Predictive Eviction: A Novel Policy for Optimizing TLS Session Cache Performance

Ryan Stevens and Hao Chen
University of California, Davis
{rcstevens, chen}@ucdavis.edu

***Abstract*—Transport Layer Security (TLS) is the most commonly used security protocol to encrypt web traffic. TLS connections are computationally expensive to set up, so the TLS protocol supports session resumption, where previously negotiated connection parameters can be used to short-circuit the TLS handshake. The server assigns new sessions a session identifier (ID) and caches each session by its ID so it can be retrieved later. As clients come and go, sessions in the server's cache will have to be evicted according to the server's eviction policy. We find that first-in-first-out (FIFO) and least-recently-used (LRU) are the most common session cache eviction policies among popular TLS libraries, however, for applications whose clients connect at regular intervals, such as mobile advertising, the performance of these policies may be far worse than randomly evicting policies from the cache. To handle this, we propose a novel eviction policy for TLS session caches, *predictive eviction*, that relies on the server knowing the next time each client will connect again. Using a real-world application of such a policy, Android in-application advertising, we build a client that is able to simulate the behavior of a large number of devices requesting mobile advertisements over TLS. We use this simulated client to benchmark the hit rate of the predictive policy compared with eviction policies found in popular TLS library implementations. In addition, we demonstrate that our policy can be implemented efficiently by benchmarking its performance in transactions per second compared with OpenSSL's session cache implementation, and compared with TLS session tickets (an alternative to session caching for resuming TLS sessions). We find that our policy has better hit rate performance than other eviction policies, and can achieve comparable performance to session tickets. To the best of our knowledge, this is the first study of the performance of TLS session resumption strategies.**

I. Introduction

TLS (and its predecessor SSL) is a collection of cryptographic protocols that are designed to support confidentiality, integrity, and authenticity of network connections. It is the *de facto* standard for securing Internet communications such as web browsing (HTTPS) and email (SMTP over TLS). As TLS connections are computationally expensive to set up, because they require public key cryptography, TLS supports session resumption, where parameters of a previous connection can be reused to avoid the cost of establishing a new connection with a returning client. If the client and server both have stored session information from a previous connection, they can resume the previous session without public key cryptography and using fewer messages than a full handshake, making resuming sessions much cheaper than creating new ones. To resume sessions, clients and servers must store recently negotiated sessions. Whereas clients only need to store a handful of sessions, one for each TLS server with which the client has recently communicated, servers may need to store a large number of sessions if they have many active clients, but the memory required to store these sessions may exceed the server's capacity or degrade the server's performance. Moreover, sessions become stale over time as clients terminate and purge their session information, either because sessions are held in memory or because it would be insecure to continue using session information for long periods of time. To handle these problems, a TLS server provides session caches to store recently negotiated sessions and evicts sessions when the cache overflows.

To decide which session to evict, the server uses an *eviction policy*. The simplest policy is random eviction. To improve the cache hit rate, more elaborate policies are used — e.g., first-in-first-out (FIFO) and least-recently-used (LRU) — where the server tries to speculate which session is the least likely to be used in the near future based on past observations. These policies have been shown to improve the cache hit rate over the random policy and are used in lieu of the random policy in most cases.

However, the popularity of web and mobile ads creates a type of traffic that both FIFO and LRU may handle poorly. Ad traffic has the following properties:

- Ad requests are often sent at constant intervals (when ads are refreshed).
- Ad clients (the browser or mobile app) often terminate or become inactivate without notifying the ad server.

To understand why both FIFO and LRU may handle such traffic poorly, consider a TLS server with a session cache size of two that has three clients which connect at the same regular interval. If the server evicts sessions with FIFO or LRU policy, every request will result in a cache miss (we can generalize this observation to a cache of any fixed size n when there are greater than n clients). On the other hand, if the server uses a random eviction policy, the cache hit rate would be 33%. Finally, the optimal eviction policy would keep the first two sessions in the cache even when the third client connects, resulting in a cache hit rate of 66%. Since ad servers must

be able to handle a large number of clients concurrently and swiftly, they must keep their cache miss rates low. Clearly, neither FIFO nor LRU can reliably achieve this.

A. Predictive Cache Eviction Policy

We propose a novel eviction policy for TLS session caches, *predictive eviction*, when the server knows when each client will connect again. This policy can be applied in two cases: 1) when clients connect periodically, such web and mobile ads, allowing the server to know the next connection time without any information from the clients, or 2) when clients can predict their next connection time and must explicitly send this information to the server. In our evaluation, we use approach (2), and include the next connection time as an HTTP header. By knowing when clients will connect, the server is able to evict sessions that will be used again farthest in the future. In a special case, if the new session is also the one that will be used again farthest in the future, it will not be added to the cache and no session will be evicted. We demonstrate the advantage of the predictive policy over FIFO, LRU, and random: When the number of active clients is equal to or less than the cache size, the cache hit rate of the predictive policy is almost 100%. When the total number of active clients increases beyond the cache size, the cache hit rates of both FIFO and LRU precipitate to near zero, whereas the cache hit rate of the predictive policy decreases gracefully. Although the cache hit rate of the random policy also decreases gracefully as the cache size decreases, it always underperforms the predictive policy considerably.

We evaluate our policy on one practical application of it: mobile application advertising. Android applications (apps) which include advertisements (ads) will often regularly request a new ad to show to the user. These requests have been shown to contain sensitive user data, such as device identifiers or location information, and should be encrypted to protect this data, but rarely are [15, 8]. Serving ads over TLS using the predictive policy would narrow the performance gap between TLS and unencrypted ad traffic, incentivising ad providers to adopt encryption. Based on our knowledge of Android app advertising, we build a client to model the behavior of many Android devices which periodically come online and request new ad content at regular intervals. We use these client traffic models to benchmark the performance of our eviction policy compared to other commonly adopted eviction policies and TLS session tickets, an alternative way to implement TLS session resumption without a server cache. We find that our policy has significantly better hit rate performance than other eviction policies and can be implemented efficiently, such that it rivals TLS tickets in terms of transactions per second for reasonable cache sizes.

II. Background

A. Transport Layer Security (TLS)

Transport Layer Security (TLS) [6] contains two main protocols: the handshake protocol and the record protocol. The handshake protocol is used to negotiate the parameters of a connection while the record protocol is responsible for transferring the application data to and from the server once the connection is established. During the handshake protocol, the cipher suite is negotiated, the server (and sometimes client) is authenticated, and the master secret is transferred using a public-key encryption scheme. The number of messages transferred during the handshake varies depending on which features the server or client want to use (for example, client authentication requires three additional messages). For a full handshake with server authentication but no client authentication, a total of nine messages must be transferred before any application data is sent. The `ClientKeyExchange` and `Certificate` messages require public key operations, and are the most computationally expensive part of the handshake.

1) TLS Session Resumption: To avoid the cost of performing a full handshake when the client connects to the server again, the TLS protocol supports an abbreviated handshake using TLS sessions. When a client first connects, the server assigns a new session ID to the connection and sends it in the `ServerHello` message. Once the master secret is transferred, both the client and server store the master secret and the session ID. If the client connects again, it can send the session ID as part of the `ClientHello`, indicating it is able to reuse that session. If the `ServerHello` contains the same session ID in response, then the handshake skips to the `ChangeCipherSpec` message, without the need to send a new master secret or authenticate the server. This avoids all public-key operations, saving computation at the server and client, and reducing the number of messages sent.

2) TLS Tickets: RFC 5077 defines an extension to TLS which allows for session resumption without the need for the server to store session information [12]. To do so, the TLS server produces a TLS session ticket, which is stored by the client and contains all the information needed to resume the session. The ticket is encrypted and signed by the server using an ephemeral key, and sent during the handshake using a `NewSesssionTicket` message. The client can use this ticket to resume the session by including it as part of the `ClientHello`. The format of the ticket, as well as the way it is encrypted and signed, is not specified by the RFC and can vary by implementation.

III. Case Studies

In order to better understand how TLS session caches are implemented, we manually analyzed a number of popular TLS libraries. For each library, we determine whether the library provides a server session cache implementation, and if so which eviction policy is used. Additionally, we observe whether or not the library supports TLS tickets. We find that there is a wide variety of ways that session caches are implemented. Three libraries use a fully associative (FA) cache to store sessions, guaranteeing that all of the space allocated to the session cache will be available to store each session. Fully associative caches have the highest hit rate (in theory), however they suffer from reduced concurrency as the entire cache must be locked when accessed. Of the fully associative

TLS Library	Eviction Policy	Ticket Support
OpenSSL (v1.0.1)	FA-FIFO	Yes
GnuTLS (v3.2.3)	n/a	Yes
NSS (v3.15)	SA-FIFO	Yes
JSSE (v1.7)	FA-LRU	No
CyaSSL (v2.8.0)	SA-FIFO	No
PolarSSL (v1.3.1)	FA-FIFO	Yes

TABLE I: Session cache eviction policies and TLS ticket support for various TLS libraries. Each session cache is either fully-associative (FA) or set-associative (SA), and uses first-in-first-out (FIFO) or least-recently-used (LRU) as its eviction policy.

cache implementations, two use FIFO to evict sessions, while the JSSE library uses LRU. On the other hand, two of the libraries we investigated use set associative (SA) session caches. Set associative caches allow for more concurrent access, as only one of the sets needs to be locked when the cache is accessed, however the full size of the session cache may not be used, depending on how sessions are allocated to each set. The eviction behavior of SA caches depends on the number of sets and the associativity of the cache. Both libraries which used SA caches evict sessions from each set according to FIFO, but CyaSSL supports caches with nearly 6,000 sets and only 11 entries per set. Assuming sessions are allocated to sets uniformly at random, the behavior of such a cache is closer to RR than FIFO. In general, the lower the associativity of the cache, the more the eviction policy will behave like RR, with the extreme case being direct mapped caches. Lastly, many libraries allow for the built in session cache to be overridden by a custom one. One library, GnuTLS, provides no session cache implementation at all, requiring that a custom session cache be specified to enable session resumption. The results of our manual analysis are summarized in Table I.

IV. Predictive Eviction Algorithm

To improve server session cache hit rate, we propose an eviction policy that tries to evict sessions whose next use is farthest in the future. To do so, the server must know when each client is supposed to connect again, and so we call our policy the predictive (PRED) eviction policy. Each entry in the server's session cache contains the session ID, the next connection time of the client (we call this *the next-connect time*), and the session data. Upon a cache hit, the server updates the next-connect time of the found session in the cache to the new next-connect time (generally the current time plus the request period of the client). Upon cache misses, the server attempts to insert the new session into the cache. When the cache is full, the server evicts sessions according to our eviction algorithm (see Algorithm 1). First, the eviction algorithm attempts to evict sessions for clients that are no longer actively making requests by looking for sessions whose next-connect time is in the past. To prevent the server from wrongly evicting sessions due to network delays, the server should give sessions a grace period during which sessions are still considered active even after their predicted next-connect time has passed. If all sessions are active, the server then selects the session from the cache with the largest next-connect time and compares this next-connect time with that of the new session to be added. If the new session will make its next connection sooner than the session selected from the cache, the selected session is evicted from the cache and the new session is added; otherwise, the new session is not added to the cache and no session is evicted.

Algorithm 1 Predictive eviction algorithm (PRED).

```
1: function EVICT(new_session)
2:     cur_time ← time()
3:     old_session ← cache.get_session_by_min_next_connect()
4:     if old_session.next_connect + GRACE_PERIOD < cur_time then
5:         cache.remove(old_session)
6:         cache.add(new_session)
7:     end if
8:     old_session ← cache.get_session_by_max_next_connect()
9:     if new_session.next_connect < old_session.next_connect then
10:        cache.remove(old_session)
11:        cache.add(new_session)
12:    end if
13: end function
```

This algorithm can be thought of as an approximation of Bélády's algorithm, which is the optimal page replacement algorithm for virtual memory management [2]. Bélády's algorithm states that the page to be replaced should be the page whose next access will be farthest in the future. When clients notify the server (e.g., by the setting next-connect time to infinity) before they become inactive, PRED is equivalent to Bélády's algorithm. However, when clients become inactive without notifying the server, which is common for ad clients, PRED becomes an approximation of Bélády's algorithm, as PRED can only detect that a client has become inactive after the client has missed its next scheduled connection. This approximation should not decrease the cache hit rate of PRED much, because if the inactive session's next-connect time is short, PRED will detect its inactivity after this short period and make it a candidate for eviction (Line 4 in Algorithm 1); on the other hand, if the inactive session's next-connect time is long, it will become a likely candidate for eviction because PRED evicts the session with the largest next-connect time (Line 9 in Algorithm 1).

A. TLS Tickets

Here we consider an alternative to using a TLS session cache: TLS tickets. TLS tickets provide a way for the server to offload storing the TLS session state to the client by encrypting and signing the session state with an ephemeral key. The primary performance benefits of this approach are twofold. First, sessions stored in tickets can always be resumed as long as the server still has the ephemeral key used to encrypt and sign the ticket. Second, tickets do not require that separate connections share a centralized cache, allowing for more concurrency in some applications. The performance drawback to tickets, however, is that the server must decrypt and verify the ticket for each connection request, which may be more computationally expensive than a lookup in a

cache. In addition, tickets are an extension to TLS and are not supported by all TLS implementations, as we found in Section III. Finally, tickets make it more difficult to provide perfect forward secrecy with TLS, as the single ephemeral key used by the server to distribute tickets also allows an attacker to decrypt these tickets to get the master secret of previous connections, should the key ever be compromised [10]. On the other hand, session caching uses a separate ephemeral key per session, and so compromise of the server only allows an attacker to decrypt sessions whose information is currently in the cache, not every session that the server had negotiated since starting. We empirically measure the performance of session tickets compared to session caching in Section VI.

V. Experimental Setup

In order to evaluate the performance of PRED, we implement a TLS server and client to benchmark the performance of different session resumption strategies. We model the client's behavior based on one possible application of the PRED eviction policy: Android in-application advertising. Android applications (apps) often include advertisements (ads) that are shown to the user while the application is being used. These ads are fetched from an ad server over the Internet and are regularly refreshed so that the user is shown fresh content. The frequency that the ads are refreshed varies between apps, however it will remain consistent while the app is running, which allows the server to reliably predict the next-request time of clients. Apps which contain advertisements include an ad library that is responsible for communicating with the ad server and displaying the ads to the user. Normal ad serving expects the vast majority of requests to be ad requests (usually the click rate of ads is less than one percent). It is impressions which we are interested in encrypting, as they contain potentially sensitive user data used for ad targeting. Encrypting these requests would protect this user data from being read by network sniffers (e.g., [8, 15]). We do not consider encrypted click requests in our evaluation for the following reasons: 1) clicks usually only contain the ID of the impression that generated them and not any sensitive user data, and 2) clicks are rare and would not impact performance much. For the purpose of our benchmark, our TLS server represents the ad server, while our TLS clients represent Android devices running apps that use the ad server's ad library.

A. Modeling Client Behavior

To model the behavior of many Android devices, we assume that the ad server has a pool of c devices in the wild which contain apps that make regular requests to the ad server while they are being run by the user. When the device is making requests to the ad server, we consider it to be in the *running* state, otherwise it is *waiting*. Each device switches between the running and waiting states based on a Poisson process. A device waits for some amount of time according to an exponential distribution with parameter λ_w, after which it runs for some amount of time according to an exponential distribution with parameter λ_r. Given enough time, the expected ratio of time spent running to total time is $\frac{\frac{1}{\lambda_r}}{\frac{1}{\lambda_r}+\frac{1}{\lambda_w}}$, which we will consider to be the steady state. While running, the device connects and requests an ad at some regular interval p, which may vary between devices, but will not vary while the device is running. P, the distribution of p, depends on the value of p for different apps, as well as the popularity of each app.

B. Modeling Server Behavior

To model the server, we assume the ad server will accept ad requests from clients on some interface and respond to each client with a chosen ad. In practice, ad requests are typically performed over HTTP and have a mean size r_{req}, while responses have a mean size r_{resp}. If the ad server uses a TLS session cache, its session cache size is n. If the ad server uses the PRED eviction policy, it must also choose a grace period g, as described in Section IV. We note that the value of g is independent of the other parameters, and should be chosen based on network delays. After running for some fixed amount of time T, the server's performance is measured by the server's session cache hit rate and the average transactions per second during the time period.

C. Picking Parameters

To better understand ad request patterns of Android apps, we downloaded the top 100 apps from the Google Play market which contain one ad library (to simplify analysis). We determined how many ad libraries an app had by unpacking the app and observing the included libraries, comparing the included libraries with a list of Android ad libraries from [8]. We manually ran each app in an Android emulator and captured its network traffic. By observing the app's user interface and network traffic, we were able to determine how frequently each app refreshes its ads[1]. From this, we built P, the distribution of request intervals, weighting each interval by the number of downloads from the associated apps with that interval. Additionally, from the network captures we observed the values of r_{req} and r_{resp} to be 773 Bytes and $1,405$ Bytes, respectively.

We chose the values of λ_w and λ_r based on observed Android application usage behavior from Böhmer et al [4]. The authors found that the average length of time an app is used in one sitting differs significantly by category of app. We chose to use the "games" category for our benchmark, as games commonly have ads, which have a average app session length of about 120 seconds, giving us a λ_r of $\frac{1}{120}$. The authors also noted that the average app usage on a device is about 1 hour per day. Assuming that the usage is distributed evenly throughout the day (it isn't, but this discrepancy shouldn't affect the relative performance of session resumption schemes), we derive λ_w to be $\frac{1}{2760}$ (see Appendix A). To get a better understanding of how our choice of parameters affects the

[1]In total, 38 of the apps auto-refreshed ads. 23 of these used an interval of 60 seconds, 13 used 30 seconds, and 2 used 15 seconds. 41 apps would not display any ad, 12 crashed on startup, and 9 did not automatically refresh ads.

Parameter	Symbol	Real Value	Simulated Value
Test Time	T	$3,600$ seconds	600 seconds
Run Parameter	λ_r	$\frac{1}{120}$	$\frac{1}{20}$
Wait Parameter	λ_w	$\frac{1}{2760}$	$\frac{1}{460}$
Timing Distribution	P	P	$P * \frac{1}{6}$
Request Size	r_{req}	773 Bytes	773 Bytes
Response Size	r_{resp}	$1,405$ Bytes	$1,405$ Bytes
Grace Period	g	2 seconds	2 seconds

TABLE II: Parameters used in our experiment to model Android application advertising.

behavior of clients, we ran a simple test program that simulates client request behavior without making any requests. From this, we determined that the average number of clients running at a time is approximately $0.04c$, with a standard deviation of $0.005c$, and the expected maximum number of sessions that will actively be in use at any time is about $0.065c$ (c is the number of devices defined in Section V-A). For our benchmark, we wish to select a value for T which is large enough to ensure that a steady state is reached. Our chosen value of T which satisfies this constraint is 1 hour (3600 seconds), determined via our test program. Because 1 hour is too long to run a large number of benchmarks, we chose to speed up our simulation by a factor of 6, allowing us to model one hour of requests in 10 minutes. The final values of our parameters are summarized in Table II.

VI. Evaluation

We evaluate the performance of different session cache eviction policies by measuring the session cache hit rate of a server using the policy, while varying the size of the server's session cache. To evaluate how efficiently the PRED policy can be implemented, we measure the transactions per second of the server while varying the number of clients.

A. Implementation

We implement the client and server using OpenSSL's TLS implementation [11]. OpenSSL is a popular encryption library that is used in both server implementations (Apache's `mod_ssl`, for example) and is available as a Java security provider on Android, making it an appropriate choice for our benchmarks.

We implemented the TLS server as a multi-threaded server in C. To benchmark different session cache eviction policies, we override OpenSSL's default session cache with our own, implemented as an in-memory SQLite database [14]. This allows us to describe each policy as a simple SQL query. For benchmarking transactions per second, we implement PRED using a hash table and sorted skip list. The hash table allows for constant time lookups of session IDs, while the skip list keeps track of the next-request times of each session. Skip lists have similar performance to binary search trees, but allow for constant time access of the first and last elements, which is used for the PRED eviction algorithm.

The client is implemented in C, and simulates each device in its own thread. Although our tests simulate over 100k

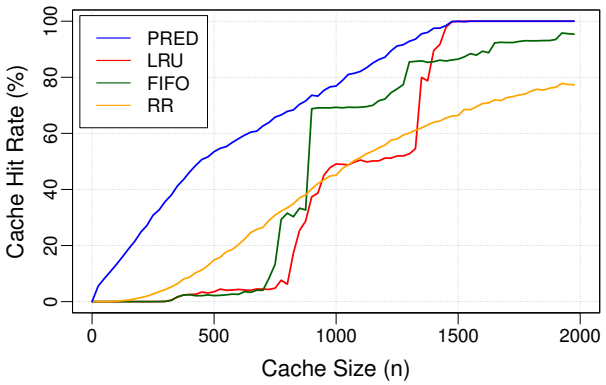

Fig. 1: Comparison of hit rates for various session cache eviction policies on Android application advertising. The size of the pool of clients, c, is $20,000$.

devices, threads spend a majority of the time sleeping (in the waiting state) or blocked waiting for the server, so the high concurrency is not a problem in practice. Once a test is complete, each thread writes a summary of its transactions to a file, which can be analyzed to determine the session cache hit rate and transactions per second. When computing cache hit rate, the first request from each thread is not considered, as these will be compulsory misses.

The client ran on a desktop machine with an Intel Xeon W3530 processor (2.8 GHz 4-core CPU) with 24 GB of RAM. To ensure that the client can saturate the server, the server ran on a less powerful desktop machine with an Intel Pentium 4 CPU (3.2 GHz 2-core CPU) with 2 GB of RAM. Neither machine supports encryption instructions in hardware. The machines were connected with a gigabit Ethernet connection, to ensure that network bandwidth was not a bottleneck.

B. Hit Rate Results

We benchmark the hit rate of four cache eviction policies: first-in-first-out (FIFO), least-recently-used (LRU), random-replacement (RR), and our proposed policy (PRED). Additionally, we benchmarked the least-frequently-used (LFU) and most-recently-used (MRU) policies, but do not report on their performance as it was significantly worse than other policies. This is because both LFU and MRU tend to keep sessions which are no longer in use in the cache indefinitely, which is very detrimental to performance. Each session cache is implemented as a fully associative cache, as fully associative caches have higher hit rates than an equivalent size set associative cache. We perform the benchmark by keeping the pool of clients, c, consistent while increasing the cache size, n, incrementally. We chose c to be $20,000$: less than the number of clients required to saturate our TLS server when no session resumption is enabled, ensuring that server load did not impact hit rate measurements. The results are presented in Figure 1.

1) RR: The hit rate of the random policy increases consistently when the cache size increases.

2) FIFO and LRU: When the cache size is smaller than the average number of clients that will be running at a time (about 800, with a standard deviation of about 100, determined via our test program from Section V-C), both FIFO and LRU perform poorly. In fact, increasing the cache size within this range has negligible impact on improving the cache hit rates of FIFO and LRU, as explained in the introduction.

As the cache size grows larger than the average number of clients that will be running at a time, the hit rates of FIFO and LRU experience a big jump because now the cache is often large enough to hold the sessions of all the running clients. However, as clients come and go, FIFO and LRU will have to evict sessions from the cache to add newly established sessions. An optimal policy would evict sessions which are no longer in use because the client is no longer running. Since older sessions are more likely to no longer be in use, FIFO performs better than LRU because LRU will be more likely to evict sessions that are just about to be used again.

As the cache size grows larger than the expected maximum number of sessions that will be in use at a time (about $1,300$, with a standard deviation of about 100, also determined via our test program), LRU achieves close to 100% hit rate, while FIFO's hit rate only increases modestly as the cache size increases. We can explain the different behavior between LRU and FIFO as follows. If each client's interval between requests is the same, then LRU and PRED are equivalent when the cache size is larger than the maximum number of sessions active at any time; in both cases the sessions that are evicted are those that remain unused longer than the request interval of the clients. Since PRED is close to an optimal policy, the fact that FIFO deviates from this behavior explains its sub-optimal performance for large cache sizes, as it may make incorrect decisions about which session to evict. For example, long-running clients would be subject to having their session evicted from a FIFO cache while the client is still active.

3) PRED: PRED is superior to RR, FIFO, and LRU under all cache sizes. For smaller cache sizes, the performance of the common FIFO and LRU policies varies greatly with the number of running clients. This could be problematic if a server experiences a sudden increase in the number of running clients, putting much greater load on the server as cache misses cause more full TLS handshakes. By comparison, the hit rate of PRED and RR degrade gracefully as the ratio of clients to cache size increases. However, RR results in a hit rate consistently about 20% lower than PRED, until the cache size grows very large.

C. Transaction Per Second Results

In the previous section, we showed that PRED achieves superior session cache hit rates compared to other eviction policies. We would now like to demonstrate that it can be implemented efficiently, so that the hit rate benefits are not offset by using a more complicated policy. To do so, we benchmark the server's performance in transactions per second as the number of clients, c, increases. We compare the performance of our implementation of PRED with TLS tickets and OpenSSL's default cache implementation. Obviously, the performance of session caches is dependent on the cache size, which leads to the difficulty of selecting a cache size such that session caching and tickets can be reasonably compared. Based on our results from the previous section, we chose n to be $6,750$, which is large enough so that PRED will achieve 100% hit rate up to $90,000$ clients, the number of clients needed to saturate our TLS server using tickets. Connections used 1024-bit RSA as the public key cipher, 256-bit CBC-AES as the bulk cipher, and SHA1 as the hash algorithm. TLS tickets in OpenSSL are encrypted with 128-bit CBC-AES and signed with HMAC-SHA256, which is not configurable. We benchmark four session resumption approaches: PRED, TLS tickets, OpenSSL's default session cache implementation, and no session resumption. The results of our test are presented in Figure 2.

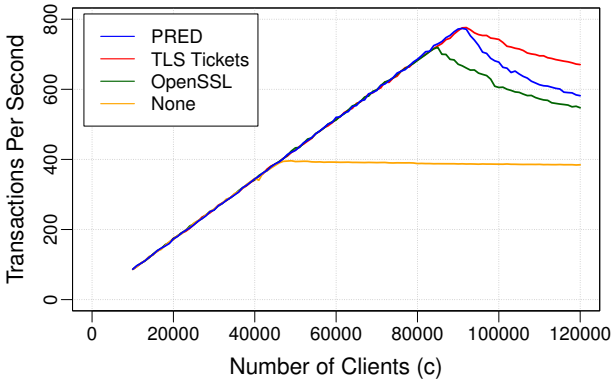

Fig. 2: Performance of each of our server implementations in transactions per second. For session caching schemes, the cache size, n, is $6,750$.

The servers which had session resumption enabled all experienced decreased performance as the number of clients increased beyond what was needed to saturate the server. This is because having a larger pool of clients means that more connections will be from clients which have not been assigned a session ID or ticket, and thus require a full handshake. Session caching schemes experienced greater slowdown than tickets once the server became saturated due to the limited size of the session cache, as increasing the number of clients reduces the hit rate of the cache. Regardless, the PRED policy was able the achieve comparable performance to TLS tickets and OpenSSL's cache implementation, up to the point the server became saturated. This lends credence to our claim that the PRED policy can be implemented efficiently, although we hesitate to draw too many conclusions from comparing PRED with tickets, as it is difficult to conclude how the effects of locking the session cache would effect the server performance in other deployments. For highly concurrent applications, it may be worth using a set-associative cache to allow concurrent writes to the cache. In our case, 780 transactions per second is a hard limit of what our server can achieve: setting OpenSSL's cache to be large enough to hold sessions for all clients does

not surpass this limit.

VII. Related Work

Prior work measures the performance benefits of SSL session caching, finding the server response time is significantly faster when sessions are reused [7]. The impact of various hardware configurations (including the hardware cache size) on SSL performance is documented in [9]. The cost of various parts of the TLS protocol can be determined by replacing operations with no-ops [5], or profiling system usage of a TLS server while running web benchmarks [1]. The former confirmed the benefits of abbreviated handshakes, but found that cache lookup times did not have a significant impact on server performance. Similarly, Zhao et al. investigate the most time consuming operations of commonly used cryptographic algorithms in TLS [16]. Analyzing the performance of SSL connections for mobile devices is done by [3] and [13]. Unlike our work, this focuses on the cost of connections for mobile clients, not servers which communicate with mobile clients.

VIII. Conclusion

We proposed PRED, a predictive eviction policy that can achieve nearly optimal cache hit rates when the server is able to know the next time each client will connect. Our evaluation shows that PRED outperforms other common eviction policies (such as random, FIFO, and LRU) and can achieve comparable throughput to session tickets.

References

[1] George Apostolopoulos, Vinod Peris, and Debanjan Saha. "Transport Layer Security: How much does it really cost?" In: *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*. Vol. 2. IEEE. 1999, pp. 717–725.

[2] Laszlo A. Belady. "A study of replacement algorithms for a virtual-storage computer". In: *IBM Systems journal* 5.2 (1966), pp. 78–101.

[3] Diana Berbecaru. "On measuring SSL-based secure data transfer with handheld devices". In: *Wireless Communication Systems, 2005. 2nd International Symposium on*. IEEE. 2005, pp. 409–413.

[4] Matthias Bohmer, Brent Hecht, Johannes Schning, Antonio Krger, and Gernot Bauer. "Falling asleep with Angry Birds, Facebook and Kindle: a large scale study on mobile application usage". In: *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*. ACM. 2011, pp. 47–56.

[5] Cristian Coarfa, Peter Druschel, and Dan S Wallach. "Performance analysis of TLS Web servers". In: *ACM Transactions on Computer Systems (TOCS)* 24.1 (2006), pp. 39–69.

[6] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246 (Proposed Standard). Updated by RFCs 5746, 5878, 6176. Internet Engineering Task Force, Aug. 2008. URL: http://www.ietf.org/rfc/rfc5246.txt.

[7] Arthur Goldberg, Robert Buff, and Andrew Schmitt. "Secure Web server performance dramatically improved by caching SSL session keys". In: *Workshop on Internet Server Performance (held in conjunction with SIGMETRICS98)*. Citeseer. 1998.

[8] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. "Unsafe exposure analysis of mobile in-app advertisements". In: *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*. ACM. 2012, pp. 101–112.

[9] Krishna Kant, Ravishankar Iyer, and Prasant Mohapatra. "Architectural impact of secure socket layer on internet servers". In: *Computer Design, 2000. Proceedings. 2000 International Conference on*. IEEE. 2000, pp. 7–14.

[10] Adam Langley. *How to Botch TLS Forward Secrecy*. 2013. URL: https://www.imperialviolet.org/2013/06/27/botchingpfs.html.

[11] *OpenSSL: The Open Source Toolkit for SSL/TLS*. 2013. URL: http://www.openssl.org/.

[12] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. *Transport Layer Security (TLS) Session Resumption without Server-Side State*. RFC 5077 (Proposed Standard). Internet Engineering Task Force, Jan. 2008. URL: http://www.ietf.org/rfc/rfc5077.txt.

[13] Youngsang Shin, Minaxi Gupta, and Steven Myers. "A Study of the Performance of SSL on PDAs". In: *INFOCOM Workshops 2009, IEEE*. IEEE. 2009, pp. 1–6.

[14] *SQLite*. 2013. URL: http://www.sqlite.org/.

[15] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. "Investigating user privacy in android ad libraries". In: *Workshop on Mobile Security Technologies (MoST)*. 2012.

[16] Li Zhao, Ravi Iyer, Srihari Makineni, and Laxmi Bhuyan. "Anatomy and performance of SSL processing". In: *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*. IEEE. 2005, pp. 197–206.

Appendix

Here we describe how we determined the client wait parameter, λ_w, used for our experiments. Given λ_r and λ_w, the expected amount of time a client will run for in a time period of length T is given by:

$$a = \frac{\frac{1}{\lambda_r}}{\frac{1}{\lambda_r} + \frac{1}{\lambda_w}} * T \tag{1}$$

From Böhmer et al [4], we determined the value of λ_r to be $\frac{1}{120}$. Additionally, the authors found that the average app usage time per device is about 1 hour per day. Assuming app usage is evenly distributed throughout the day, we use this finding to set T to be $86,400$ seconds (the number of seconds in a day) and a to be $3,600$ seconds (the number of seconds in an hour). Filling in these values from Equation 1 allows us to derive λ_w:

$$\frac{120}{120 + \frac{1}{\lambda_w}} * 86400 = 3600$$

$$120 + \frac{1}{\lambda_w} = 2880$$

$$\lambda_w = \frac{1}{2760}$$

Which is the value presented in Table II. For our benchmark, we simulated a speedup of time by a factor of 6. Redoing the derivation with a T of $14,400$, a of 600, and λ_r of $\frac{1}{20}$ yields the simulated λ_w value of $\frac{1}{460}$.