching algorithm (§3.3.2). Finally, (❸) it sorts the messages by receiver IDs and sends them to the clients (§3.3.3).

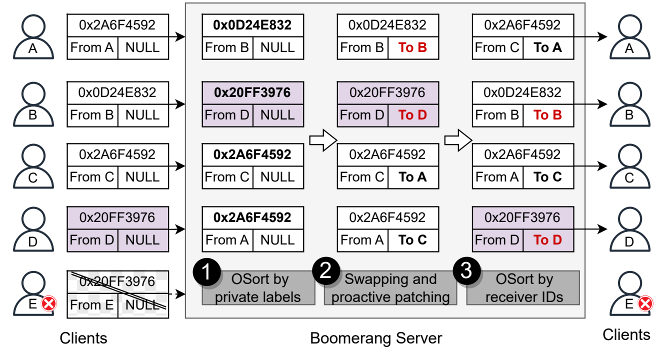


Figure 1: Overview of Boomerang. In this example, A and C are talking. B is idle. D and E are talking, but E is blocked as described at the end of §3.3.2. Boomerang proactively patches the pattern for D and B (Step ❷).

3.2 Client

Figure 2 shows the pseudocode for client operations. We consider two modes of clients: active clients in conversations with their buddies and idle clients sending dummy cover traffic (as discussed in §2.1). Regardless of being active or idle, each connected client needs to send one message to the Boomerang server in each round.

Packet preparation. A packet `Pkt` includes the following fields: 1) private label with l bits, `priv_label`; 2) receiver's identifier, `R`; 3) sender's identifier, `S`; 4) round number, `round_num`; and 5) content encrypted by the session key shared with the buddy (Lines 4-6). When preparing the packet, the client fills in all fields except for the receiver's identifier (e.g., set to `NULL` by default). If the active client has nothing to deliver to her buddy, she should fill the content anyway, e.g., with a message saying "this is a blank message". The client then encrypts the packet using the session key shared with the secure enclaves (Line 7). Finally, the client sends the encrypted message to the Boomerang server.

Idle clients. To hide real communication actions, clients need to keep sending messages even if they are not in an active conversation [56, 88]. We let idle clients randomly generate a private label (not shared with others). By design, the server will loop all unpaired messages back to senders.

Exception alert. Normally, after sending the prepared packet to the Boomerang server, the client is expected to receive one packet carrying the message from the buddy/herself every round. Otherwise, the system will raise exceptions for abnormal cases. If there is no message returned by the decryption procedure, it means that the packet to/from the server has been lost. In this case, both active and idle clients should be

```
def active_client(round_num, priv_label, my_id,\
            content, ses_key_encl, ses_key_buddy):
    # Prepare the packet
    Pkt pkt; pkt.round_num = round_num
    pkt.priv_label = priv_label; pkt.S = my_id
    pkt.enc_content = Encrypt(content, ses_key_buddy)
    enc_pkt = Encrypt(pkt, ses_key_encl)
    # Send the packet to the Boomerang server \
    #                  running boomerang_server()
    send_msg = (my_id, enc_pkt)
    # Receive the packet from the Boomerang server
    recv_msg = rpc.request(send_msg)
    pkt = Decrypt(recv_msg, ses_key_encl)
    # Exception alert
    # If there is message loss
    if pkt ==  None: raise("Message loss")
    # If the original message is looped back
    if pkt.R == my_id: raise("Buddy blocked")

    return Decrypt(pkt.enc_content, ses_key_buddy)

def idle_client(round_num, my_id, ses_key_encl):
    # Prepare the packet
    Pkt pkt; pkt.round_num = round_num
    pkt.priv_label = random(); pkt.S = my_id
    pkt.enc_content = ciphertext.random()

    send_msg = (my_id, Encrypt(pkt, ses_key_encl))
    recv_msg = rpc.request(send_msg)

    if Decrypt(recv_msg, ses_key_encl) ==  None:
        raise("Message loss")

    return True
```

Figure 2: Pseudocode for client operations.

alerted about their unreliable network conditions (Lines 13-16 and Lines 31-32). For those successfully decrypted packets, active clients then check whether their messages are from their buddies (Line 18). Note that in Boomerang design, if the receiver ID is the same as the sender ID (indicating that the message is sent back like a "boomerang"), the client should be alerted that his/her buddy has been blocked and can choose to resend the message in the next round or stop the conversation immediately (by changing to idle mode).

3.3 Server

Figure 3 shows the pseudocode for oblivious server operations, including two main functions: 1) oblivious irregular pattern detection and 2) oblivious proactive pattern patching. Below we introduce the background of oblivious primitives and then describe the main ideas of our designs.

3.3.1 Background of Oblivious Primitives

We build Boomerang's oblivious algorithms over an existing library of general-purpose oblivious primitives developed by XGBoost contributors [2, 55], which is also based on libraries provided in previous noteworthy enclave-based data analytic systems [71, 73]. The library offers basic oblivious functions, including comparisons, assignments, sorting, etc. These oblivious functions are fundamentally built on register-to-register operators, which are private to the processor and immune to memory access pattern leakages [55, 71, 73].

Oblivious comparisons/assignments. This set of primitives can conditionally assign or compare values on the register level without revealing the results of the comparison or assignment. In this paper, we use `O_Equal(a, b)`, `O_Less(a, b)`, and `O_Choose(cond, a, b)` [2] for comparison and conditional assignment. `O_Equal(a, b)` outputs `True` if $\mathtt{a} = \mathtt{b}$ (otherwise outputs `False`). `O_Less(a, b)` outputs `True` if $\mathtt{a} \leq \mathtt{b}$ (otherwise outputs `False`). `O_Choose(cond, a, b)` chooses from two values given a boolean condition without leaking which value is chosen. If `cond` = `True`, it outputs value `a` (otherwise outputs value `b`).

Oblivious sort (`O_Sort(key, array)`). An oblivious sort algorithm outputs ordered data without revealing any information (e.g., the original order) about the input data [10]. In our instantiation, we choose bitonic sort [12], which is highly parallelizable with $O(n \log^2 n)$ computational complexity. Bitonic sort compares and swaps the items in a fixed and data-independent order, and thus is oblivious by design.

3.3.2 Oblivious Proactive Pattern Patching

Detecting irregular pattern. When receiving a batch, the server first needs to identify all the regular and irregular messages for subsequent processing. The server first obliviously sorts the batch by `priv_label` (Step 2 in Figure 3, Line 12). This step is to map the messages of the same label (which implies a communicating pair) together for further pattern detection. Before introducing the detection algorithm, we first show the possible label patterns after sorting. Since the private label is a l-bit pseudorandom string shared between the pair, the possibility that attackers can guess it and forge messages is negligible if l is sufficiently large (e.g., 256 bits). Therefore, a communicating pattern indicates a double-accessed label, which we regard as a regular pattern. Otherwise, the pattern is regarded as irregular. There are three possible label patterns after `O_Sort`, as shown below

- double: messages from communicating pairs;
- single: messages from idle clients or incomplete pairs (caused by client churn or malicious blocking);
- more-than-two: multiple messages with repeating labels controlled by an attacker.

To detect irregular patterns, the server linearly scans the ordered packets using oblivious comparison primitives to identify the above three patterns. Basically, we can achieve this by scanning the messages with a sliding window of three each time, as shown in Step 3.1 in Figure 3. For each message, we compare its private label with its previous, next, and next next messages, respectively (Lines 17-22). This will give us the relationship of the packets in the sliding window.

We first clarify how to determine more-than-two cases. If the label of a message is the same as its two subsequent messages (`is_next2_same` = `True`), this implies that this private

```
1  def boomerang_server(round_num, recv_msgs, session_keys):
2      pkts = [], R_list = [] # Receiver IDs
3      recv_msgs = Dedup(recv_msgs)
4      # Step 1: decrypt the messages
5      for (S, enc_pkt) in recv_msgs:
6          pkt = Decrypt(enc_pkt,session_keys[S])
7          # Filter out messages not in this round
8          if pkt.round_num != round_num:
9              continue
10         pkts.append(pkt)
11     # Step 2: Pair the private labels
12     pkts.O_Sort(key=pkt.priv_label)
13     # Step 3: Oblivious proactive pattern patching
14     is_mtt = False # mtt: more_than_two
15     for pkt in pkts:
16         # Step 3.1: Detect irregular pattern
17         is_prev_same = O_Equal(pkt.priv_label,\
18                                  Prev(pkt).priv_label)
19         is_next_same = O_Equal(pkt.priv_label, \
20                                  Next(pkt).priv_label)
21         is_next2_same = O_Equal(pkt.priv_label, \
22                              Next(Next(pkt)).priv_label)
23         # Detect more-than-two (mtt) pattern
24         is_mtt = is_mtt and is_prev_same or is_next2_same
25         # Detect and patch single patterns in Lines 27-28
26         # Step 3.2: Swapping and patching
27         pkt.R = O_Choose(is_next_same, Next(pkt).S, pkt.S)
28         pkt.R = O_Choose(is_prev_same, Prev(pkt).S, pkt.R)
29         pkt.R = O_Choose(is_mtt, pkt.S, pkt.R)
30     # Step 4: Re-order messages by receiver IDs
31     pkts.O_Sort(key=pkt.R)
32     # Step 5: Encrypt and send the messages
33     send_msgs = {}
34     for pkt in pkts:
35         enc_pkt = Encrypt(pkt, session_keys[pkt.R])
36         send_msgs[pkt.R] = enc_pkt
37         R_list.append(R)
38     rpc.response(R_list, send_msgs)
```

Figure 3: Pseudocode for server operations.

label must occur at least three times. Then, we can identify this message as a more-than-two case (by setting the flag `is_mtt` = `True`). Another observation is that, if a message's label repeats as a detected more-than-two case, this message must belong to the same case. We identify this by checking whether a message's previous neighbor belongs to a more-than-two case (based on the flag `is_mtt` as set in the previous iteration), and whether the message's label repeats that of its previous neighbor (based on the flag `is_prev_same`).

To determine single cases, we check both the message's previous and next neighbors. If inequality holds for both neighbors (`is_prev_same` = `False` and `is_next_same` = `False`), this message belongs to the single case. To save operational costs, we integrate this logic with the swapping process in Step 3.2 (Lines 27-28). This finishes the identification part. In this step, the server has obtained information about the relationship of the private labels in an ordered sequence and determined whether the label pattern falls into an irregular case. Next, the Boomerang server will swap the regular messages and patch the irregular ones.

Swapping and patching. For regular patterns, the server swaps the sender IDs of the two messages and assigns their values to the receiver ID fields (Lines 27-28). Specifically, if the current label is the same as that of the previous (next) packet, we assign its receiver ID field to have the value of the sender ID of its previous (next) packet. The oblivious choose function helps us conditionally assign the sender ID values to the right packets. It ensures that the server will not learn which part is actually copied to the receiver ID field.

For irregular patterns, Boomerang proactively patches them by looping back such "single" and "more-than-two" messages to its sender (like a boomerang), i.e., setting `pkt.R` = `pkt.S`, where `R` and `S` denote the identifiers of the receiver and sender, respectively. For single cases, the server obliviously assigns `pkt.R` to have the value of `pkt.S`, if the conditions of `is_next_same` and `is_prev_same` are both `False` (Lines 27-28). Finally, for more-than-two cases, where `is_mtt` = `True`, the server also loops back the packet (Line 29). Note that the original messages do not explicitly carry the information of the receivers. This makes sure that only expected "collisions" on private labels can push forward message delivery. In other words, the server will not obtain the receiver IDs from messages with irregular label patterns.

This step patches the receiving pattern of idle clients, active clients with blocked buddies, and clients controlled by an attacker. With Boomerang, messages with irregular label patterns will not influence their receiver's receiving pattern. For the example in Figure 1, the message from E to D is blocked, in which case D would not receive any message if there were no pattern patching. The Boomerang server loops back D's message by setting D as the receiver (as shown in Step ❷), defeating active attacks.

3.3.3 Oblivious Re-order

After the proactive patching, Boomerang needs to re-order the label-ordered sequence. This step is necessary, because adjacent messages in the label-ordered sequence will imply a high possibility that they share the same private label, indicating that the receivers are talking to each other. Hence, we choose to use `osort` with the order of receiver identifiers (IDs) to obliviously re-order the sequence.

Sorting the outgoing messages by receiver IDs would reveal the set of receiver IDs (along with their order), which are essential fields to be reported to the server for message transmission anyway. Recall that in the proactive patching step, the server carefully assigns the values of sender IDs to receiver IDs via swapping and patching. This ensures that the receiver set is exactly the same as the sender set (publicly observable to attackers), and each receiver will receive exactly one message. Therefore, sorting the messages by receiver IDs, which reveals the ordered receiver ID list, would not reveal additional information about the conversation metadata.

Remark. This completes our round-based oblivious message exchange in Boomerang design. Note that at the beginning of each round, we can further employ a textbook deduplication procedure in enclaves to filter repeated packets, just in case a malicious host might try to cause an exchange failure

(DoS attack) by injecting replicated packets from the network stack to let the packets' private labels appear more than twice. Similar treatment has also been done in prior art [32, 56, 57].

4 Boomerang+: Horizontal Scalability

4.1 Overview

As noted in §1, for horizontal scalability, directly replicating the single-server Boomerang, with each server processing a subset of clients, would not work because it immediately implies that clients connecting to different servers are not able to talk to each other. To address the problem, one direct idea is to employ an entry node as a load balancer to obliviously distribute batches of messages from all clients to a group of Boomerang nodes for message exchange. Here, the oblivious design is supposed to hide the mapping between the messages and the Boomerang nodes from an attacker. While this opens up a possible pathway to scale Boomerang, subtle issues remain: how to set up the batch structure?

Recent studies on distributed oblivious data stores [32, 89] have pushed forward the understanding on security-aware scaling designs. Particularly, an oblivious load balancer must set up the batch structure only using public information, independent of the input distribution. In this way, an attacker would not observe any sensitive information from the load balancing. We note that these requirements for setting up an oblivious load balancer are generic to any security-aware scalable system designs. Yet, answering them can be quite design specific. Indeed, a batch structure of data requests generated by an oblivious balancer for a distributed data store, where data partitions are fixed at each server across all rounds [32], would be ill-suited for obliviously distributing batches of messages in Boomerang+, where messages might be mapped to different Boomerang nodes each round.

Setting a batch size for Boomerang+. This has motivated us to search for solutions specific to our Boomerang+ design. In our context, the public information at an entry node is the m messages from m connected clients, and n Boomerang nodes (the back-end nodes for message exchange). Functionality-wise, we need to partition the messages by their private labels, which are random (§4.1), to facilitate the exchange of messages sharing the same labels at the same servers. For security, we need to ensure that the batch structures reveal nothing about the input distribution. For performance, we must set the batch size B as small as possible, but without triggering the overflow (otherwise, there will be dropped messages).

Inspired by the balls-into-bins analysis [32], our problem of finding the bound on batch size B in distributing m messages by their random labels (balls) to n servers (bins) can be translated to: what is the maximum load of balls into any bin? In §3.3.2, we have shown each message's label pattern as single, double, or more-than-two. This suggests that each message carries a different weight in the distribution, and

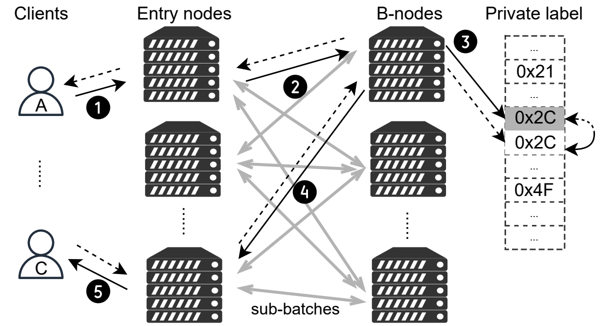


Figure 4: Overview of the scalable Boomerang+ instantiation.

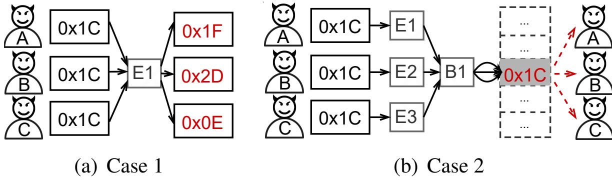
thus we need to answer the question in a *weighted* balls-into-bins game. In Appendix B, we show the complete analysis and proofs to derive the maximum batch size in Boomerang+ (results listed below for easy reference), by applying classic results [13, 76] from the balls-into-bins literature to our problem context.

Theorem 4.1 *For any set of m messages, n Boomerang nodes, and a security parameter λ, satisfying $m \gg n(\ln n)^3$ and $\lambda/\log_2 n > 1$, let $B(m,n)$ be a function that outputs the maximum batch size B for each node in Boomerang+. Then the probability of overflow is negligible in λ if we choose*

$$B = \left\lceil \frac{m}{n} + 4\sqrt{\frac{m \ln n}{3n}\left(1 - \frac{1}{\lambda}\frac{\ln\ln n}{2\ln 2}\right)} \right\rceil.$$

Based on the formula, the maximum batch size will be dominated by m/n, with the extra padding size per batch varying with different choices of messages m and Boomerang nodes n. Note that $\lambda/\log_2 n > 1$ can always hold in practice because n is always smaller than 2^λ for any reasonable security parameter λ. As shown in Appendix B, Figure 13, with $\lambda = 128$ and $m = 2^{16}$, when we scale the number of Boomerang nodes n from 4 to 28, the ratio of extra paddings over real messages ranges from 2% to 8%. With the maximum batch size settled, we will describe our oblivious "load balancers" (entry nodes) and the architecture of Boomerang+ next.

Architecture. Figure 4 shows Boomerang+. We leverage the classic two-layer architecture, consisting of entry nodes and Boomerang nodes (B-node for short) to share traffic and computation workload. The message transmission flow is summarized as follows. (❶) Each client connects to one entry node. (❷) Entry nodes generate oblivious sub-batches for B-nodes (§4.2). (❸) Each B-node merges the sub-batches from entry nodes and processes messages like a single Boomerang server (e.g., proactive irregular pattern detection and patching, §3), except for one additional step to swap the entry node identifiers of the pairs (§4.3). (❹) Upon done with the processing, B-node sends the swapped messages back to entry nodes. (❺) Finally, entry nodes merge the sub-batches, pad for possible lost messages, and send them back to receivers.



(a) Case 1 (b) Case 2

Figure 5: An illustration of more-than-two label patterns on (a) entry nodes and (b) B-nodes. The red dotted arrows in (b) imply that the messages to "0x1C" will be looped back to the malicious clients ultimately.

4.2 Entry Nodes

4.2.1 Irregular Pattern Patching

Similar to the basic single-server Boomerang, when receiving a batch, the entry nodes first decrypt and obliviously sort the batch by `priv_label`, and then detect and patch irregular label patterns. But different from the proactive pattern patching algorithm at the basic Boomerang (§3.3.2), here the entry nodes only need to handle the more-than-two cases and leave the single ones to B-nodes.

Patching the more-than-two cases on entry nodes, in case of multiple clients controlled by an attacker sending multiple messages carrying the same private label, is essential. Because such a threat will cause workload skew [32] and potentially influence the non-overflow guarantee of our weighted balls-into-bins algorithm. To eliminate the skew, we let the entry node replace the redundant private labels with new random labels. To detect the more-than-two label patterns, entry nodes adopt the detection algorithm in Boomerang (Step 3.1 in Figure 3, Lines 17-25). Next, instead of looping the irregular messages back, the entry node reassigns new random private labels to them, as shown in Figure 5(a). To set it obliviously, we assign `pkt.priv_label` to have the value of `O_Choose(is_mtt, random(), pkt.priv_label)`.

Since the fresh private label is randomly chosen with negligible possibility of collision, the irregular messages will be regarded as a single pattern to be looped back in subsequent processes on the Boomerang node. In this way, we can eliminate workload skew without revealing the number of malicious messages or changing the communication pattern.

4.2.2 One-time Message Assignment

The message assignment function has two main goals:

• (function goal) assigning the messages with the same private labels to the same B-node, no matter which entry nodes the clients are connecting to, and

• (security goal) ensuring the assignment is (pseudo)random, and the distribution of the sub-batches across rounds will not leak the communication pattern.

For the function goal, the intuition is to derive B-node ID

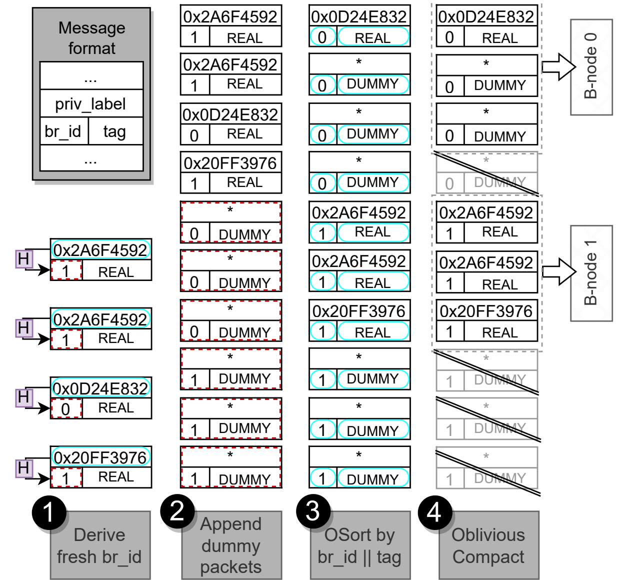


Figure 6: An example of entry node operations. "H" in Step 1 refers to the mapping function (Eq. (1)). B is set to 3. The blue box with rounded corners refers to read operations, and the red box with dotted lines refers to write operations.

(denoted as `br_id`) from the private label using the same deterministic function across all entry nodes. For example, we can simply compute the identifier from the private label modulo the number of B-nodes: $\texttt{br_id} = \texttt{priv_label}\%n$, where n refers to the number of B-nodes. For the security goal, we make the mapping function change across rounds.

Specifically, we use a keyed hash function $\mathsf{H}_k(\cdot)$ to derive a fresh string from the private label, round number, and a secret key k shared among the enclaves of all entry nodes. We apply the modulo function to the fresh string and the number of B-nodes. The assigned B-node is:

$$\texttt{br_id} = \mathsf{H}_k(\texttt{priv_label}||\texttt{round_num})\%n. \quad (1)$$

The keyed hash function provides a fresh mapping from the private label to the B-node every round, so attackers cannot predict any future assignments in new rounds.

4.2.3 Oblivious Sub-batch Padding

Although the distribution is fresh across rounds, the true size of each sub-batch is revealed, which brings security concerns in long-term communications [32, 52, 87]. Based on our derived maximum batch size in §4.1, we opt to pad the sub-batches to equal size B, which is calculated from public information, namely m messages and n B-nodes, and will not carry any private information about the content.

The remaining step is to obliviously pad the sub-batches without leaking the original sizes. We here follow the oblivious padding algorithm in Snoopy [32]. The padding steps are

shown in Figure 6 and described as follows. Firstly (❷), the entry node appends B dummy messages (with `tag = DUMMY`) for each sub-batch to the message sequence. The private labels in dummy messages are randomly generated. Secondly (❸), it obliviously sorts the message sequence by `br_id||tag`. After the sorting, we can get a group of sub-batches, each of which is formed of real messages followed by B dummy messages. Finally (❹), it squeezes extra dummy messages using oblivious compaction. An oblivious compaction algorithm [40] (`O_Compact(flag, array)`) can remove the items in an array with certain flags without leaking which items are removed. The complexity is $O(n \log n)$.

To decide which messages to send or remove obliviously (that is, to set the `flag` for each message), the entry node linearly scans the sorted batch and keeps a counter (`c`) to record the relative position of the message in a sub-batch. If the message is among the first B messages in its sub-batch ($c \le B$), the message should be sent (`flag = True`). Otherwise, the flag should be set to `False`. We next introduce how to obliviously iterate `c` and assign `flag`. The counter is initially set to 1. When iterating through the batch, the node accumulates the counter (`c+1`) if the current `br_id` repeats its previous one. Otherwise, set the counter to 1, meaning that the node has finished processing the current sub-batch and encounters the first message for a new sub-batch. With the right counter c, the node can obliviously assign `flag` accordingly. The pseudocode for the above step is as follows.

```
is_br_same = O_Equal(pkt.br_id, Prev(pkt).br_id)
c = O_Choose(is_br_same, c+1, 1)
flag = O_Less(c, B)
```

B is the upper bound calculated from our weighted balls-into-bins algorithm, which guarantees that real messages assigned to the same B-node will not exceed size B (except with negligible probability). Therefore, all real messages will be marked with `flag = True` and not be dropped. Finally, the node uses oblivious compaction to remove extra dummy messages (those marked with `flag = False`).

With the steps above, the size of the sub-batch for each group is exactly B, and the attacker cannot differentiate the dummy messages from the real ones. The entry nodes then send sub-batches to B-nodes for further processing.

4.3 Boomerang Node

Like the basic Boomerang, B-nodes also need to: 1) detect irregular access patterns and patch them (i.e., the patching algorithm in §3.3.2); and 2) swap the messages carrying the same private label. Figure 7 shows operations on B-nodes, among which most operations are the same as Boomerang.

Similarly, there are two types of irregular label patterns on B-nodes: 1) single pattern and 2) more-than-two pattern. Besides idle clients and incomplete pairs as discussed before, single patterns may also come from the irregular messages

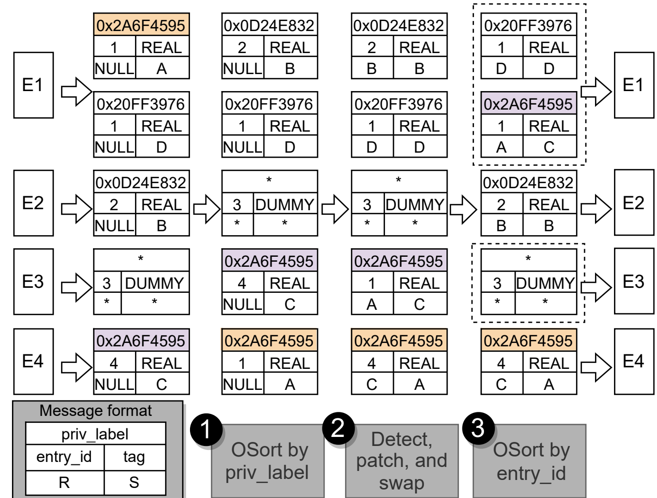


Figure 7: An example of B-node operations. In this example, A and C are talking to each other, B talks to herself (idle client), and the message from someone to D is blocked.

with private labels reassigned by entry nodes. Interestingly, although entry nodes have eliminated workload skew, more-than-two patterns might still appear on B-nodes. As shown in Figure 5(b), malicious clients could send messages with the same private label through multiple entry nodes, with only one message to each entry node, evading the irregular pattern detection. Ultimately, these messages will end at some B-node and appear as more-than-two patterns. To detect such irregular patterns, B-nodes follow the same detection and patching algorithm as Boomerang (Steps 2 and 3 in Figure 3).

Different from Boomerang, B-nodes cannot directly send the processed messages to clients, because this reveals the identities of clients connecting to the same B-node, indicating they are more likely to be talking. Instead, we let B-nodes send each message back to the entry node first, and then let the entry node send the message back to the receiver connecting to it. The entry node ID of the receiver can be obtained from the paired message (if any) with which it was swapped. Hence, (❷) the B-node can swap entry node IDs (`entry_id`) of the paired messages during the linear scan. Similar to the receiver ID swapping step in Boomerang, the B-node swaps (or loops back) both the entry node IDs and receiver IDs using oblivious choose. Finally (❸), the B-node obliviously sorts the sequence by entry node IDs (to group the batch for each entry node together) and sends the messages back to the corresponding entry nodes.

In Boomerang, the last step is to re-order messages. We accordingly move this step to entry nodes. When receiving the sub-batches from B-nodes, entry nodes merge the sub-batches, sort the batch by receiver IDs, pad for possible missing messages (as we will discuss in §4.4), remove extra dummy messages, and send the messages to receivers.

4.4 Server Churn

So far, we have introduced our mechanisms (e.g., one-time message assignment and oblivious sub-batch padding) to make the communication pattern among servers oblivious to attackers. However, such mechanisms only consider the "accidents" from clients. For completeness, here we discuss some possible accidents that may happen to servers and introduce Boomerang+'s enhancing strategies.

B-node churn. B-nodes can be blocked (e.g., under DoS attacks) or accidentally go offline without sending back the messages, leading to missing patterns on the clients whose messages are processed on the corresponding B-node. This gives an attacker a chance to link the clients with certain B-nodes. Towards this threat, we let the entry node proactively patch the dropping pattern due to the lost connection with some B-nodes. The patching algorithm is very similar to the oblivious sub-batch padding algorithm in §4.2.3. Firstly, the entry node appends one dummy message for each client connecting to itself. This is feasible because the node can record the list of connecting clients when it processes the incoming messages in the very beginning. Secondly, the entry node obliviously sorts the batch by receiver identifiers. Finally, it linearly scans the sequence, marks which messages to send, and obliviously compacts the batch. After the patching, we can ensure that all clients connecting to the entry node will receive one message in each round, no matter whether there is any B-node churn or not. For efficiency, we only trigger this patching algorithm when a connection loss occurs. Once a B-node goes offline, entry nodes will ignore this node and not assign messages to it in the next round (according to the instructions from the coordinator). In this way, we make sure that B-node churn will not cause severe service denial.

Entry node churn. This case is similar to the case where the clients are blocked, but on a larger scale. The direct impact is that the buddies of clients connected to this entry node will fail to form a paired private label, and thus their messages will be looped back (by other entry nodes). In this case, attackers observe nothing but a predictable situation where a set of clients cannot link to the churned entry node.

5 Analysis

We now show Boomerang and Boomerang+ achieve communication pattern indistinguishability as we claimed in §2. Thanks to the oblivious designs, we ensure that all connected clients behave the same in every round, either exchanging messages with a buddy or sending messages to themselves.

Theorem 5.1 *Given oblivious comparison/assignment operations, an oblivious sort algorithm, and an oblivious patching and swapping algorithm, Boomerang achieves communication pattern indistinguishability presented in Definition 2.1.*

Proof sketch. We focus on the setting defined in Definition 2.1, where $m-2$ clients are compromised by the attacker. As Boomerang operates in rounds, in view of the attacker, the only way to distinguish the communication pattern of two given clients is from the observation of memory access patterns during the "obfuscated" message exchange procedure. Thus, the security of Boomerang hinges on the oblivious algorithms designed for proactive pattern patching and re-order (in §3.3.2 and §3.3.3). As the re-order algorithm is essentially the oblivious sorting [10], we mainly pay attention to proving the obliviousness of proactive pattern patching, captured by Lemma C.1 in Appendix C.1. When all involved algorithms are oblivious, it is easy to derive that the attacker cannot identify whether two clients are communicating or not within each round, except with negligible probability. For the fixed system configuration, it follows that the attacker's view will remain the same across rounds. The full proof of the Theorem 5.1 can be seen in Appendix C.1.

We next show that Boomerang+ achieves the same communication pattern indistinguishability as Boomerang, even though multiple servers are introduced.

Theorem 5.2 *Given a cryptographic hash function, oblivious comparison/assignment operations, an oblivious sort algorithm, an oblivious patching and swapping algorithm, an oblivious sub-batch padding algorithm, and an oblivious compaction algorithm, Boomerang+ achieves communication pattern indistinguishability presented in Definition 2.1.*

Proof sketch. Following the proof sketch in Theorem 5.1, here we also focus on the settings where $m-2$ clients are controlled by the attacker. With multiple entry nodes and B-nodes deployed, we will first show that Boomerang+ assigns a message to each B-node uniformly due to the deployment of the cryptographic hash function and our security-aware load-balancing design. Then, we show that for the two targeted clients, whether they are communicating or not, the probabilities that their messages are assigned to one single B-node are always equal. Thus, combined with Theorem 5.1, we have that an attacker cannot identify whether two clients are communicating or not in Boomerang+. The detailed analysis can be seen in Appendix C.2.

It is intuitive to see that Boomerang+ is horizontally scalable. See Appendix B for the scalability analysis.

6 Implementation and Evaluation

We implement Boomerang and Boomerang+ in about 4000 lines of C++ code. We build secure enclaves using the framework of Intel SGX v2.16 on Intel SGX DCAP Driver v1.14 and use Intel's AVX-512 SIMD instructions for basic oblivious primitives. We build our oblivious algorithms using the oblivious library from XGBoost [2]. Clients and Boomerang(+) servers communicate using gRPC v1.35 on asynchronous RPC mode over TLS. The Boomerang(+) prototype is available online at https://github.com/CongGroup/boomerang.

6.1 Evaluation Overview

We experimentally answer the following questions: 1) How fast are Boomerang and Boomerang+? 2) Can Boomerang+ scale by adding servers? We highlight some results below:

- Boomerang achieves 99th percentile latency of 1.41 second for 2^{16} clients on one 16-core server. For space, we present the results in Appendix D.
- Boomerang+ achieves 99th percentile latency of 615 ms for 2^{16} on 16 servers and 7.76 second latency for 2^{20} clients on 32 servers, respectively.

Experiment setup. We evaluate Boomerang(+) on Tencent Cloud M6ce VMs [83], with Intel Xeon Ice Lake processors with Intel SGX support [45]. For Boomerang, we use one `M6ce.4XLARGE128` instance (16 vCPU, 128 GB of memory, and 13 Gbps of network bandwidth). For Boomerang+, we assign 12 `M6ce.4XLARGE128` instances as entry nodes and 4 `M6ce.4XLARGE128` instances as B-nodes by default. For completeness, we also evaluate three metadata-private communication systems: Pung (XPIR) [8], XRD [54] and Addra [4]. According to Azure pricing [65], instances with TEEs are roughly twice as expensive as those without TEEs at the same level of computing power. Therefore, to compensate for the machine cost of the trusted hardware Boomerang uses, we allocate twice the number of machines (or total CPU cores) for these systems. For XRD, we use 32 `M6.4XLARGE128` instances (16 vCPU, 128 GB of memory, and 13 Gbps of network bandwidth) to construct the chain-based architecture. For Pung, we use one `M6.4XLARGE128` instance and run 1/32 total traffic over it, following the setting in XRD [54]. Since Pung is directly parallelizable, letting one server share 1/N of the total traffic is the best performance it can possibly achieve. For Addra, we use one `S4.8XLARGE128` instance (32 vCPU, 128 GB of memory, and 11 Gbps of network bandwidth) as the master and 30 `M6.4XLARGE128` instances as the workers. For clients in the four systems, we use one `C5.26XLARGE368` instance (104 vCPU, 368 GB of memory, 36 Gbps of network bandwidth) to run simulated clients, each sending/receiving one RPC request at a time. We run instances in the same data center to save bandwidth and simulate client-server round trip latency of 100 ms by using Linux `tc` command. Results are averaged 20 times for each experiment.

Parameters. We set the message size to 256 Bytes and the private label to 256 bits. We set the conversation round to 12 seconds, according to the latency results from our experiment on Boomerang+ dealing with 2^{20} messages. We set the batch size B according to Theorem 4.1, with the security parameter $\lambda = 128$. For XRD, we construct 32 chains, each of which consists of 30 machines. The length ensures that the probability of the existence of a group of malicious servers is less than 2^{-64} if 20% of the servers are malicious. We let each client send 8 messages to 8 chains, following XRD's recommendations. For Pung, we use recursion with a depth of 2 and set the bucket size to 64.

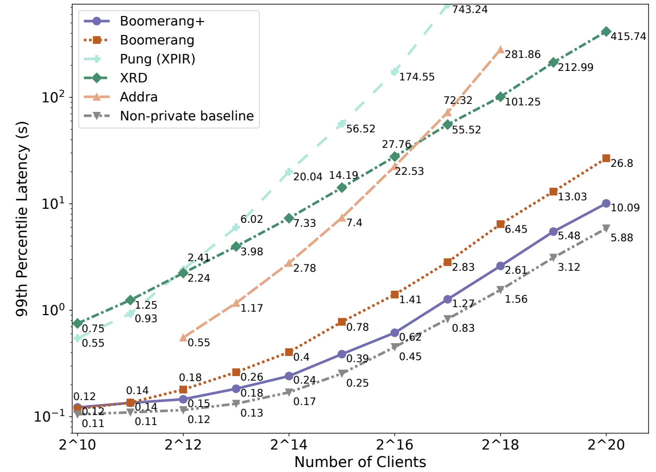


Figure 8: 99th percentile latency of Boomerang, Boomerang+, and other baselines with a varying number of clients.

6.2 Boomerang+ Performance

Latency and throughput. Figure 8 depicts the latency of Boomerang, Boomerang+, Pung (XPIR) [8], XRD [54], Addra [4], and a non-private baseline. Pung, XRD, and Addra are the latest work achieving pairwise metadata-private communication under cryptographic security, but with different trust assumptions. XRD operates on fractional trusted servers, and Addra and Pung operate on fully untrusted servers. We notice that the latency results of Addra and XRD are higher than those reported in their papers. This is perhaps because we used fewer and less powerful machines. Besides, we only ran Addra over up to 2^{18} clients, as our testbed (one 32-core master and 30 16-core workers) could not afford a workload for clients more than 2^{18}.

Boomerang+ achieves 615 ms latency for 2^{16} clients and 10.09 second latency for 2^{20} clients. The throughput of Boomerang+ reaches 85.4K messages per second under 16 machines. Compared to the prior systems based on cryptographic primitives, Boomerang+ is significantly more efficient thanks to leveraging the trusted hardware. For example, for 2^{16} clients, Boomerang+ is 36× faster than Addra and 45× faster than XRD. We also evaluate a non-private version of Boomerang+ to show the cost of non-oblivious operations (most of which is from the network operation cost). In this baseline, we keep the same round-based design and traffic transmission flow (i.e., the same two-layer network architecture) but remove all security-enhancing operations in enclaves (e.g., oblivious sort, padding, patching, etc.). Results (the grey dotted line in Figure 8) indicate that the computational overhead of the oblivious operations for metadata hiding over that of other basic operations is small. Interestingly, compared to Boomerang (the basic instantiation), Boomerang+ does not have an overwhelming advantage when the clients are less than 2^{15}. This is because Boomerang+ involves more obliv-

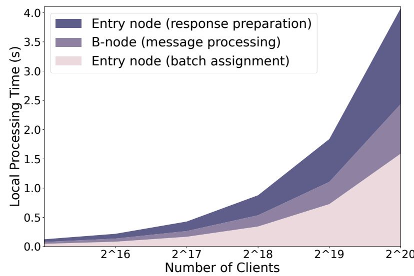


Figure 9: Breakdown of Boomerang+ operation costs.

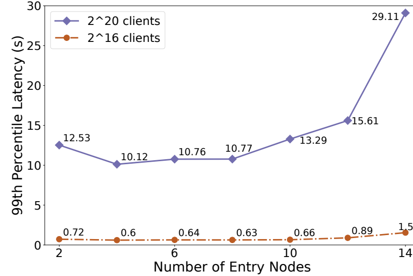


Figure 10: Latency with varying numbers of entry and B-nodes (sum fixed to 16).

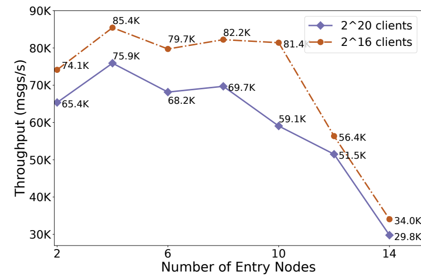


Figure 11: Throughput with varying numbers of entry nodes and B-nodes.

ious operations (e.g., the padding, oblivious sort, and compaction on the entry nodes) to take care of potential threats due to the scalable design. When the clients scale to 2^{20}, the benefits of horizontal scaling gradually show up: Boomerang+ runs 2.66× faster than Boomerang. In §6.3, we will further show the horizontal scalability of Boomerang+; that is, it can achieve lower latency by adding more servers.

Bandwidth cost. Boomerang+ requires one message per round for both active and idle clients. On the server side, Boomerang+ involves dummy paddings, and the overhead is relatively small. According to our calculation, for a workload of 2^{15} messages on one entry node to four B-nodes, dummy messages account for less than 4% of all messages (details about the padding overhead in Figures 13 and 14 in Appendix B). Luckily, on Azure Cloud and many other clouds, data transfer within one VPNet is free [63].

6.3 Microbenchmarks

Breakdown of Boomerang+ operational cost. To evaluate the computational cost on entry nodes and B-nodes, we break down Boomerang+ latency into three parts: 1) entry node batch assignment, 2) B-node message processing, and 3) entry node response preparation. Figure 9 shows the processing time for 8 entry nodes plus 8 B-nodes, each of which handles (almost) an equivalent volume of messages. Besides, we only record the local processing time on each node and eliminate the network cost (e.g., RPC request and response with clients) here. In this way, we can clearly see the computational overhead at each stage. Note that the total operation latency is smaller than the end-to-end latency presented in Figure 8. This is because processing RPC requests is time-consuming in our implementation, especially under a large number of requests. Generally, facing the same volume, the entry nodes (both the first and third stages) operate longer than B-nodes. We conjecture that assigning a few more machines as entry nodes than B-nodes with a fixed total number of machines may save latency, as shown below.

Resource allocation on entry nodes and B-nodes. As discussed before, resource allocation can be important for Boomerang+'s overall performance. We have tested the latency and throughput under different allocations on entry nodes and B-nodes using 16 machines in total, as shown in Figures 10 and 11, respectively. We find that Boomerang+ performs the best over 12 entry nodes and 4 B-nodes. This is because the operational cost is higher on entry nodes, and assigning more entry nodes reduces the average workload on each entry node. This observation is consistent with the breakdown results, where processing on entry nodes takes longer than that on B-nodes. We also report the performance of different allocations given different total numbers of servers in Appendix E.

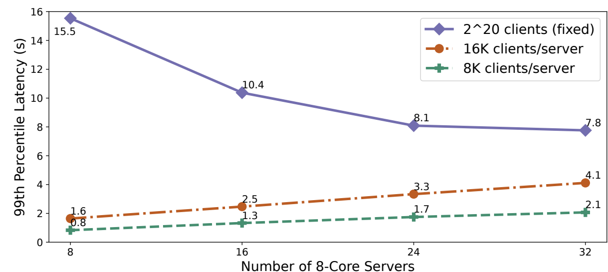


Figure 12: Latency with a varying number of servers.

Scaling by adding servers. With horizontal scalability, Boomerang+ should, ideally, achieve the same level of latency by adding more servers when handling more clients. In this experiment, we adopt 32 8-core `M6ce.2XLARGE64` instances to test the scalability of Boomerang+ over more servers.

We first set a fixed total number of clients to 2^{20} and gradually increase the number of servers from 8 to 32 to see how the latency can be reduced by adding servers. We then evaluate the scalability by proportionally adding servers and clients, by keeping a fixed number of clients on one server group. Specifically, we let every 8 servers handle 2^{17} and 2^{16} clients (amortized to about 16K and 8K clients per server, respectively). For different numbers of servers, we choose the best ratio of entry nodes to B-nodes according to the supplementary resource allocation experiment (Appendix E). The ratio is 5:3 for 8 servers and 3:1 for 16, 24, and 32 servers. As reported in Figure 12, the latency reduces from 15.5 seconds to 7.8 seconds when Boomerang+ scales from 8 to 32 servers. Adding more servers to decrease the amortized workload helps re-

duce latency when client size scales, but this also increases the system dollar cost (as discussed in Appendix F). This is a trade-off between performance and cost. For example, with the same total number of clients, supporting an average of 8K clients per server doubles the machine cost of supporting 16K clients per server, while gaining a 2× speed-up. Fortunately, the host cost for a 16-core SGX instance on Azure cloud is only 1121 USD/month, which we believe is affordable when amortized to 8K clients (0.14 USD/month per client).

7 Related Work

Mix-nets and follow-up enhancements. There has been significant progress in metadata-private communication designs recently. One category of results follows the concept of mix networks (mix-nets) [21]. Based on that, recent results [15,16,52–54,56–58,72,87,88] have proposed a number of noteworthy security and privacy enhancement strategies to deal with powerful attackers, including: 1) batch processing of messages from all clients in synchronous rounds to mitigate traffic analysis threats [31,39]; 2) carefully structuring protocols to reveal less observable variables (and thus exposing reduced useful information) to attackers, with representative examples of adopting private virtual addresses for obfuscated message exchanges [54,56,87,88]; 3) generating cover traffic with calibrated parameters to obscure the communication patterns among users (as well as users) [56,87,88]; 4) adding verifiability to the message shuffling to defeat misbehaving servers among the mix-nets [54, 56, 87]; and 5) designing proactive self-recovering schemes (usually relying on the assumption of honest servers) in the face of inadvertent user disconnections or even active network disruptions [54,57]. One latest effort has tried to shift the online burden of clients [11] through oblivious delegation to untrusted proxy servers.

Boomerang's design draws insights from these strategies but differs from them on instantiations with hardware enclaves. Trusting the enclaves enables Boomerang to use fewer servers for traffic mixing with reduced latency. But the unique enclave security context demands Boomerang to bring together tailored oblivious designs for message shuffling, horizontal scaling, and proactive patching against active attacks.

Metadata-private messaging via cryptographic designs. There have been cryptographic designs to facilitate metadata-private communications. One category of results follows the dining cryptographers network (DC-net) [22], which demands broadcasting data linear to the size of the participating clients in the anonymous communication at a high level [28]. Later systems [3, 29, 91] propose scalability improvement and resilience designs against unreliable clients and untrusted servers. Another category of cryptographic designs utilizes private information retrieval [4,6,8], MPC [5] and distributed point function techniques [3,27,37,70] to facilitate oblivious read / write to a database with private mailboxes, based on which metadata-private messaging (and broadcast) system can be constructed. Recently, Clarion [36] gives an MPC-based shuffling design for anonymous communication.

Despite providing cryptographic security (sometimes even under fully untrusted server [4,8]), these systems do not easily scale to more than hundreds of thousands of users while maintaining low latency and high throughput. The inherent cryptographic operations also present unfavorable operational dollar cost, which might yet be attractive for voluntary adoptions in practice.

Security-aware scaling of oblivious data stores. Recent results have studied how to scale the oblivious data access systems without leaking information about the data requests [32,89]. The key is an oblivious load balancer design that distributes access batches independent of the input distribution (with security-aware paddings) and sets the batch size using only public information. The entry-node design in Boomerang+ is inspired by these generic observations. Yet, with context-specific modeling and analysis, we have derived the bound of the maximum batch size that best suits our metadata-private messaging system, through a weighted balls-into-bins game.

Enclave-based network systems. While enclave-based network applications are many, e.g., SGX-Tor [49, 50], SGX-middleboxes [35, 42, 74], to our best knowledge, usage of enclaves is rarely attempted for metadata-private communications, except for one recent work DAEnet [80]. The focus of DAEnet is different from Boomerang, as it tends to hide the conversation route through a peer-to-peer infrastructure, where each peer as a personal computer is assumed to be equipped with an enclave. Unfortunately, this assumption seems no longer in line with the industry movement [44].

8 Conclusion

We have presented Boomerang, a metadata-private messaging system that leverages the readily available trust assumption on hardware enclaves. Boomerang draws many insights from the prior art and achieves efficient pairwise communication with cryptographic security. Its technical instantiation involves tailored oblivious algorithms for message shuffling, horizontal scaling, and proactive resistance against active attacks. We hope Boomerang's comparably high efficiency and low operational cost could make metadata-private messaging systems one step closer to mass adoption in practice.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Dr. Jay Lorch, for their helpful and valuable feedback, and Tencent Yunding lab for providing the Intel SGX cluster and generous technical support. This work was partially supported by the NSFC under Grants U20B2049, U21B2018, 62202228, and 62032021, the HK RGC under Grants N_CityU139/21, RFS2122-1S04, C2004-21GF, R1012-21, and R6021-20F, and the Natural Science Foundation of Jiangsu Province under Grant BK20210330.

References

[1] Confidential Computing Zoo repository. https://github.com/intel/confidential-computing-zoo. Accessed Sept. 2022.

[2] XGBoost repository. https://github.com/mc2-project/secure-xgboost. Accessed Sept. 2022.

[3] Ittai Abraham, Benny Pinkas, and Avishay Yanai. Blinder - scalable, robust anonymous committed broadcast. In *Proc. of ACM CCS*, pages 1233–1252, 2020.

[4] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Addra: Metadata-private voice communication over fully untrusted infrastructure. In *Proc. of USENIX OSDI*, 2021.

[5] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. MCMix: Anonymous messaging via secure multiparty computation. In *Proc. of USENIX Security*, pages 1217–1234, 2017.

[6] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. PIR with compressed queries and amortized query processing. In *Proc. of IEEE S&P*, pages 962–979, 2018.

[7] Sebastian Angel, Sampath Kannan, and Zachary B. Ratliff. Private resource allocators and their applications. In *Proc. of IEEE S&P*, pages 372–391, 2020.

[8] Sebastian Angel and Srinath T. V. Setty. Unobservable communication over fully untrusted infrastructure. In *Proc. of USENIX OSDI*, pages 551–569, 2016.

[9] Apache. Mutual Attestation: Why and How. https://teaclave.apache.org/docs/mutual-attestation/. Accessed Jan. 2023.

[10] Gilad Asharov, T.-H. Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Bucket oblivious sort: An extremely simple oblivious sort. In *Proc. of SOSA*, pages 8–14, 2020.

[11] Ludovic Barman, Moshe Kol, David Lazar, Yossi Gilad, and Nickolai Zeldovich. Groove: Flexible metadata-private messaging. In *Proc. of USENIX OSDI*, pages 735–750, 2022.

[12] Kenneth E. Batcher. Sorting networks and their applications. In *Proc. of AFIPS*, volume 32, pages 307–314, 1968.

[13] Petra Berenbrink, Tom Friedetzky, Zengjian Hu, and Russell A. Martin. On weighted balls-into-bins games. *Theor. Comput. Sci.*, 409(3):511–520, 2008.

[14] Oliver Berthold and Heinrich Langos. Dummy traffic against long term intersection attacks. In *Proc. of PETS*, pages 110–128, 2002.

[15] Stevens Le Blond, David R. Choffnes, William Caldwell, Peter Druschel, and Nicholas Merritt. Herd: A scalable, traffic analysis resistant anonymity network for VoIP systems. In *Proc. of ACM SIGCOMM*, 2015.

[16] Stevens Le Blond, David R. Choffnes, Wenxuan Zhou, Peter Druschel, Hitesh Ballani, and Paul Francis. Towards efficient traffic-analysis resistant anonymity networks. In *Proc. of ACM SIGCOMM*, 2013.

[17] Alexandra Boldyreva, David Cash, Marc Fischlin, and Bogdan Warinschi. Foundations of non-malleable hash and one-way functions. In *Proc. of ASIACRYPT*, volume 5912 of *Lecture Notes in Computer Science*, pages 524–541, 2009.

[18] Thomas Bourgeat, Ilia A. Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. MI6: secure enclaves in a speculative out-of-order processor. In *Proc. of MICRO*, pages 42–56, 2019.

[19] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *Proc. of WOOT*, 2017.

[20] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proc. of USENIX Security*, pages 1041–1056, 2017.

[21] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–88, 1981.

[22] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *J. Cryptol.*, 1(1):65–75, 1988.

[23] Guoxing Chen and Yinqian Zhang. Mage: Mutual attestation for a group of enclaves without trusted third parties. In *Proc. of USENIX Security*, pages 4095–4110, 2022.

[24] Weikeng Chen and Raluca Ada Popa. Metal: A metadata-hiding file-sharing system. In *Proc. of NDSS*, 2020.

[25] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D. Garcia. Voltpillager: Hardware-based fault injection attacks against intel SGX enclaves using the SVID voltage scaling interface. In *Proc. of USENIX Security*, pages 699–716, 2021.

[26] David Core. We kill people based on metadata. *The New York Review*, 2014.

[27] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *Proc. of IEEE S&P*, pages 321–338, 2015.

[28] Henry Corrigan-Gibbs and Bryan Ford. Dissent: accountable anonymous group messaging. In *Proc. of ACM CCS*, pages 340–350, 2010.

[29] Henry Corrigan-Gibbs, David Isaac Wolinsky, and Bryan Ford. Proactively accountable anonymous messaging in verdict. In *Proc. of USENIX Security*, pages 147–162, 2013.

[30] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *Proc. of USENIX Security*, pages 857–874, 2016.

[31] Debajyoti Das, Sebastian Meiser, Esfandiar Mohammadi, and Aniket Kate. Anonymity trilemma: Strong anonymity, low bandwidth overhead, low latency - choose two. In *Proc. of IEEE S&P*, pages 108–126, 2018.

[32] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *Proc. of ACM SOSP*, pages 655–671, 2021.

[33] Roger Dingledine and Nick Mathewson. Anonymity loves company: Usability and the network effect. In *Proc. of WEIS*, 2006.

[34] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In *Proc. of USENIX Security*, pages 303–320, 2004.

[35] Huayi Duan, Cong Wang, Xingliang Yuan, Yajin Zhou, Qian Wang, and Kui Ren. LightBox: Full-stack protected stateful middlebox at lightning speed. In *Proc. of ACM CCS*, pages 2351–2367, 2019.

[36] Saba Eskandarian and Dan Boneh. Clarion: Anonymous communication from multiparty shuffling protocols. In *Proc. of NDSS*, 2022.

[37] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In *Proc. of USENIX Security*, pages 1775–1792, 2021.

[38] Electronic Frontier Foundation. Why metadata matters. `https://ssd.eff.org/module/why-metadata-matters`. Accessed Jan. 2023.

[39] Yossi Gilad. Metadata-private communication for the 99%. *Commun. ACM*, 62(9):86–93, 2019.

[40] Michael T. Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *Proc. of SPAA*, pages 379–388, 2011.

[41] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *Proc. of USENIX ATC*, pages 299–312, 2017.

[42] Juhyeng Han, Seong Min Kim, Jaehyeong Ha, and Dongsu Han. SGX-Box: Enabling visibility on encrypted traffic using a secure middlebox module. In *Proc. of APNet*, 2017.

[43] Alejandro Hevia and Daniele Micciancio. An indistinguishability-based characterization of anonymous channels. In *Proc. of PETS*, volume 5134, pages 24–43, 2008.

[44] Intel. Intel SDP for desktop based on Alder Lake S - 12th Generation Intel Core Processors. `https://edc.intel.com/content/www/us/en/design/ipla/software-development-platforms/client/platforms/alder-lake-desktop/12th-generation-intel-core-processors-datasheet-volume-1-of-2/009/`. Accessed Feb. 2023.

[45] Intel. Intel Xeon scalable platform built for most sensitive workloads. `https://www.intel.com/content/www/us/en/newsroom/news/xeon-scalable-platform-built-sensitive-workloads.html`. Accessed Feb. 2023.

[46] Intel. SGX remote attenstation services. `https://api.trustedservices.intel.com/documents/sgx-attestation-api-spec.pdf`. Accessed Sept. 2022.

[47] Bastien Inzaurralde. The cybersecurity 202: Leak charges against treasury official show encrypted apps only as secure as you make them. *The Washington Post*, 2018.

[48] Van Bulck Jo, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *Proc. of IEEE S&P*, pages 54–72, 2020.

[49] Seongmin Kim, Juhyeng Han, Jaehyeong Ha, Taesoo Kim, and Dongsu Han. Enhancing security and privacy of Tor's ecosystem by using trusted execution environments. In *Proc. of USENIX NSDI*, pages 145–161, 2017.

[50] Seongmin Kim, Juhyeng Han, Jaehyeong Ha, Taesoo Kim, and Dongsu Han. SGX-Tor: A secure and practical Tor anonymity network with SGX enclaves. *IEEE/ACM Transactions on Networking*, 26(5):2174–2187, 2018.

[51] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. Integrating remote attestation with transport layer security. *CoRR*, 2018.

[52] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. Atom: Horizontally scaling strong anonymity. In *Proc. of ACM SOSP*, pages 406–422, 2017.

[53] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. In *Proc. of PETS*, pages 115–134, 2016.

[54] Albert Kwon, David Lu, and Srinivas Devadas. XRD: Scalable messaging system with cryptographic privacy. In *Proc. of USENIX NSDI*, pages 759–776, 2020.

[55] Andrew Law, Chester Leung, Rishabh Poddar, Raluca Ada Popa, Chenyu Shi, Octavian Sima, Chaofan Yu, Xingmeng Zhang, and Wenting Zheng. Secure collaborative training and inference for XGBoost. In *Proc. of PPMLP*, pages 21–26, 2020.

[56] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *Proc. of USENIX OSDI*, pages 711–725, 2018.

[57] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Yodel: Strong metadata security for voice calls. In *Proc. of ACM SOSP*, pages 211–224, 2019.

[58] David Lazar and Nickolai Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *Proc. of USENIX OSDI*, pages 571–586, 2016.

[59] Alleyne Llanor. Enterprise end-to-end encryption is on the rise. *IT Business Edge*, 2021.

[60] Nick Mathewson and Roger Dingledine. Practical traffic analysis: Extending and resisting statistical disclosure. In *Proc. of PETS*, pages 17–34, 2004.

[61] Jonathan R. Mayer, Patrick Mutchler, and John C. Mitchell. Evaluating the privacy properties of telephone metadata. *Proc. Natl. Acad. Sci. USA*, 113(20):5536–5541, 2016.

[62] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proc. of HASP*, page 10, 2013.

[63] Microsoft. Azure bandwidth pricing. https://azure.microsoft.com/en-us/pricing/details/bandwidth/. Accessed Sept. 2022.

[64] Microsoft. Azure cloud messaging services. https://azure.microsoft.com/en-us/solutions/messaging-services/#overview. Accessed Sept. 2022.

[65] Microsoft. Azure virtual machine pricing. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/#pricing. Accessed Sept. 2022.

[66] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *Proc. of IEEE S&P*, pages 279–296, 2018.

[67] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In *Proc. of CHES*, pages 69–90, 2017.

[68] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. In *Proc. of IEEE S&P*, pages 1466–1482, 2020.

[69] Netsfere. Netsfere pricing. https://www.netsfere.com/Product/Free-Pro-Custom-Enterprise-Messaging-Pricing. Accessed Sept. 2022.

[70] Zachary Newman, Sacha Servan-Schreiber, and Srinivas Devadas. Spectrum: High-bandwidth anonymous broadcast. In *Proc. of USENIX NSDI*, pages 229–248, 2022.

[71] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *Proc. of USENIX Security*, pages 619–636, 2016.

[72] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The Loopix anonymity system. In *Proc. of USENIX Security*, pages 1199–1216, 2017.

[73] Rishabh Poddar, Ganesh Ananthanarayanan, Srinath Setty, Stavros Volos, and Raluca Ada Popa. Visor: Privacy-preserving video analytics as a cloud service. In *Proc. of USENIX Security*, pages 1039–1056, 2020.

[74] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. SafeBricks: Shielding network functions in the cloud. In *Proc. of USENIX NSDI*, pages 201–216, 2018.

[75] Christian Priebe, Kapil Vaswani, and Manuel Costa. EnclaveDB: A secure database using SGX. In *Proc. of IEEE S&P*, pages 264–278, 2018.

[76] Martin Raab and Angelika Steger. "balls into bins" - A simple and tight analysis. In *Proc. of International Workshop on Randomization and Approximation Techniques in Computer Science*, volume 1518, pages 159–170, 1998.

[77] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *Proc. of IEEE S&P*, pages 1852–1867, 2021.

[78] Mark Russinovich, Manuel Costa, Cédric Fournet, David Chisnall, Antoine Delignat-Lavaud, Sylvan Clebsch, Kapil Vaswani, and Vikas Bhatia. Toward confidential cloud computing. *Commun. ACM*, 64(6):54–61, 2021.

[79] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *Proc. of IEEE S&P*, pages 38–54, 2015.

[80] Tianxiang Shen, Jianyu Jiang, Yunpeng Jiang, Xusheng Chen, Ji Qi, Shixiong Zhao, Fengwei Zhang, Xiapu Luo, and Heming Cui. DAENet: Making strong anonymity scale in a fully decentralized network. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2286–2303, 2021.

[81] Jatinder Singh, Jennifer Cobbe, Do Le Quoc, and Zahra Tarkhani. Enclaves in the clouds. *Commun. ACM*, 64(5):42–51, 2021.

[82] Emil Stefanov and Elaine Shi. ObliviStore: High performance oblivious cloud storage. In *Proc. of IEEE S&P*, pages 253–267, 2013.

[83] Tencent. Tencent Cloud instance type documentation. https://intl.cloud.tencent.com/document/product/213/11518. Accessed Sept. 2022.

[84] Tencent. Tencent cloud instant messaging pricing. https://intl.cloud.tencent.com/products/im. Accessed Sept. 2022.

[85] Tencent. Tencent Cloud virtual machine pricing. https://intl.cloud.tencent.com/pricing/cvm/overview. Accessed Sept. 2022.

[86] Trillian. Trillian instant messaging pricing. https://trillian.im/pricing/. Accessed Sept. 2022.

[87] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. In *Proc. of ACM SOSP*, pages 423–440, 2017.

[88] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proc. of ACM SOSP*, pages 137–152, 2015.

[89] Midhul Vuppalapati, Kushal Babel, Anurag Khandelwal, and Rachit Agarwal. SHORTSTACK: Distributed, fault-tolerant, oblivious data access. In *Proc. of USENIX OSDI*, pages 719–734, 2022.

[90] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proc. of ACM CCS*, pages 2421–2434, 2017.

[91] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In *Proc. of USENIX OSDI*, pages 179–182, 2012.

[92] David Isaac Wolinsky, Ewa Syta, and Bryan Ford. Hang with your buddies to resist intersection attacks. In *Proc. of ACM CCS*, pages 1153–1166, 2013.

A Discussion and Limitations

Establishing trust on Boomerang. Boomerang relies on the trust assumption on secure enclaves, which provide confidentiality and integrity for data and codes. While trusting a single enclave node can be done through the standard remote attestation, like the one from Intel SGX [46], trusting multiple-enclave applications would additionally involve mutual attestation, which is necessary among interacting enclaves to establish a trust relationship [23,79]. One common practice is to rely on a trusted third party (TTP) to facilitate mutual attestation [9]. The TTP can perform remote attestation with each enclave individually and serve as a trusted anchor to bootstrap the mutual trust among those enclaves. In the case of Boomerang, we suggest following the above common practice for any client connecting to the system to establish trust on Boomerang. The TTP that all clients need to rely on can be the trusted developer of Boomerang or his/her delegated server in a trusted domain [9, 23]. A very recent work has proposed a new way for mutual attestation among enclaves without relying on a TTP [23], which can be beneficial to the trust establishment in the Boomerang system. In the future, to mitigate the concern on the centralized trust of a single enclave vendor (e.g., Intel), we can further consider employing a mix of enclaves from different vendors, e.g., Intel SGX, ARM

TrustZone, AMD SEV, etc., to distribute the trust, which is also a trendy subject in recent literature [23].

Reduce client online burden. Metadata-private messaging systems usually assume clients should always keep online and send messages at regular rates to hide the real communication behavior [4,33,39,54,57,87,88]. To simplify the problem statement, Boomerang's security analysis also follows this assumption. Although Boomerang achieves acceptable bandwidth cost, we acknowledge that this "always-online" requirement may be a barrier to the practical use of Boomerang. The latest work Groove [11] studied this issue and proposed an oblivious delegation mechanism to reduce client online burden, by introducing proxies between clients and mixnets. To improve Boomerang's client flexibility, a feasible way is to integrate the oblivious delegation mechanism and the proxy design with Boomerang, considering that Boomerang can be an alternative to mixnets for message shuffling. We believe our Boomerang can serve as a performant and secure backend for metadata-private message shuffling.

B Balls into Bins

To analyze the overflow probability in Boomerang+, we introduce the balls-into-bins game to estimate the probability that a message may be dropped during batch distributions.

Balls-into-bins studies the allocation problem that throws m balls into n bins by placing each ball into a bin chosen independently and uniformly at random [76]. One natural question in this area is to ask for the maximum number of balls in any bin. According to prior art [13,76], an interesting result about the maximum number of balls in any bin problem is introduced below, which will be used to estimate the maximum load bound of Boomerang+.

Lemma B.1 (Maximum load [76]) *Let ℓ be the random variable that counts the maximum number of balls in any bin, if we throw m balls independently and uniformly at random into n bins and $m \gg n(\ln n)^3$. Then we have*

$$\Pr[\ell > \frac{m}{n} + \sqrt{\frac{2m\ln n}{n}\left(1-\frac{1}{\alpha}\frac{\ln\ln n}{2\ln n}\right)}] = 1-\frac{1}{n^{\alpha}},$$

where α is a positive constant larger than 1.

Here α serves a role to ensure the tightness of the bound of the maximum load [13,76]. Next, we prove Theorem 4.1 below.

Theorem 4.1 *For any set of m messages, n Boomerang nodes, and a security parameter λ, satisfying $m \gg n(\ln n)^3$ and $\lambda/\log_2 n > 1$, let $B(m,n)$ be a function that outputs the maximum batch size B for each node in Boomerang+. Then the probability of overflow is negligible in λ if we choose*

$$B = \left\lceil \frac{m}{n} + 4\sqrt{\frac{m\ln n}{3n}\left(1-\frac{1}{\lambda}\frac{\ln\ln n}{2\ln 2}\right)} \right\rceil.$$

Proof. Like most existing works, allocating a set of m messages to a given number of servers can be formalized as a balls-into-bins game. But a bit different from their works, the balls are weighted in Boomerang+ because messages with the same private labels will be allocated to the same servers. Suppose m_1 is the number of messages with single-pattern private labels (see §3.3.2 for possible causes), and m_2 is the number of messages with double-pattern private labels (i.e., regular messages from communicating pairs). Since assigning messages with single-pattern private labels and assigning messages with regular private labels are independent, the task of allocating m messages can be separated as two subtasks: 1) allocating m_1 messages with single-pattern private labels; and 2) allocating m_2 messages with double-pattern private labels. It's clear that $m_1 + m_2 = m$.

For the task of allocating m_1 messages to n B-nodes, it can be formalized as the game throwing m_1 balls into n bins. For the task of allocating m_2 messages, a pair of messages with the same regular private labels will be allocated to the same B-node. Thus, we can tie m_2 balls together in pairs (by their double-pattern labels), and throw $m_2/2$ times. Namely, it is a $m_2/2$-balls-into-n-bins problem. Let ℓ_1 and ℓ_2 be two random variables that count the maximum messages assigned in any B-node in the above two tasks, respectively. Let

$$B_{m_1} = \frac{m_1}{n} + \sqrt{\frac{2m_1\ln n}{n}(1-\frac{1}{\alpha}\frac{\ln\ln n}{2\ln n})}$$

and

$$B_{m_2} = \frac{m_2}{n} + 2\sqrt{\frac{m_2\ln n}{n}(1-\frac{1}{\alpha}\frac{\ln\ln n}{2\ln n})}.$$

According to Lemma B.1, we have

$$\Pr[\ell_1 > B_{m_1}] = 1-\frac{1}{n^{\alpha}} \text{ and } \Pr[\ell_2 > B_{m_2}] = 1-\frac{1}{n^{\alpha}}.$$

To prevent overflow, the probability of a message being dropped should be confined to a negligible function in the security parameter λ. Thus we have

$$1-\frac{1}{n^{\alpha}} = 1-\frac{1}{2^{\lambda}} \Rightarrow \alpha = \frac{\lambda}{\log_2 n}.$$

Let $B' = B_{m_1} + B_{m_2}$, then

$$B' = \frac{m}{n} + \left(\sqrt{m_1}+\sqrt{2m_2}\right)\sqrt{\frac{2\ln n}{n}\left(1-\frac{1}{\lambda}\frac{\ln\ln n}{2\ln 2}\right)}. \quad (2)$$

With the arithmetic-geometric mean inequality, we have that

$$\left(\sqrt{m_1}+\sqrt{2m_2}\right) \leq \sqrt{2(\sqrt{m_1})^2+2(\sqrt{2m_2})^2},$$

which attains its equality if and only if $\sqrt{m_1} = \sqrt{2m_2}$.

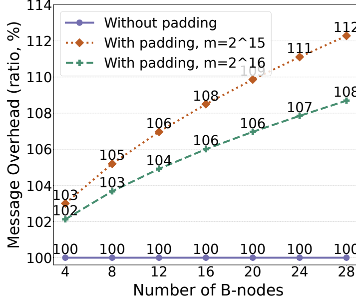


Figure 13: Message overhead (ratio), which describes the ratio of overall padded messages to real messages, i.e., nB/m.

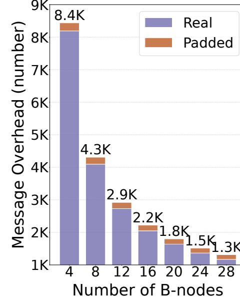


Figure 14: Message overhead (number) under 2^{15} real messages concretely.

Combined with the condition $m_1 + m_2 = m$, we can solve for $m_1 = 2m/3$ and $m_2 = m/3$. Then

$$(\sqrt{m_1} + 2\sqrt{m_2}) \leq \sqrt{2\left[\left(\sqrt{\frac{2m}{3}}\right)^2 + \left(\sqrt{\frac{2m}{3}}\right)^2\right]} \leq 2\sqrt{\frac{2m}{3}}. \tag{3}$$

Applying Eq. (3) to Eq. (2), it is easy to get

$$B' \leq \frac{m}{n} + 4\sqrt{\frac{m\ln n}{3n}\left(1 - \frac{1}{\lambda}\frac{\ln\ln n}{2\ln 2}\right)} \leq B.$$

Finally, we compute the probability that a message is dropped on a server. It equals the probability that the maximum number of messages allocated to a node is larger than the maximum load. As mentioned, the above two subtasks are independent, and thus we have

$$\begin{aligned}
\Pr[\ell > B] &\leq \Pr[\ell > B'] = \Pr[\ell > B_{m_1} + B_{m_2}] \\
&\leq 1 - \Pr[\ell \leq B_{m_1} + B_{m_2}] \\
&\leq 1 - \Pr[\ell_1 \leq B_{m_1} \wedge \ell_2 \leq B_{m_2}] \\
&\leq 1 - (1 - \Pr[\ell_1 > B_{m_1}])(1 - \Pr[\ell_2 > B_{m_2}]) \\
&\leq \frac{2}{n^\alpha} - \frac{1}{n^{2\alpha}} = \frac{2}{2^\lambda} - \frac{1}{2^{2\lambda}}.
\end{aligned}$$

That is to say, the probability that a message is dropped (aka overflow) is negligible in λ. This completes the proof.

In practice, we usually round the above bound up to an integer that is greater but nearest to it. Figures 13 and 14 have demonstrated that our maximum batch size only incurs marginal overhead (with extra paddings) on our horizontal scaling design.

Scalability Analysis. We borrow the idea from XRD [54] to define the scalability of the designed system. Specifically, we say that a system is scalable if the number of requests that one server needs to handle trends to zero when the number of deployed servers increases to infinite. Without loss of generality, we start with one entry node that obliviously distributes the incoming messages to a set of B-nodes in Boomerang+. Let m and n denote the number of messages and deployed B-nodes, respectively. According to Theorem 4.1, we know that the upper bound of the number of messages sent to a B-node server is $\frac{m}{n} + 4\sqrt{\frac{m\ln n}{3n}(1 - \frac{1}{\lambda}\frac{\ln\ln n}{2\ln 2})}$. It is clear that

$$\lim_{n\to\infty} \frac{m}{n} + 4\sqrt{\frac{m\ln n}{3n}(1 - \frac{1}{\lambda}\frac{\ln\ln n}{2\ln 2})} = 0.$$

Now let us consider having more entry nodes in the system. It is intuitive to see that in Boomerang+ each entry node can run independently and in parallel, which effectively eliminates a potential bottleneck at a single entry node. Similar to the observations in Snoopy [32], in Boomerang+, adding more entry-nodes is not entirely free, because it would increase the total amount of messages sent to each B-node server. Suppose we have ν entry-nodes, n B-nodes, and each entry-node has m incoming messages (out of the total $\nu \times m$ messages) for oblivious distribution. The upper bound of the total amount of messages from ν entry-nodes sent to a B-node server is $\nu \times (\frac{m}{n} + 4\sqrt{\frac{m\ln n}{3n}(1 - \frac{1}{\lambda}\frac{\ln\ln n}{2\ln 2})})$, which would still approach 0 when $n \to \infty$. Thus, Boomerang+ can scale with more clients and messages by adding more entry nodes and more B-nodes.

Our analysis mainly focuses on the heavy load cases, where $m \gg n(\ln n)^3$ generally holds. For example, for those deployments in practice with good anonymity, m at each entry node can easily reach at least in the order of 10^5, while the overall workload can largely be handled with no more than n = 100 B-nodes. If in the extreme cases where m becomes small, we have some fallback options: 1) setting the maximum load $B = m$; or 2) adaptively falling back to single-server Boomerang mode. We leave this adaptive switching design as our future work.

Note that our scalability analysis does not give specific configuration guidelines on how to add entry-nodes and B-nodes, because this would be highly dependent on specific requirements on performance, e.g., latency, throughput, etc., and cost, e.g., expenses of adding respective nodes, bandwidth, etc. We would resort to the abstract configuration planner in Snoopy [32] as a good starting point when we push Boomerang+ to a more practical realm in the future.

C Security Analysis

C.1 Proof of Theorem 5.1

As mentioned, due to the deployment of secure enclaves and a round-based communication model in Boomerang, the only thing left is to prove that memory access patterns in the obfuscated message exchange are oblivious. From the description in §3, Boomerang is designed by combining several oblivious algorithms. Here we prove the indistinguishability of memory access patterns in Boomerang in a modular way. First, we demonstrate that our proactive pattern patching (denoted

```
Real_PPP(pkts) : // pkts: the real input packets prepared by clients
  Parse pkts as (pkt_1,...,pkt_m), where m is the number of packets
  is_mtt = False
  for pkt in pkts do
    is_prev_same = O_Equal(pkt.priv_label, Prev(pkt).priv_label)          ▷ Step 3.1: Detect irregular pattern
    is_next_same = O_Equal(pkt.priv_label, Next(pkt).priv_label)
    is_next2_same = O_Equal(pkt.priv_label, Next(Next(pkt)).priv_label)
    is_mtt = is_mtt and is_prev_same or is_next2_same                     ▷ Detect more-than-two pattern
    pkt.R = O_Choose(is_next_same, Next(pkt).S, pkt.S)                    ▷ Step 3.2: Swapping and patching
    pkt.R = O_Choose(is_prev_same, Prev(pkt).S, pkt.R)
    pkt.R = O_Choose(is_mtt, pkt.S,pkt.R)
  end for
Ideal_PPP(pkts') : //pkts': the dummy input generated using public information on the number of packets and the packet size
  Parse pkts' as (pkt_1,...,pkt_m), where m is the number of packets
  is_mtt = False
  for pkt in pkts' do
    is_prev_same = Sim_O_Equal(pkt.priv_label, Prev(pkt).priv_label)      ▷ Step 3.1: Detect irregular pattern
    is_next_same = Sim_O_Equal(pkt.priv_label, Next(pkt).priv_label)
    is_next2_same = Sim_O_Equal(pkt.priv_label, Next(Next(pkt)).priv_label)
    is_mtt = is_mtt and is_prev_same or is_next2_same                     ▷ Detect more-than-two pattern
    pkt.R = Sim_O_Choose(Next(pkt).S, pkt.S)                              ▷ Step 3.2: Swapping and patching
    pkt.R = Sim_O_Choose(Prev(pkt).S, pkt.R)
    pkt.R = Sim_O_Choose(pkt.S, pkt.R)
  end for
```

Figure 15: Real and ideal experiments for an oblivious proactive pattern patching algorithm

as PPP) algorithm built on existing oblivious algorithms is oblivious according to Definition C.1 below. Then we prove that our Boomerang system protects metadata privacy when built on the oblivious PPP. We first give the definition of a secure PPP algorithm.

Definition C.1 *The proactive pattern patching algorithm* PPP *is secure if for any PPT attacker, there exists a PPT simulator such that*

$$|\Pr[\mathrm{Real}_{\mathrm{PPP}}(\lambda)=1]-\Pr[\mathrm{Ideal}_{\mathrm{PPP}}(\lambda)=1]|\leq \mathrm{negl}(\lambda), \quad (4)$$

where λ *is the security parameter,* $\mathrm{Real}_{\mathrm{PPP}}$ *and* $\mathrm{Ideal}_{\mathrm{PPP}}$ *are experiments defined in Figure 15.*

Below we show that our PPP algorithm satisfies the above definition by proving the $\mathrm{Real}_{\mathrm{PPP}}$ and $\mathrm{Ideal}_{\mathrm{PPP}}$ experiments are indistinguishable.

Lemma C.1 *Given the oblivious primitives for comparison and assignments* O_Choose *and* O_Equal*, the proactive pattern patching protocol described in §3 and formally defined in Figure 15 is also an oblivious algorithm.*

Proof. To simplify the proof and our description of the simulator, we assume that the packets (aka encrypted messages) received by the server are indistinguishable by size and traffic patterns, which the attacker cannot exploit to distinguish the memory access patterns. Then we need to demonstrate that the memory accesses of the simulated experiment $\mathrm{Ideal}_{\mathrm{PPP}}$ that takes public information as input are indistinguishable from those of the real experiment $\mathrm{Real}_{\mathrm{PPP}}$. As shown in Figure 15, we leverage the following oblivious building blocks in the $\mathrm{Real}_{\mathrm{PPP}}$ experiment.

- O_Choose(cond, a, b): If cond = True, it outputs value a. Otherwise, it outputs value b.
- O_Equal(a, b): Obliviously assigns True to the output if a equals b. Otherwise, it outputs False.

The simulated experiment $\mathrm{Ideal}_{\mathrm{PPP}}$ is built on top of simulations of the above oblivious building blocks.

- Sim_O_Choose(a, b): Simulates choosing from (a, b) as the output, given a hidden bit.
- Sim_O_Equal(a, b): Simulates testing whether a equals b and outputs True if they are equal.

With these building blocks, we show that the memory access patterns in the real and ideal experiments defined in Figure 15 are indistinguishable. Specifically, if an attacker can distinguish the $\mathrm{Ideal}_{\mathrm{PPP}}$ and $\mathrm{Real}_{\mathrm{PPP}}$, then the distinguishability must occur in at least one of the steps. But this happens only with negligible probability because: 1) the simulator uses the public information to simulate indistinguishable

dummy input from real input (i.e., the number of packets m and the packet size), and accordingly follows the same `for` loop structure as the $\mathsf{Real}_{\mathsf{PPP}}$; 2) for each iteration inside the `for` loop structure, the security of oblivious building blocks `O_Choose` and `O_Equal` ensures that the corresponding simulations `Sim_O_Choose` and `Sim_O_Equal` produce indistinguishable memory access patterns; and 3) the operations on direct assignment to `is_mtt` in both $\mathsf{Ideal}_{\mathsf{PPP}}$ and $\mathsf{Real}_{\mathsf{PPP}}$ are also indistinguishable. This completes the proof of Lemma C.1.

Proof of Theorem 5.1. Based on the fact that the proactive pattern patching (PPP) algorithm is oblivious, we continue to prove Theorem 5.1 under the security notion defined in Definition 2.1. Let b and b' be the choices of the challenger and the attacker $\mathcal{A}$, respectively, in the experiment EXP defined in Definition 2.1.

From the attacker's view, as all messages are encrypted, the only way to distinguish the communication patterns of two given clients is by observing memory access patterns during the "obfuscated" message exchange procedure. We assume all involved oblivious algorithms are computationally indistinguishable, except with negligible probability in λ (aka $\text{negl}(\lambda)$). As seen, Boomerang leverages an oblivious proactive pattern patching algorithm (as shown in Lemma C.1) and an oblivious sorting algorithm [10]. Let "M fails" denote the event that the memory access patterns of at least one of the algorithms mentioned above fail to achieve obliviousness, which happens only with negligible probability in λ. Thus, we have

$$
\begin{aligned}
\mathsf{Adv}_{\mathsf{EXP},\mathcal{A}} &= \left|\Pr[b=b'] - \Pr[b \neq b']\right| \\
&\leq \max\{\Pr[b=b', \texttt{M}\text{ fails}], \Pr[b \neq b', \texttt{M}\text{ fails}]\} \\
&= \max\{\Pr[b=b'|\texttt{M}\text{ fails}], \Pr[b \neq b'|\texttt{M}\text{ fails}]\} \cdot \Pr[\texttt{M}\text{ fails}] \\
&\leq \Pr[\texttt{M}\text{ fails}] = \text{negl}(\lambda).
\end{aligned}
$$

The above shows that the attacker cannot identify whether two clients are communicating or not within each round, except with negligible probability. For the fixed system configuration across rounds, it is easy to see that the attacker's view will remain the same across rounds. This completes the proof of Theorem 5.1.

C.2 Proof of Theorem 5.2

The proof of Theorem 5.2 is analogous to that of Theorem 5.1. The only difference between Boomerang and Boomerang+ is that Boomerang+ employs entry nodes as load balancers to distribute batches of messages from all clients to a group of B-nodes for message exchange. Thus, the key is to show that this distribution procedure is oblivious.

We assume that the system configuration is fixed across rounds, including the number of entry nodes, B-nodes, and connected clients to each entry node. First, we show that Boomerang+ assigns a message to each B-node uniformly. Note that Boomerang+ assigns a message to each B-node by computing $\texttt{br_id} = \mathsf{H}_k(\texttt{priv_label}||\texttt{round_num})\%n$, where H is a keyed cryptographic hash function, and n refers to the number of B-nodes. According to the classical simulation-based security definition of a keyed hash function [17], the result of a hash function and a random value is computationally indistinguishable. It implies that the distribution of `br_id` is uniform, and further confirms that allocating messages to different B-nodes can indeed be formulated as a random balls-to-bins assignment. Therefore, we can apply the batch size derived from Theorem 4.1 to set up the batch structure without overflow. Moreover, the batch size is determined by the public information (as shown in §4.2.3) only, independent of the input.

Based on the above initial result, it follows that the probability that a message is assigned to any individual B-node is always equal. Without loss of generality, we assume that the number of deployed entry nodes is ν. Let c be an encrypted message sent by a client and e be its refreshed copy by the entry node. The probability for the message assigned to the t-th B-node is

$$
\Pr[c \to \mathbb{B}_t] = \sum_{j=1}^{\nu} \Pr[c \to \mathbb{E}_j] \cdot \Pr[e \to \mathbb{B}_t] = \frac{1}{n},
$$

where $\mathbb{B}_t$ denotes the t-th B-node, $\mathbb{E}_j$ denotes the j-th entry node, and $c \to \mathbb{B}_t$ denotes that message c is assigned to $\mathbb{B}_t$ finally. This result holds as long as no empty B-node exists during one communication round, which is guaranteed by our uniform message assignment and oblivious sub-batch padding algorithms. Our oblivious sub-batch padding algorithm is largely based on existing building blocks, including the oblivious padding algorithm in Snoopy [32]. Thus, we omit the proofs for its obliviousness here. With the above results, we can obtain that messages from any two clients i and j, whether they are communicating or not, are assigned to the same B-node t with the same probability $1/n^2$. In other words, whether the two clients are communicating or not, their message assignment from entry node(s) to a single B-node is indistinguishable from the attacker.

Now let's focus on messages assigned to each individual B-node. According to Theorem 5.1, an attacker cannot identify whether any pair of clients are communicating or not at a single-server Boomerang. Thus, it follows that at each individual B-node in Boomerang+, an attacker cannot identify whether any pair of messages are from two communicating clients or not. It holds in any single round. As the assignment mapping (through the keyed hash function H) is refreshed for every round, it is easy to see that the attacker's view will remain the same across rounds, with a fixed system configuration. It ensures the indistinguishability of whether two clients communicate or not in Boomerang+.

Finally, we need to prove that the memory access patterns in Boomerang+ are oblivious. To achieve such obliviousness, we build Boomerang+ on top of a group of oblivious primitives

Table 1: Latency with varying numbers of entry nodes (#E) and B-nodes (#B). The best ratios with the lowest latency are marked in bold.

	#E	#B	Latency	#E	#B	Latency	#E	#B	Latency	#E	#B	Latency
8 Servers	7	1	20.10	6	2	16.08	**5**	**3**	**15.54**	4	4	16.19
	3	5	20.57	2	6	28.71	1	7	58.47			
16 Servers	14	2	12.19	**12**	**4**	**10.37**	10	6	10.45	8	8	10.71
	6	10	11.65	4	12	17.16	2	14	27.54			
24 Servers	22	2	12.62	20	4	10.18	**18**	**6**	**8.09**	16	8	9.59
	14	10	8.41	12	12	9.11	10	14	9.46	8	16	10.71
	6	18	12.12	4	20	14.95	2	22	27.81			
32 Servers	30	2	11.77	28	4	9.35	26	6	8.45	**24**	**8**	**7.76**
	22	10	8.26	20	12	8.41	18	14	8.78	16	16	8.23
	14	18	8.75	12	20	8.99	10	22	9.27	8	24	9.83
	6	26	12.47	4	28	16.49	2	30	28.46			

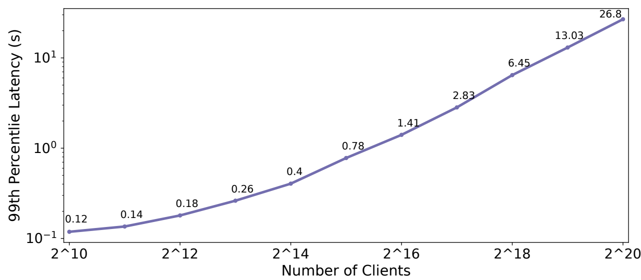


Figure 16: 99th percentile latency of Boomerang.

including oblivious comparisons, assignments, compaction, and sorting functions. Based on these oblivious primitives, we develop the oblivious batch generation and distribution procedures, and those based on the oblivious PPP algorithm with slight modifications in Boomerang+, just like the way we develop Boomerang. Thus, we do not spend more space repeating that their algorithms are oblivious. By the above two derivations, we show that Boomerang+ is secure according to Definition 2.1. This completes the proof.

D Boomerang Performance

Latency. We evaluate Boomerang on one 16-core server and test the latency over up to 2^{20} clients. Figure 16 shows the 99th percentile latency of Boomerang with a varying number of clients. For 2^{15} clients, the latency is 778 ms, which is also enough for VoIP communication. Notably, Boomerang achieves 1.41 second latency for 2^{16} clients using only one server. We can observe that the latency increases (almost) linearly with the number of clients (messages). There are mainly two factors: 1) with the increasing of clients, the server needs to handle more RPC requests; and 2) the most expensive computation in enclaves is oblivious sort (twice), which is of $O(n(\log n)^2)$ complexity. When the client number increases to 2^{20}, the latency reaches 26.8 seconds, which is no longer suitable for latency-sensitive applications. The increased latency also shows the need for horizontal scalability.

Bandwidth cost. Boomerang requires each client to send a message of 256 Bytes every round. It occupies 512 Bps bandwidth for each client if we set a 500 ms round. If the client keeps online on Boomerang for one month, the bandwidth cost is 2.47 GB (including sending and receiving data), which we believe is affordable for ordinary users.

E Supplementary Experiments for Resource Allocation

Recall that in §6.3 we show the best resource allocation for entry nodes and B-nodes with 16 servers that can achieve the lowest latency. This section further reports the best allocation with 8, 16, 24, and 32 servers. We have exhaustively tried different combinations of entry nodes and B-nodes for each set of servers and let them handle 2^{20} messages. The results are reported in Table 1. The best allocation for entry nodes and B-nodes for 8, 16, 24, and 32 servers are 5:3, 12:4, 18:6, and 24:8, respectively. Note that this is a brute-force way to get the best ratio. A possibly more efficient way to configure the system is to design a configuration planner [32], which we will leave as our future work.

F System Dollar Cost

We here report the estimated cost in US dollars of running Boomerang+ servers. Leaving human-operation costs aside, we calculate the total cost for hosting Boomerang+ servers and data egress costs and show how much each user would (at least) pay for joining Boomerang+ for one month.

Server cost. Since the Tencent Cloud machines we use did not announce the international pricing for M6ce.4XLARGE124 [85], we alternatively choose Standard_Dc16s_v3 instances from Azure Cloud for price estimation, which have the same properties as M6ce.4XLARGE128. According to the prices for Azure Cloud VMs [65], the host (machine) cost is 1121 USD/month. The cost varies with different performance goals. For example, running 16 instances for 2^{16} clients achieves 615 ms latency, which results in $0.274 amortized monthly cost per client. In the scaling experiment, Boomerang+ runs 32 8-core instances for 2^{20} clients, resulting in $0.017 amortized cost and 7.76 second latency based on the price for the alternative instance Standard_Dc8s_v3 (560 USD/month). Ideally, adding more servers will further reduce the latency, but it will also increase the overall server cost.

Bandwidth cost. We assume that each client sends a 256 Byte message every 500 ms, adding up to 1.24 GB data ingress to (or egress from) the server if the client stays online for one month. To save data transfer costs, we can set the servers in the same availability zone, within which the transferred data is free of charge. If the clients and servers transfer data between different continents, the price is at most 0.16 USD/GB [63]. Then, the server-side bandwidth cost is 0.198 USD/month for each client. Overall, we believe Boomerang+ is affordable for clients while maintaining good privacy and high-performance services. Its promising cost is comparable to non-private cloud-based IM services today [64, 69, 84, 86].