X-Ray-TLS: transparent decryption of TLS sessions by extracting session keys from memory

Florent Moriconi, Olivier Levillain, Aurelien Francillon, Raphael Troncy

X-Ray-TLS: Transparent Decryption of TLS Sessions by Extracting Session Keys from Memory

Florent Moriconi
EURECOM
Amadeus
France
florent.moriconi@eurecom.fr

Olivier Levillain
Samovar, Télécom SudParis
Institut Polytechnique de Paris
France
olivier.levillain@telecom-sudparis.eu

Aurélien Francillon
EURECOM
France
aurelien.francillon@eurecom.fr

Raphael Troncy
EURECOM
France
raphael.troncy@eurecom.fr

ABSTRACT

While internet communications have been originally all in the clear, the past decade has seen secure protocols like TLS becoming pervasive, significantly improving internet security for individuals and enterprises. However, encrypted traffic raises new challenges for intrusion detection and network monitoring. Existing interception solutions such as Man-In-The-Middle are undesirable in many settings: they tend to lower overall security or are challenging to use at scale. We present X-Ray-TLS, a new target-agnostic TLS decryption method that supports TLS 1.2, TLS 1.3, and QUIC. Our method relies only on existing kernel facilities and does not require a hypervisor or modification of the target programs, making it easily applicable at scale. X-Ray-TLS works on major TLS libraries by extracting TLS secrets from process memory using a memory changes reconstruction algorithm. It works with TLS hardening, such as certificate pinning and perfect forward secrecy. We benchmark X-Ray-TLS on major TLS libraries, CLI tools, and a web browser. We show that X-Ray-TLS significantly reduces the manual effort required to decrypt TLS traffic of programs running locally, thus simplifying security analysis or reverse engineering. We identified several use cases for X-Ray-TLS, such as large-scale TLS decryption for CI/CD pipelines to support the detection of software supply chain attacks.

CCS CONCEPTS

• **Networks** → **Security protocols**; • **Security and privacy** → *Software reverse engineering*.

ACM Reference Format:
Florent Moriconi, Olivier Levillain, Aurélien Francillon, and Raphael Troncy. 2024. X-Ray-TLS: Transparent Decryption of TLS Sessions by Extracting Session Keys from Memory. In *Proceedings of The 19th ACM ASIA Conference on Computer and Communications Security (ACM ASIACCS 2024).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM ASIACCS 2024, ASIACCS, 2024

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Modern encryption protocols provide confidentiality and integrity guarantees essential for commercial and private communications over untrusted channels (e.g., the Internet). In 2022, 97% of pages loaded by Google Chrome in the USA were over HTTPS [20]. In addition to web traffic (HTTPS), many application protocols have adopted encryption, for example, emails (IMAP/SMTP) using STARTTLS or implicit TLS, VoIP, DNS over HTTPS, Virtual Private Networks (VPN), and QUIC. While some protocols use custom encryption, most rely on Transport Layer Security (TLS). Traffic encryption has become pervasive, greatly enhancing communications security; however, this also raises new challenges for network monitoring and intrusion detection. This creates, for example, a blind spot for Intrusion Detection Systems (IDS) between the internet and internal company systems [46]. Multiple solutions have been proposed to allow network analysis of encrypted traffic. However, they are often tricky to use in practice as they require target cooperation or break the TLS security model, which is often unacceptable. Therefore, network administrators must use context-specific solutions that are often costly to implement and maintain or even impossible to deploy at scale. In this work, we focus on the decryption of TLS sessions in a generic and non-invasive way to allow easy real-world usage, e.g., in corporate networks. We propose a generic mechanism for key extraction from the process memory to solve this problem. Our approach relies on eBPF, an in-kernel feature enabling the safe execution of privileged code, to identify processes that start a TLS connection, taking a process memory snapshot before and after the TLS key exchange. We then recover secret keys from the difference between snapshots.

In this paper, we make the following scientific contributions:

- We highlight that previous approaches to TLS traffic inspection are undesirable in many settings.
- We propose a generic approach to capture key material in memory with minimal intrusiveness, no static signatures, and low overhead.
- We implement X-Ray-TLS, a tool that reliably collects TLS key material on a Linux host that only relies on existing kernel facilities, allowing easy, large-scale deployment.
- We evaluate X-Ray-TLS on different setups: major TLS libraries, CLI tools, and a web browser.

- We identify several use cases where X-Ray-TLS helps to improve security (e.g., CI/CD pipelines)

The remainder of this paper is organized as follows. Section 2 describes how TLS protocol works, synthesizing existing approaches and related work on TLS interception. Section 3 presents X-Ray-TLS' approach. In Section 4, we evaluate our approach on an extensive dataset of TLS clients and describe some of its limitations. In Section 5, we detail real-world use cases of X-Ray-TLS and compare existing interception solutions. Section 6 concludes on TLS interception approaches and describes potential improvements.

2 BACKGROUND

2.1 TLS Protocol

Transport Layer Security (TLS) is a widely used client-server protocol that provides confidentiality, server authentication, integrity protection, and, optionally, client authentication. TLS is often used to secure communications over untrusted networks such as the Internet. TLS is used to provide secure HTTP (HTTPS), virtual private network security (e.g., OpenVPN), or industrial protocols (e.g., DNP3), among others. TLS 1.3 is also used to secure QUIC [25], a network transport protocol announced by Google in 2013, which is based on UDP. It provides layer 4 and layer 6 features as an all-in-one replacement for TCP and TLS. Therefore, it aims to provide faster session establishment by reducing required round-trips.

TLS is a transport security protocol that provides security guarantees only during transit. Therefore, the local security on each channel side (i.e., client and server) is outside its scope. Therefore, accessing encrypted content while having complete control of either client or server is not considered a security vulnerability of TLS. TLS 1.0 was first defined in 1999 as an upgrade of SSLv3 (Secure Sockets Layer). Since then, TLS 1.1 (2006), TLS 1.2 (2008), and TLS 1.3 (2018) have been defined. As of 2022, only TLS 1.2 and TLS 1.3 should be used. Older versions are deprecated and not supported by major web browsers (e.g., Firefox, Chrome).

A TLS session starts with a handshake. The TLS handshake aims to agree on a shared secret between the client and the server over an untrusted channel. First, the client sends (1) a ClientHello message advertising the supported TLS version, a random number (client random), and a list of supported cipher suites. The server answers (2) with a ServerHello packet containing the chosen TLS version, a random number (server random), and the chosen cipher suite. The following packets are encrypted using handshake-specific keys agreed upon during the initial ClientHello/ServerHello exchange using the ECDHE algorithm. The client will verify that the server certificate is signed by a trusted (i.e., in client trust store) Certificate Authority. Furthermore, the certificate contains a Subject field. It represents server names protected by the TLS certificate (e.g., example.com, www.example.com, *.apps.examples.com). The certificate is valid only if the requested hostname matches the certificate's Subject. It is up to certificate authorities to ensure that a certificate requester owns all CNs matched by a certificate before issuing the certificate. Server certification validation ensures the client connects to a legitimate server for the requested hostname. Then, the client sends a final Finished message to indicate that the handshake is finished and that future traffic must be encrypted using application traffic secrets. In conclusion, TLS provides confidentiality, integrity, and server authentication guarantees in transit over an untrusted channel. Handshake for TLS 1.3 with only server-side authentication is depicted in Figure 8. Beyond the TLS 1.3 handshake we just described, various variants of TLS secure communication exist: TLS 1.2, session resumption, middlebox compatibility, or QUIC, which reuses TLS 1.3 message flow. In the following section, we describe the challenges when inspecting TLS traffic.

2.2 Related Work

TLS provides security guarantees that are essential to secure communications. However, in some contexts, traffic must be decrypted to ensure compliance, security, or optimization (e.g., cache). To this end, multiple approaches were developed. Man-In-The-Middle (MITM) places a proxy between the client and server. It will relay traffic and impersonate servers from a client perspective. Therefore, clients must trust fake certificates emitted by the proxy. Another approach is to instrument (e.g., function hook, library replacement) target program to directly retrieve plaintext traffic or disable security checks (e.g., certificate validation). While this approach often works, it is time-consuming and challenging to implement (e.g., statically linked binaries, large binaries) as it is program-specific. Cooperative approaches were developed to enable programs to log TLS session keys for debugging. The *de facto* standard SSLKEYLOGFILE environment variable allows dumping session keys to a text file. However, support for this feature is limited and often disabled, as it opens new security vulnerabilities. We detail the benefits and limitations of each approach in Section 5.1. We present below TLS inspection approaches in the literature.

Dolan-Gavitt et al. [10] proposed a framework based on PANDA for memory forensics. They applied their work for TLS 1.2 session decryption. Each target program must find a tap point that reads or writes the TLS master secret. To this end, they used an instrumented TLS server that saves the master secret; then, they used a string-based search to identify the part of the program that wrote the secret. Therefore, their method requires that the target program allows connection to an arbitrary server (i.e., instrumented server). They tried their approach on OpenSSL `s_client`, Chrome, and Firefox, among others. For all programs in the experiment, they found the tap point that wrote the master key. However, it is unclear if programs should contain a unique tap point to write TLS keys. Large software like Chrome or Firefox includes multiple subsystems to establish TLS connections: fetching pages, checking updates, account synchronization, downloading safe-browsing lists, etc. The authors did not mention if they succeeded in decrypting all TLS traffic or only a part of it. Furthermore, their approach requires the target program to run in a hypervisor: they concede a 5x slowdown over native execution. While this overhead is fine for some specific analyses, it is not acceptable for deployment on production systems.

Taubmann et al. [46] propose TLSkex for extracting TLS keys for processes running inside a virtual machine. They leverage Virtual Machine Introspection (VMI) techniques to snapshot guest memory, and they further reduce the search space using heuristics: limit to memory pages that are writable and entropy-based lookup. Furthermore, they use virtual network interfaces to slow down (i.e., block for a short period) TLS packets to allow memory snapshots to happen. Their method works on TLS 1.2. However, in 2023,

more than 90% of the traffic to the top content delivery network provider Cloudflare is encrypted using TLS 1.3 or QUIC [7]. As TLS 1.3 prevents downgrade attacks, their method can only be used on 10% of Cloudflare's traffic. Furthermore, TLS 1.3 uses 4 keys for handshake and application traffic security instead of 1 master key for TLS 1.2. Authors support their approach is useful to decrypt TLS sessions of malicious programs. However, malware tends to act differently when they detect they are running in a virtual machine, e.g., they do not run their malicious payload or contact their C&C servers [5]. In addition, they experiment with their method only on small client programs (curl, wget, OpenSSL s_client) and one server program (Apache2). Large programs such as web browsers or web-based apps (e.g., Electron-based) may be significantly more challenging, considering their large memory footprint. Finally, the method has not been tested with simultaneous handshakes. However, parallel TLS connections are pervasive with such programs. As TLSkex requires virtualization, large-scale deployment would significantly impact performance since virtualization implies a 5 times slowdown [10].

Nubeva [24] is a commercial product that allows automated decryption of TLS traffic in cloud-native environments. To this end, an agent must be installed on each virtual machine running TLS clients. They claim to support TLS 1.2 and TLS 1.3. At the start of the TLS session, the agent extracts the TLS session keys from the process memory. Their approach relies on a signature database to locate keys in process memory. The signature database contains memory patterns, such as memory content located before or after session secrets, for known programs. Therefore, a signature must exist for the target program. If no signature is found, TLS sessions cannot be decrypted. It is unclear whether the signature applies to the TLS library or the whole target software. To generate new signatures, Nubeva would need access to the target program for inspection, which is not always desirable because of confidentiality or intellectual property constraints. Furthermore, it is unclear if they require the target program to start a TLS connection to a complicit server to generate the signature (which is impossible with some programs, e.g., hard-coded URLs). Finally, there is no public insight into the manual work required to add support for a new TLS client. While Nubeva has similarities with our approach, Nuveba's approach is not generic (i.e., it does not work on unknown software), significantly increasing the efforts required to use it at scale.

DroidKex [45] provides a method for fast data extraction from process memory applied to Android applications. They demonstrate their ability to extract TLS ephemeral keys without prior knowledge of memory structure. However, Android applications tend to have a smaller memory footprint than desktop apps. Furthermore, the TLS libraries ecosystem is reduced compared to libraries available for desktop operating systems. Therefore, this method does not seem directly applicable to large programs like browsers.

An ideal method would work on TLS 1.2, TLS 1.3, and QUIC. Furthermore, it should not require the target program to run in a virtual machine or program modification. Finally, the method should be generic to work on many software, including large programs such as web browsers.

3 X-RAY-TLS' APPROACH

X-Ray-TLS allows inspection of TLS sessions made by local programs. We applied our approach to TLS clients. No fundamental limitation prevents the application of the same approach on TLS server programs. However, in the context of the use cases detailed in Section 5.2, we commonly do not have access to the TLS server (e.g., remote server over the Internet). Therefore, we focused on extracting TLS session keys from TLS libraries used in client mode. For TLS 1.3, inspection targets only application traffic (i.e., not handshake traffic). Our approach works as follows. When a TLS handshake starts, the process initiating the connection is frozen (i.e., SIGSTOP), its memory is dumped, and the process is released (i.e., SIGCONT). We only dump writable memory regions as TLS session keys cannot be stored in read-only memory areas. We hypothesize that TLS clients are not re-mapping the memory region containing TLS secrets from read-write to read-only during the handshake. When experimenting on common TLS libraries (as described in Section 4.1), we did not identify a TLS library that remaps the memory region containing TLS keys during runtime, thus verifying the hypothesis. Dumping target process memory requires root privileges on the machine running the target TLS client program. While root privileges would allow instrumenting the target program (e.g., to retrieve plaintext using function hook), X-Ray-TLS allows to achieve the same result with significantly less setup effort. When the TLS handshake is completed (i.e., first `ApplicationData` TLS record), the source process is frozen, memory is dumped, and the process is released. The two memory dumps are used to generate a memory difference (i.e., content added between the first and second memory dumps), which is expected to contain the TLS secret keys. Looking for session keys in the memory difference instead of the full memory dump significantly reduces the search space (usually from GB to KB). We end up with several key candidates as various data is written between the beginning and end of the handshake, such as TLS certificates and internal state updates. Finally, we do an optimized exhaustive key search on key candidates.

We leverage eBPF to detect TLS connections and, therefore, can identify arbitrary TLS sessions without requiring intrusive instrumentation, like program instrumentation or loading a custom Linux kernel module. X-Ray-TLS requires eBPF features added in Linux kernel 4.4 (released in 2016), so it can be used on systems with kernel 4.4 and above. eBPF lets us map network sockets (IPs/ports tuple) with the PID. Tshark is Wireshark's command line interface (CLI). It implements dissectors for TLS (from TLS 1.0 to TLS 1.3), among many other protocols. We leverage tshark for network traffic monitoring (i.e., dumping traffic to PCAP files) and brute force key candidates without re-implementing the TLS stack. Finally, we

Method	Fully generic	Require hypervisor
Dolan-Gavitt et al. [10]	no	yes
TLSkex [46]	yes	yes
Nubeva [24]	no	no
X-Ray-TLS	**yes**	**no**

Table 1: Comparison of methods that extract TLS keys from process memory.

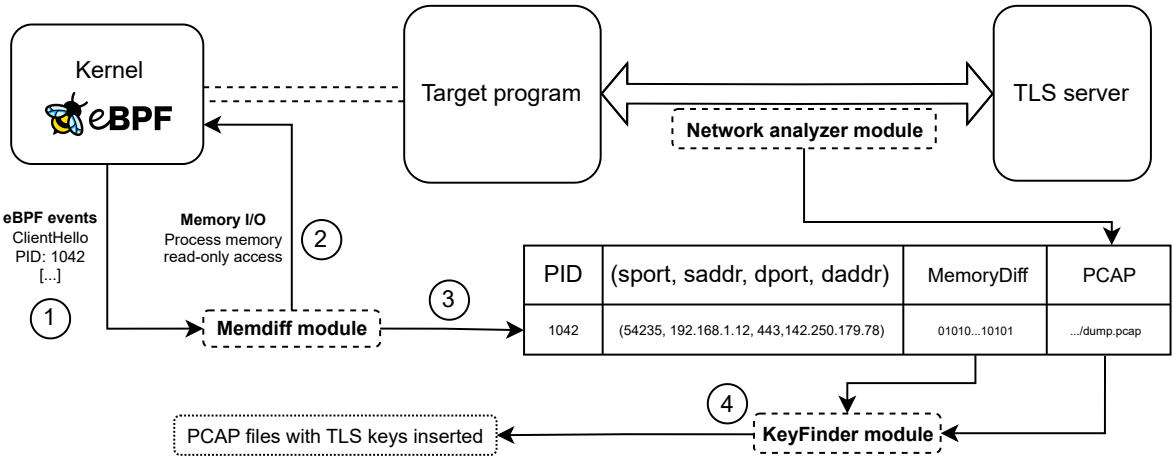

Figure 1: Overview of X-Ray-TLS architecture. `Memdiff` module computes the memory changes between the beginning and end of the handshake. `Network Analyzer` dumps network traffic to PCAP files. `KeyFinder` looks for secrets in the memory diff. Session decryption works as follows: (1) detection of the TLS session; (2) target process memory is dumped twice; (3) memory changes between dumps are saved; (4) key candidates are generated using heuristics, and then an exhaustive search is performed.

leverage Docker to provide an all-in-one, easy-to-use tool that can be used in corporate and research environments.

We detail in Section 3.1 how we detect TLS sessions and map them to the source program. Then, we detail in Section 3.2 how to generate key candidates from memory snapshots. Finally, Section 3.3 details how to test key candidates and find session keys.

3.1 TLS Session Detection

To intercept TLS sessions, we first need to reliably detect when TLS sessions are started and identify the source program. To this end, we leverage eBPF to safely run custom code in a privileged context (e.g., kernel space). Therefore, our method does not require a kernel module or patch. We use BCC [40] to compile and load eBPF code and set up a shared ring buffer between eBPF code and a managing user-space program. Ring buffers allow triggering events in the user-space program from eBPF code efficiently. We detail the performance of event triggering in Section 4.3. We attach kernel probes [26] (kprobes) on two system calls: `tcp_v4_connect` for TCP sockets and `ip4_datagram_connect` for UDP sockets. Kprobes are breakpoints set on any kernel system call (syscall). Therefore, they allow running eBPF code when a new TCP or UDP socket opens. Kprobes are run before the execution of the syscall, while kretprobes are run when the syscall returns. Kprobes allows retrieval of initial syscall arguments (before the system call might modify them). Kretprobes allow retrieving results of the syscall, e.g., socket remote peer information after socket initialization. We used kernel probes to create the mapping between sockets and the corresponding PID. Sockets are defined by the attributes of the connection: source IP, source port, destination IP, and destination port. Therefore, it allows us to map arbitrary packets on the network to the originating process.

We also leverage eBPF to attach a function in `SOCKET_FILTER` mode to a network interface. The interface should be the one used by target programs. Then the function is executed for any packet transmitted over the network interface. The processing time of the function does not impact network latency as the function works on a copy of the original packet. We detect if the packet contains an Ethernet header (which is not mandatory, e.g., layer 3 tun interface) for each packet. Then we parse TCP or UDP headers (i.e., for TLS over TCP or QUIC). Finally, when relevant packets are detected, we return events to user space over a ring buffer. Packet matching will be used to trigger memory snapshots. Therefore, we have to distinguish between initial and final snapshots. An initial snapshot should be triggered before session keys are stored in memory. For TLS, we identify `ClientHello` record by checking the first 6 bytes of the packet. For QUIC, we identify QUIC `Initial` record. The final snapshot should be triggered when session keys are stored in memory. For example, this condition is valid when the client sends packets encrypted with traffic secrets. Therefore, we detect the first `ApplicationData` packet from client to server as the final snapshot trigger. We store session states in a hash table to avoid triggering a memory snapshot for each `ApplicationData` packet.

In conclusion, the eBPF code allows us to map each network packet to its originating process and to trigger memory snapshots before and after session secrets are written to memory. In the following section, we will detail how we generate a list of key candidates from memory snapshots.

3.2 Key Candidates Generation

X-Ray-TLS extracts TLS session secrets from the target process memory. Unfortunately, there is no standard way to store such keys in memory. Therefore, the in-memory structure of the keys depends on the cryptography library used, and the memory location

is execution-specific. A naive approach is to generate key candidates from memory content using a sliding window of secret length over memory. Finding naively 2 secrets in memory of size n bytes have a complexity of n^2: this is huge considering typical values of n ranging from MB to GB, i.e., 10^6 to 10^9. Therefore, the memory range should be as small as possible. To this end, simple optimizations can be used. First, the session keys should be in writable memory regions. Dumping only writable regions (i.e., region with `w` flag) significantly reduces dump size. Second, the keys are aligned with CPU word size (often 8 bytes). Therefore, only memory addresses that can be divided by the CPU word size will be considered. This helps to divide by 8 the search space. Third, keys are high-entropy byte sequences. Entropy filtering excludes low-entropy parts of memory that are unlikely to be cryptographic keys. However, large programs tend to have a large memory footprint. They often open multiple concurrent TLS connections (which store in memory session keys, certificate chains, etc.) or load in memory content that can have high entropy (e.g., images, compressed archives). Therefore, entropy filtering performs poorly in this context.

In this work, we propose to use those techniques but also further reduce the search space by computing the difference between two memory snapshots. Figure 2 illustrates a TLS handshake and memory dumps. The target program starts by sending a ClientHello packet over the network. Note that for TLS over TCP, we do not consider the transport handshake (i.e., 3-way TCP handshake) as part of the TLS handshake. We define T_{SecGen} as the duration between detecting the ClientHello packet by network traffic analysis and when traffic secrets are stored in memory. Similarly, we define T_{stop} as the duration between the detection of the ClientHello and when the target process is frozen to be snapshot. We discuss in Section 3.2.1 different strategies for memory snapshotting. After secrets are stored in memory, the program continues processing and storing data (e.g., downloading a resource). Therefore, T_{noise} should be the lowest possible value. While a low T_{noise} is not required by design, it helps to reduce the number of key candidates and, ultimately, the key search time.

The secrets will be present in the difference between the two snapshots when the secrets do not appear in the first memory snapshot (*Initial dump*) but are present in the second memory snapshot (*Final dump*). This condition is validated if and only if $T_{stop} < T_{SecGen}$. We detail in Section 4.3 the validity of this hypothesis. We emphasize that T_{SecGen} is larger than the round-trip time of the network. Indeed, session secrets cannot be generated on the client side before receiving ServerHello (among others) records.

Method	First event	Following events	Intrapage
full-full	full dump	full dump	yes
rst-partial	RST	partial dump	no
rst-partial-rst	RST	partial dump + RST	no
full-partial	full dump + RST	partial dump	yes
full-partial-rst	full dump + RST	partial dump + RST	yes

Table 2: Memory snapshot methodologies. Partial dumping consists of dumping only the pages with dirty flags. The notion of first event/following events is per PID.

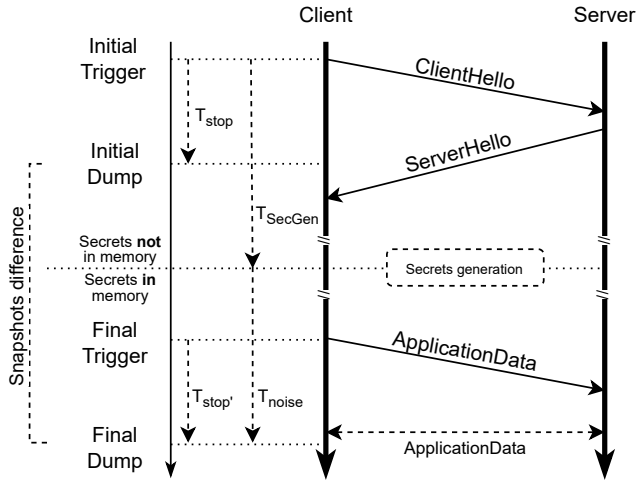

Figure 2: Memory dump cinematics in TLS handshake. T_{stop} represents the time to stop the process from a trigger event. T_{SecGen} represents the time to generate session secrets from the initial trigger. T_{noise} represents the time between when secrets are stored in memory and when the final snapshot is performed.

3.2.1 Memory Snapshot Methods. Extracting memory writes without a hypervisor is more challenging due to the lack of Virtual Machine Introspection (VMI) techniques. X-Ray-TLS dumps process memory by copying pages along with page addresses by reading pseudo-files `/proc/{pid}/pagemap` and `/proc/{pid}/mem`. For programs with large memory space or many TLS connections, doing full memory dumps twice per TLS handshake tends to slow down program execution. Therefore, we leverage a kernel feature to track memory changes. Memory tracking [15] was initially introduced in Linux kernel v3.9 (2013) to support process checkpoint-restore project CRIU [14]. Soft-dirty bit concept helps to track changed user memory pages between a reference point (reset) and a snapshot point. Therefore, tracking memory changes requires two steps: (i) reset soft-dirty bits (ii) read the pagemap and check the soft-dirty bit for each page: if set, the respective page was written to since the last reset. Therefore, in case of multiple memory dumps, the following dumps will only dump modified pages since the initial dump. This allows a significant reduction of the following dump sizes. Therefore, we developed 5 memory snapshot strategies that leverage the soft-dirty bit concept. Methods are presented in Table 2. The intra-page column refers to the ability to do intra-page differentiation. This is possible only when the first event comprises a full dump. In most cases, memory pages are 4kB in size. Then in the absence of intra-page differentiation, the memory difference can only be made with a 4kB increment (i.e., adding an entire page to the difference). Memory snapshot strategies are listed below:

- *Full-full* method takes 2 complete (i.e., all pages) process memory dumps. This method is the most straightforward, allowing fine-grained difference (i.e., intra-page). However,

it has the most impact on the target program with a high freeze time.
- *Rst-partial* method resets dirty flags on the first event and then uses partial dumps (i.e., pages with soft-dirty flag only) without resetting dirty flags. However, this approach tends to accumulate pages to be dumped as soft-dirty flags are not reset.
- *Rst-partial-rst* is similar to rst-partial with the addition of resetting soft-dirty bits after every partial dump.
- *Full-partial* allows fine-grained differences at the expense of a full initial dump.
- *Full-partial-rst* allows intra-page difference while reducing freeze time by dumping soft-dirty pages only for events after the first one. Resetting soft-dirty bits after partial dumps helps to keep the number of pages to dump low. However, a reconstruction algorithm must be used to retrieve memory changes between two arbitrary dumps.

The dump strategy's benefits and limitations will depend on the target program, particularly the number of TLS sessions. For programs that do only one TLS session and exit, resetting soft-dirty flags after the final dump will have no impact as the program will exit. We detail in Section 4 the performance of memory dump strategies.

Figure 3 illustrates two overlapping handshakes. In the case of a dumping strategy that resets soft-dirty bits after dump (namely full-partial-rst and rst-partial-rst), special care should be taken in case of overlapping handshakes. Indeed, memory dump (2) triggers a reset of soft-dirty bits. Therefore, memory dump (3) will not contain memory pages modified between memory dump (1) and (2). If secrets of session 1 are written to memory between (1) and (2), they will not appear in memory difference. This might happen depending on network RTT and the start time difference between session 1 and session 2. To avoid this problem, we develop a reconstruction algorithm to identify memory changes between two arbitrary snapshots (i.e., point in time) without requiring full dumps for all events. For the context of Figure 3, memory changes for session 1 are recovered as follows. Memory pages modified between (1) and (2) are present in partial dump (2) (and similarly for (3)). Memory changes are reconstructed by taking partial dump (2) and overloading it with partial dump (3). Overloading a memory dump refers to replacing pages that appear in both dumps with pages of the most recent dump. Generally, changes between dumps i and j, where $j > i+1$, are reconstructed by starting at dump $i+1$ and then overloading every dump by the next dump ranging from $i+2$ to j (excluded). When the first dump for the session is not a full dump but a partial dump, we go backward until we find a full dump or a reset. To avoid having a continuously growing set of memory snapshots for long-lived programs, a full dump along with an RST is done when there is no in-flight TLS *handshakes* (as illustrated as (5)). This algorithm allows the reconstruction of memory changes between two arbitrary points during process execution by only requiring soft-dirty flags kernel feature (i.e., without fine-grained memory tracking such as using a hypervisor). Memory changes between the beginning and the end of the handshake are not only composed of traffic secrets (that we are interested in). Indeed, memory changes (i.e., memory difference) are often an order of magnitude bigger than the secret size (e.g., kB compared to the secret size of 2×48 bytes). Therefore, we describe in the following section how we look for secrets in memory differences.

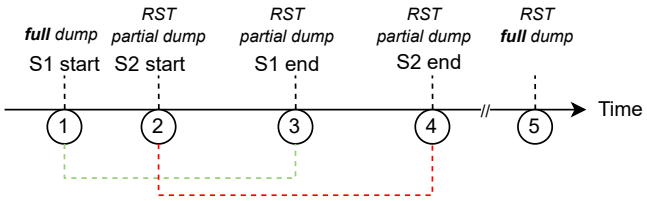

Figure 3: Illustration of changes reconstruction algorithm with the full-partial-rst method. Memory snapshots difference for session 1 (S1) is represented using a green dotted line. Memory snapshots difference for session 2 (S2) is represented using a red dotted line.

3.3 Exhaustive Key Search

Heuristics help to reduce the key search space significantly. However, depending on the target program, we still end up with potentially thousands of key candidates. We modified tshark to brute force TLS session keys to test key candidates efficiently. This task is managed by the `KeyFinder` module illustrated in Figure 1. Our modified version of tshark takes a path to a binary file containing key candidates (concatenated) and a path to a PCAP file containing encrypted traffic. Tshark will read packet by packet the PCAP file, and as soon as all required material is available (e.g., client random, server random, cipher suite), a brute force loop will start. The brute force loop stops when all secrets are found: client-to-server and server-to-client traffic secrets for TLS 1.3 and master secret for TLS 1.2. Finally, results are printed, and tshark exits. Our patch to tshark is less than 200 lines and allows brute force TLS keys without re-implementing a TLS decryption stack, which is complex and challenging to implement correctly. This approach should also simplify the implementation of new TLS versions or new protocols in the future. The next section will evaluate X-Ray-TLS on different settings.

4 EVALUATION

4.1 Benchmark

Experiments were done on a laptop with Intel(R) Core(TM) i7-8665U CPU @ 1.90GHz with 16GB of RAM. The operating system was Ubuntu 20.04.5 LTS with Docker 20.10.21. Software versions (e.g., TLS library versions) are hardcoded in benchmark source files. We present in the following section a benchmark of our tool on different programs and contexts. Then we evaluate conditions for X-Ray-TLS to work.

4.1.1 Baseline. We define a baseline approach as generating key candidates from the memory contents using a sliding window of secret length over the memory. The number of key candidates is proportional to the size of the memory contents. Therefore, the memory extent should be as short as possible. To this end, we consider optimizations detailed in Section 3.2: dumping only writable

regions, leveraging memory alignment, and excluding low-entropy areas of memory. The entropy threshold is determined using Figure 11. We will compare our method against this baseline.

4.1.2 TLS libraries. To ensure our approach works on major TLS libraries, we leverage curl support for different TLS backends. This allows us to easily test different TLS libraries with the same interface (i.e., same CLI). We built curl using the following TLS libraries [41]: OpenSSL, GnuTLS, NSS, WoflSSL, mBedTLS, and BearSSL. We have not found studies that estimate the usage of each TLS library. Based on discussions with experts and considering that software rarely implements by themselves the TLS protocol, we estimate that the 6 libraries we tested represent a significant proportion of the TLS libraries used by open-source and commercial software. We also compile curl with QUIC support using GnuTLS. For each program under test, we target 2 HTTPS URLs: a TLS 1.2-only server and a TLS 1.3-capable server. We choose TLS servers we do not control (i.e., on the internet) to highlight that X-Ray-TLS does not require any server collaboration. As X-Ray-TLS supports different memory snapshotting strategies, we benchmarked each strategy independently. Finally, we end with a test matrix of TLS library × memory snapshot strategy × URL. It shows that X-Ray-TLS can intercept TLS sessions from all major TLS libraries listed above for TLS 1.2 and TLS 1.3. We benchmarked X-Ray-TLS on CLI programs: wget, OpenSSL s_client, Python (requests module), and pip. We obtained the same results as on curl: TLS secrets extraction is successful for all snapshot strategies. To assess if our approach works on QUIC, we manually check if QUIC secrets are in memory difference generated by X-Ray-TLS when curl using the GnuTLS backend connects to a QUIC server. We used a complicit QUIC server to dump QUIC secrets. We have successfully validated the presence of QUIC secrets in the memory difference, indicating that X-Ray-TLS also works on QUIC. Therefore, all memory snapshot strategies were successful in retrieving TLS secrets. However, they differ in terms of key extraction speed.

4.1.3 Large programs. We applied both X-Ray-TLS and baseline approach on Firefox, a large software with a large memory footprint. Our approach based on memory difference generates a search space of 13MB compared to 41MB using the baseline approach. This is a reduction by 70% of key search space, leading to reduced key search time and smaller memory consumption. Therefore, X-Ray-TLS performs significantly better than naive search on large programs.

4.1.4 Continuous Integration build server. Continuous Integration (CI) is a software development practice that promotes running a regular set of checks on code change (e.g., commit). In practice, a CI orchestrator (e.g., Github Actions, Gitlab CI, Travis CI, Drone) starts a new container that will execute a set of commands to ensure updated code still meet quality requirement. Common steps are to download dependencies, compile source code and send artifacts to a remote registry. Continuous Integration systems are often used to build artifacts that will later be deployed to production systems. It is, therefore, a critical system for the security of the overall infrastructure. We discuss in Section 5.2 security considerations of CI/CD pipelines. CI builds are often managed directly by development teams, and many software stacks are used (e.g., code scanners, package managers, container image builders). Therefore, existing TLS interception methods such as Man-In-The-Middle, SSLKEYLOGFILE, or instrumentation are not usable at scale as they require deep modifications of CI builds, e.g., trusting a new CA for all programs running in the build container, or are not easily applicable to a wide range of programs (e.g., instrumentation, SSLKEYLOGFILE). Furthermore, existing solutions in the literature require virtualization: this causes a slowdown of up to 5 times [10] compared to the native execution.

To show to which extend X-Ray-TLS can be used to support continuous integration security, we used [30] to run CI builds locally. We focused on a Python-based project. The CI of the project installs development dependencies (i.e., code linter), builds a Docker image, and pushes it to a remote registry. All TLS connections made by the build container were successfully inspected. Furthermore, TLS inspection does not lead to a statistically significant build duration increase. X-Ray-TLS allows us to detect:

- Information disclosure: `pip` package manager adds various information in HTTP User-Agent header (e.g., in requests to `files.pythonhosted.org`). The header value is a JSON containing the kernel and OpenSSL versions (including patch number), OS, and glib versions, among others. This allows the remote HTTP server to detect if outdated packages (i.e., that might be vulnerable) are in use.
- Build dependencies: we were able to identify and record all resources fetched over the network. This helps to support reproducible builds even if resources are not available in the future (e.g., resources on the Internet). Being able to reproduce builds helps to detect altered artifacts by a malicious CI server.
- Artifacts traceability: detecting uploads of artifacts helps to identify the source of artifacts in the artifact registry. This helps to ensure end-to-end traceability between CI servers, artifact registry, and production systems.
- Support root cause analysis: in case of CI build failure, identifying the root cause from the build log may be challenging (many different logging formats). Decrypting HTTPS traffic allows the recovery of error messages in the HTTP body and status codes directly by parsing HTTP packets. For instance, we could identify that `pip` was trying to fetch a non-existing package by checking for HTTP 404 responses.

Therefore, we show that X-Ray-TLS allows TLS traffic decryption of CI build containers in a simpler and easy-to-use approach than existing solutions. Decrypting TLS traffic helps to identify information disclosure, build dependencies, artifacts traceability, or support root cause analysis in case of build failure.

4.2 Memory strategies comparison

After demonstrating different real-world scenarios of X-Ray-TLS, we detail the performance impact of the different parameters, such as the memory strategy. To compare the performance of snapshot strategies, we target a Python-based program that opens 10 consecutive TLS connections. It stores 100kB of random data between TLS sessions in memory to simulate parallel data processing. We describe below the interception performance of the 10th TLS session made by the target program (even if all sessions were successfully intercepted). We repeated the experiment 10 times. For programs

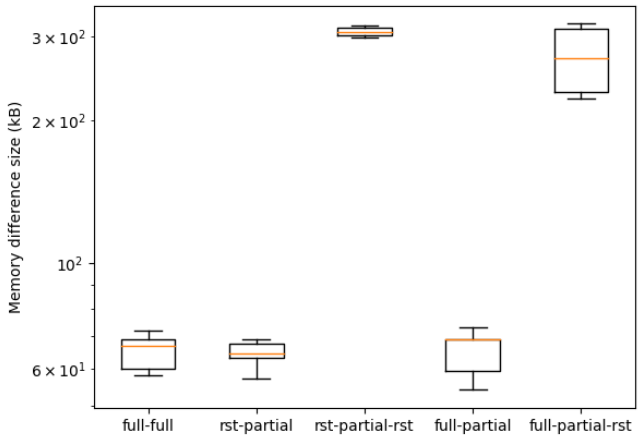

Figure 4: Size of memory snapshots difference for each memory dump strategy. Lower is better. Methods with intra-pages diff outperform per-page diff methods.

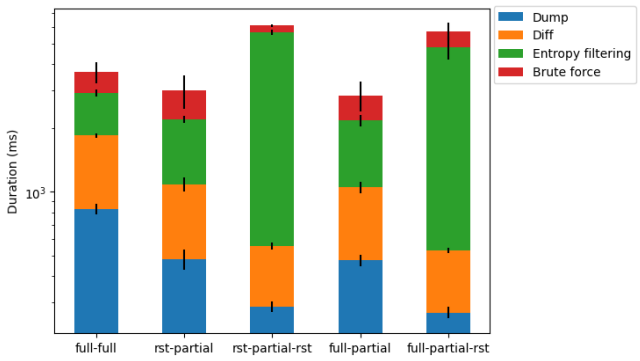

Figure 5: Duration of each step of session interception for memory snapshots strategies. Lower is better.

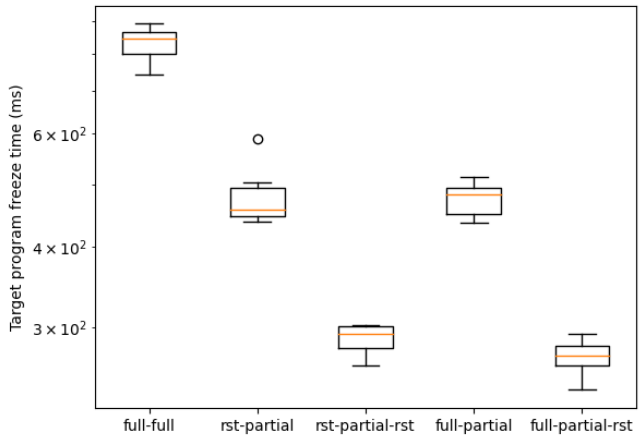

Figure 6: Target program freeze time for memory snapshots strategies. Lower is better. Strategies with partial dumps - only pages with soft-dirty bits - outperform methods with one or two full dumps.

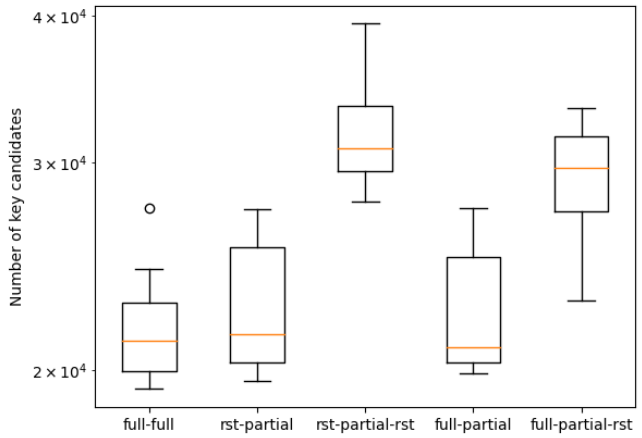

Figure 7: Number of key candidates for memory snapshot strategies. Lower is better. Strategies with smaller diff tend to have fewer key candidates. Note this is not linear because of entropy filtering (i.e., larger diff does not necessarily lead to a larger number of key candidates).

that open only one TLS connection and exits, dump strategies can be merged into 2 categories as shown in Figure 10. Figure 4 shows memory snapshots difference size for each memory strategy. Figure 5 shows the duration of each step of TLS interception. Memory strategies that reset soft-dirty flags have the lowest dump time because they only dump pages modified since the previous snapshot, compared to rst-partial or full-partial, which accumulate modified pages since the initial full dump. An entropy-filtering step is done in Python on the hex-representation of memory diff. Therefore, the performance of this step might be significantly improved with an optimized module (e.g., C binding). Figure 6 shows the target program freeze time. Reducing the number of pages to dump reduces the target program freeze time and performance impact. However, even the worst performing method (full-full) has frozen the target program for up to 900ms, which is often negligible compared to the total execution of the target program.

4.3 Measuring Time to Stop

As stated in Section 3.2, T_{SecGen} is always greater than the network RTT. Our approach requires $T_{stop} < T_{SecGen}$. We can either increase T_{SecGen} or lower T_{stop} to ensure the condition is met. T_{stop} depends on the ClientHello detection method. X-Ray-TLS relies on eBPF to monitor network traffic and push events to a Python user-space program over a ring buffer. Then the Python program will freeze the target program with a SIGSTOP signal to allow a consistent memory snapshot. We use BCC's *ring_buffer_consume()* method to reduce latency by continuously polling the ring buffer at the expense of higher CPU consumption. We estimate T_{stop} with the following methodology: after opening a TCP socket, we start

a TLS handshake using Python's SSL module. We save the time when the handshake was started using *time.perf_counter_ns()*. Then we save time when the event is received from the eBPF code. Finally, we measure T_{stop} by taking the difference between the two measured times. We measure with different 1-minute system load averages [21] (from 1 to CPU count) generated using the *stress* tool. The experiment shows that the time to stop is close to 1 ms when the load average is smaller than the number of CPU cores. When the system is overloaded (load average > CPU cores), the time to stop increases to a dozen ms (95%-percentile < 35ms). Results are shown in Figure 9. We believe this is because the scheduler does not have enough resources to run tasks as soon as they are ready, thus leading to delays. Considering overloading negatively impacts the performance of applications, production systems are rarely overloaded for long periods of time. Høiland-Jørgensen et al. [22] measure latency variations on internet. They showed that minimal RTT is above 20ms for 90% of Internet users. Therefore, for non-overloaded systems, the approach will work for more than 90% of internet users. However, Round Trip Time (RTT) might be very low (e.g., on the local network, it can be less than 0.2ms). Optimizing the pipeline to lower T_{stop} might not be enough. Therefore, another way to ensure the condition is met is to increase T_{SecGen}. This can be done by artificially delaying ClientHello or ServerHello packets. Delaying ClientHello packets brings an implementation challenge. Indeed, ClientHello is used to trigger memory snapshots. Therefore, delaying ClientHello will delay the initial memory snapshot trigger as well. Therefore, it is preferable to delay ServerHello packets. In the case of domains with multiple A/AAAA DNS records, TLS clients might try to use a different IP address if the first IP does not answer before TLS timeout (i.e., as defined in the program code). This is unlikely to happen with a delay of a dozen milliseconds. However, if it happens, this is not an issue, as the client will initiate a new TLS session that will also be processed. We have verified using a proof of concept implementation that packet delaying might be implemented using the Linux traffic control subsystem. Furthermore, it allows fine-grained packet classification using eBPF classifiers to target ServerHello records only.

4.4 Limitations

We describe in the following paragraphs the limitations and possible improvements to overcome them. Limitations can be either fundamental, related to implementation choices, or weaknesses (i.e., that can become limitations if TLS libraries evolve).

Key availability. By retrieving keys from process memory, our method is limited to TLS libraries that store keys in memory. At the time of writing, this is how major TLS libraries work. However, libraries could update their threat model to protect against the extraction of secrets from process memory. That would make key extraction with X-Ray-TLS more challenging or even impossible. We detail below mechanisms that can negatively impact the ability of our method to work. TLS 1.3 RFC [23] encourages (SHOULD) TLS implementations to remove secrets from memory as soon as they are not required anymore: "Once all the values to be derived from a given secret have been computed, that secret SHOULD be erased." For instance, handshake secrets should be removed from memory when the handshake ends. Likewise, old keys should be removed from memory on traffic key rotation. Therefore, we expect to find in memory only keys in use (i.e., application traffic secrets after the handshake) or that will be used in the future (i.e., session resumption) and not keys used in the past (i.e., pre-master secret, master secret, handshake secrets). Instead of storing raw secrets in memory, TLS clients can store XOR-ed versions of secrets in memory. In a different context, this approach is used by rclone to obfuscate credentials in the configuration file: secrets are XOR-ed with a random constant key defined in rclone's source code [42]. While it does not prevent an attacker from decrypting secrets, it still forces the attacker to write custom code to handle rclone's approach. TLS libraries could use a similar approach by XOR-ing TLS secrets with a random key. This key may even be randomly generated at the TLS client process startup. However, if session resumption is expected between runs, the random key should be saved along with the session keys. This would significantly increase the complexity of finding keys in memory as it will not be possible anymore to test keys by trying to decrypt a TLS-encrypted packet. However, the approach might impact the speed of cryptographic operations. TRESOR [34] proposes storing AES symmetric keys in processor registers to limit key availability to attackers. However, TRESOR does not protect keys during handshakes. Therefore, promptly dumping the process's memory can still allow for key recovery. As Taubmann et al. [46] state, a program can outsource cryptographic operations to another process and communicate over a named pipe or equivalent mechanism. Therefore, the process initiating the connection does not store keys in memory. To overcome this limitation, future work would identify linked processes (file descriptor, pipe, shared memory, etc.) and then dump the memory of all processes. Linux kernel allows offloading TLS operations directly to the kernel by using TLS offload feature [27]. Therefore, keys will not be accessible from a user-space process. However, a custom kernel will still allow access to keys at the expense of an increased setup effort. Furthermore, code isolation mechanisms such as Flicker [31] can run TLS code in a secure environment. Hardware-enforced secure memory could also be considered to secure keys in memory from other programs.

Performance impact. Our method requires freezing the target program for a short period to ensure a consistent memory dump. The memory dump might be large for large multi-threaded programs, leading to high freeze time.

Short-lived process. When the target program is short-lived, we might not have enough time to dump process memory. In other words, the process has gone before we could freeze it to dump memory content. We discovered this limitation with `OpenSSL s_client` without any request body: the program exits immediately after the TLS handshake. We were not able to find another example of this limitation. However, opening a TLS connection without any data transfer is unlikely to happen in real-world usage. This limitation might be mitigated by freezing the target process directly from eBPF or a kernel module.

Security impact. Our method requires to have root privilege on the client machine. Therefore, only privileged users can access TLS traffic in plaintext. When inspecting traffic from programs running in Docker containers, our program binds to the container's network

namespace. This is required to intercept network-related system calls (e.g., connect()). This approach might introduce vulnerabilities. However, containers should not be considered secure sandboxes. Therefore, the security impact should be low.

Session resumption. Session resumption is supported for TLS 1.2 if the initial session was decrypted. Indeed, the same master secret will be reused. In the context of TLS 1.3, the session resumption secret is independent of traffic secrets. Furthermore, we cannot extract this secret before a session resumption: we can not test key candidates to find the resumption key. Therefore, to support session resumption for TLS 1.3, we should keep in memory all the key candidates until session resumption. Unfortunately, this would significantly increase the storage requirement of our method.

Proof of Concept implementation. We presented here a Proof of Concept (PoC) implementation of X-Ray-TLS in Python with complete support for TLS 1.2 and TLS 1.3. However, we did not fully optimize the performance of this implementation to reduce processing time (e.g., entropy filtering time, brute force time). Therefore, performance measurements presented in Section 4 could be significantly improved (e.g., by writing time-critical sections in C). QUIC support is limited to memory snapshots, but the key oracle test has not been implemented yet. Therefore, we ensured our approach works with QUIC by testing if QUIC session secrets were present in memory diff. In this specific experience, secrets were obtained using a complicit server. QUIC supports live migration of endpoints (e.g., client IP update) using a network-independent connection ID. We did not implement session tracking; we only relied on session tracking through network attributes (IPs, ports). Finally, X-Ray-TLS allows retrieving application secrets only. While this is enough for traffic inspection, Wireshark 3.6.7 cannot decrypt PCAP files with such a key file (i.e., traffic secrets only). We provided a patch to allow easy exploration of PCAP files with Wireshark. We make our proof-of-concept implementation of X-Ray-TLS available at https://github.com/eurecom-s3/x-ray-tls. While our implementation targets Linux, there is no fundamental limitation to porting the approach to other systems (e.g., Windows, Darwin, OpenBSD) as long as they provide the same underlying features: a mechanism to detect the beginning of a TLS session and a mechanism to dump process memory. As TLS is not directly aware of the IP version in use (IPv4, IPv6), our approach can also be adapted to work for IPv6.

The following section will discuss existing TLS inspection methods and use cases where traffic inspection is desirable.

5 DISCUSSION

TLS inspection solutions are widely deployed [12] and thus raise new questions about TLS security. This section compares existing TLS inspection approaches considering their properties, merits, and limitations. Then we present different use cases where TLS traffic inspection is used.

5.1 TLS Inspection Solutions and their Limitations

Different approaches were developed to allow TLS traffic inspection. Commercial products (e.g., Netskope [35], Avast [4]) provide easy-to-use monitoring solutions for TLS traffic to support security systems such as intrusion detection, information leak detection, or web anti-tracking solutions. Open-source implementations (e.g., MITMproxy [3], Snuffy [1]) provide interception solutions mostly targeting advanced users. We describe in the following sections interception approaches along with their limitations.

Method	E2E security	Generic	Cooperative	Setup effort
Man-In-The-Middle	no	yes	yes	low
HTTPS Proxy	no	no	yes	low
Instrumentation	yes	no	no	high
SSLKEYLOGFILE	yes	no	yes	low
Memory analysis	yes	yes	no	medium

Table 3: Comparison of TLS inspection method families.

TLS interception methods can be significantly different and can be defined by the following properties:

(1) **E2E security**: no third-party have access to plaintext traffic.
(2) **Generic**: work on TLS libraries without prior knowledge of library internals (e.g., without signature database or instrumentation).
(3) **No cooperation**: the target program is not required to cooperate with interception such as dumping TLS keys by itself (i.e., SSLKEYLOGFILE), trusting a Certificate Authority, or using another hostname. We further discuss cooperative vs. malicious programs in Section 4.4.
(4) **Protocols support**: support for widely used protocols: TLS 1.2, TLS 1.3 and QUIC.
(5) **TLS hardening**: work with certificate pinning and Perfect Forward Secrecy.
(6) **Setup effort**: does not require virtualisation or kernel modification. It can be deployed in minutes (e.g., containers).
(7) **Decryption speed**: allow near real-time analysis.

We summarize in Table 3 a comparison of the main properties of existing methods. X-Ray-TLS is part of memory analysis methods. We observe that only methods based on memory analysis are generic and do not require target program cooperation. In the following sections, we will describe how existing approaches work.

5.1.1 Man-In-The-Middle (MITM). MITM approach involves an attacker placing himself between the server and the client. The attacker pretends to be a legitimate server to the client and a legitimate client to the server. Therefore, this attack requires to be an active network attacker. This can be achieved using several methods, such as ARP spoofing, DNS poisoning, or tampering with local network parameters. This approach is commonly used by network security administrators that manage network and user devices (e.g., corporate network with corporate laptops) or by locally installed software such as anti-viruses. In this context, the attacker is often called a MITM proxy. However, this method has serious limitations in terms of usability and security.

MITM allows third-party software (i.e., other than client and server) to access plaintext data. In comparison, some methods only allow the recovery of encryption keys and store them securely, which reduces the risk of plaintext leaks.

Clients must trust certificates emitted by the MITM proxy. They are generated on the fly to match the requested common names and

signed by the proxy's CA. It requires adding the proxy's CA to the trust store to avoid breaking certificate validation (i.e., preventing TLS connections from working). However, since some programs, such as Java, rely on their own trust store, enumerating all the relevant trust stores can be tiresome and lead to a significant issue in environments with much different software, such as continuous integration build servers.

Since the connection is now split into two connections, the client relies blindly on the proxy to validate the *real* server certificate. In the past, multiple examples of vulnerabilities introduced by MITM proxies were discovered [12]. Kaspersky [38] used a 32-bit key to identify generated certificates per website. Collisions enabled bypassing server certificate validation by MITM proxy, thus allowing MITM attacks between the proxy and the remote server. Avast Antitrack acts as a MITM proxy to block website trackers and ads. However, it was subject to many vulnerabilities [13], such as: not checking server certificates (which then allows trivial MITM attacks), downgrading TLS to version 1.0, and preventing the use of modern encryption algorithms, e.g., that provide Perfect Forward Secrecy. Therefore, when MITM is used, the client depends on the MITM proxy for overall session security. MITM proxy's CA allows the generation of valid certificates for any website. Therefore, it is sensitive and should be protected while generating on-the-fly certificates. Unfortunately, Kaspersky [37] failed to do so correctly. Finally, MITM will not work when a client uses certificate pinning.

Ensuring remote peer (i.e., the server from a client point of view) is legitimate can be challenging. Indeed, attackers can obtain valid certificates using a rogue or stolen certificate trusted by the client [18, 44]. Furthermore, without certificate pinning, full TLS compromise only requires the attacker to add a self-signed CA in the trust store. To address these concerns, certificate pinning is a client-side mechanism binding a host with a certificate fingerprint, whether the certificate is considered valid or not, with respect to the CA trust store. The expected certificate can refer to the server or any certificate in the chain (notably the root certificate, i.e., CA's certificate). The association can be static (defined in TLS client source code or local storage) or dynamic (discovered on the first connection). This approach is often used when the same entity manages the client and server (e.g., a mobile application connecting to the owner's servers). Special care should be taken when certificates are rotated unexpectedly (i.e., revoked due to suspected compromise). The HTTP Public Key Pinning (HPKP) [16] added certificate pinning support to HTTP. However, this is now deprecated because of its low adoption and the added complexity [39]. Instead of storing expected fingerprints in TLS clients or discovered on the first connection, they can be stored in the DNS, as defined in DNS-based Authentication of Named Entities (DANE [11]). It requires DNS records to be signed by DNSSEC. In conclusion, certificate pining prevents MITM by ensuring the client will receive a real server certificate instead of another certificate by MITM proxy (although valid). For instance, we discovered that VSCode's extension Copilot uses certificate pinning to connect to Microsoft's cloud. Certificate pinning is also widely used by Android applications [45].

5.1.2 HTTPS Proxy. Instead of opening a connection directly to a remote server, the client can be modified to open a connection to an operator-controlled domain that will act as a layer-7 proxy. For instance, a Docker client will connect to `docker.io` to fetch Docker images when using `docker pull image`. However, by using `docker pull dockerio.mydomain.com/image`, the docker client will connect to `dockerio.mydomain.com`, the operator-controlled domain. The controlled domain should be an HTTP proxy to forward HTTP requests and responses between the client and the original domain. However, the main limitation of this approach is that the target program must allow using the different hostnames to access the resource. This is often impossible for features such as telemetry or other hard-coded URLs of proprietary programs. This approach breaks end-to-end security like with MITM, lowering overall security [9].

5.1.3 Server Key Extraction. A TLS session can be decrypted by retrieving session keys. Without Perfect Forward Secrecy (PFS), session keys are encrypted using the server's public key. Therefore, by retrieving the server private key (e.g., owner cooperation, server compromise), an attacker can decrypt TLS sessions. However, this approach can only be used when the attacker controls the remote server (e.g., internet website). Furthermore, this approach does not work with TLS 1.3: PFS is optional in TLS 1.2 but mandatory in TLS 1.3. To ensure PFS, TLS decorrelates the session secret from the long-term secret (the server's private key) using an (EC)DHE key exchange signed using the private key. Thus, disclosing the private key does not reveal session keys from past sessions.

5.1.4 Instrumentation. Instrumenting target programs almost always allows retrieving plaintext traffic. Function hooks can retrieve plaintext traffic before the TLS library encrypts it. Instrumentation can also reduce connection security (e.g., disable certificate pinning [45]). However, function hooks are hard to use for programs statically linked with symbol stripped [2]. Indeed, the memory offset of relevant functions in the code should be determined manually. This is often the approach taken when performing malware analysis, where obfuscations prevent the usage of any other approach, and manual interaction becomes necessary. The instrumentation is challenging for software with large memory footprints, such as electron-based applications. Furthermore, the instrumentation is specific to each target program, thus increasing the setup effort and making it unrealistic for environments with many different software. Finally, hooking program internals adds a risk of introducing instabilities.

5.1.5 SSLKEYLOGFILE. Mozilla proposed the text-based file format NSS Key Log [33] to store TLS session secrets. It aims to provide a standard format for logging TLS secrets across programs. Wireshark supports the NSS Key Log format to decrypt TLS traffic. With compatible applications, dumping session keys is enabled using an environment variable called `SSLKEYLOGFILE` (*de facto* standard environment variable name). However, target programs must support the feature to allow TLS interception. It is up to programs to decide whether they allow TLS key logging: this is a cooperative approach. There is no guarantee that all TLS session secrets will be logged. This feature is often disabled by default at compile time (e.g., Debian packages). Indeed, it might lower TLS security. An unprivileged process running under the same user can set the environment variable (e.g., in shell startup script) and retrieve plaintext TLS traffic. Therefore, it raises a major issue for large-scale deployment.

5.1.6 Memory Analysis. TLS clients and servers must keep track of session keys in secure and fast storage. As shown in Section 4, major TLS libraries store TLS keys in process memory. X-Ray-TLS leverages this property to extract keys directly from memory. However, each TLS implementation has its own code architecture and way of storing keys in memory. Multiple TLS libraries are available [47], both open-source and closed-source. Having different TLS implementations increases the overall resiliency as a vulnerability in one implementation might not impact another implementation. In the context of session decryption using process memory, it increases the complexity as the method should be generic enough to work with different TLS stacks. Furthermore, the TLS 1.3 specification specifies that secrets should be removed from memory as soon as they are not required anymore. In particular, we observed that this recommendation is followed by the widely-used library OpenSSL. For instance, handshake secrets are removed from memory as soon as the handshake is completed. Therefore, memory snapshots should be done promptly to extract secrets. The operating system can also protect process memory from undesirable access with different mechanisms. For instance, Address Space Layout Randomization (ASLR) or Akamai's *secure heap* [8] help to prevent unauthorized access to memory containing secrets. However, these protections do not impact X-Ray-TLS as we leverage kernel pseudo-file `/proc/{pid}/mem` to access process memory, which is an interface provided by the kernel to provide access to processes memory for users with enough permissions.

TLS clients may use multiple independent TLS connections to fetch resources (e.g., web pages). Usually, there is one TLS connection per host (i.e., IP and port). For instance, Singanamalla et al. show that loading one page generates an additional median of 16 TLS handshakes [43]. Therefore, interception methods should be able to decrypt TLS sessions to allow near-real-time traffic analysis quickly. Furthermore, TLS sessions can be started simultaneously (i.e., in a lower delay than network round trip), so handshakes may overlap. This might impact, especially for methods that rely on incremental memory snapshots (e.g., using soft-dirty flags). The following section will focus on different use cases where TLS inspection by extracting session keys from memory is desirable.

5.2 Use Cases

TLS inspection can be used in multiple contexts: security analysis, to support root cause analysis, or to assist the program's internal discovery (e.g., reverse engineering). For corporate network analysis, MITM is commonly used. Instrumentation is preferred for analysis of obfuscated programs (e.g., malware) despite its complexity. Using SSLKEYLOGFILE or HTTP proxy is the easiest option when target programs are cooperative. Finally, for local non-cooperative programs, X-Ray-TLS is an appropriate option with little setup effort. We detail below different contexts where X-Ray-TLS is relevant.

Security analysis of non-cooperative programs. For security and privacy researchers, inspecting TLS traffic can help better understand how programs work. Encrypted traffic from programs using TLS hardening, such as certificate pinning, cannot be easily decrypted: MITM will not work, and instrumentation is a time-consuming and complex task. For instance, Copilot [17] is an AI-based peer programming assistant. It relies on Microsoft's cloud (over HTTPS) for making code predictions based on user development workspace. Inspecting TLS traffic allows assessing what data is sent to the cloud for code predictions. Similarly, Google Chrome processes various personally identifiable information (PII) as stated in their privacy policy [19]. Identifying which data is sent to Google may be challenging. While Chrome currently supports SSLKEYLOGFILE, there is no guarantee that all TLS keys are logged (e.g., telemetry) or that they will continue to work in the future.

Container hosting. X-Ray-TLS allows container hosting providers to decrypt customer traffic for security purposes without prior knowledge or modification of the container. For instance, 5G networks are composed of containers managed by customers. Network providers have little knowledge about what is running in containers. Decrypting TLS traffic without requiring any change from the customer side and with minimum overhead can be leveraged to detect service abuse, information leaks, or network threats.

Improving security of CI/CD pipelines. CI systems have become a target [29, 36] for compromising production systems. Indeed, gaining access to CI environment allows attackers to gain significant privileges such as altering production artifacts, accessing pipeline secrets such as access tokens (that are very often overprivileged [28]), or scanning the local network of CI/CD workers nodes. For instance, dependency confusion [6] is an effective method to run malicious code in a CI environment from the internet (i.e., without access to source code management systems or internal network). Furthermore, a one-time compromise of a CI/CD pipeline can enable long-term compromise using a self-replicating mechanism without leaving any trace in the source code [32]. Therefore, there is a strong interest in solutions that allow the monitoring of network traffic in CI/CD environments. As discussed in Section 4.1.4, existing methods for TLS inspection are difficult to use at scale, considering the high diversity of software used in CI/CD pipelines. X-Ray-TLS can be deployed on CI/CD worker nodes to allow decryption of TLS sessions made by CI/CD environments without requiring any modification of pipelines and with low overhead. Then, plaintext traffic can be saved for post-attack analysis or forwarded to an intrusion detection system for live threat detection.

6 CONCLUSION AND FUTURE WORK

We presented X-Ray-TLS, a new approach for generic and automated inspection of TLS traffic of local programs. It requires little setup effort and does not directly interact with the target program for minimum intrusiveness. X-Ray-TLS works on major TLS libraries and large software such as Firefox, does not require virtualization, and is fully generic to support current and future applications. Therefore, it directly applies to environments where many different software are used, such as CI/CD environments. To support future work using data-driven approaches, X-Ray-TLS collects pre-key and post-key memory content (i.e., content before and after secret keys) and memory offset of keys for each run. Data-driven approaches would help to reduce the search space by automatically discovering key storage patterns and therefore help to focus the key search on specific memory areas. Data-driven pattern detection would combine the effectiveness of signature-based approaches while still being generic.

AVAILABILITY AND ACKNOWLEDGMENT

The latest version of the proof-of-concept implementation of X-Ray-TLS is available online at https://github.com/eurecom-s3/x-ray-tls. The initial version of X-Ray-TLS (i.e., when the paper was published) is available at https://doi.org/10.5281/zenodo.10341300. This work was partly supported by the National Research Agency (ANR) under the Plan France 2030 bearing the reference ANR-22-PECY-0008.

REFERENCES

[1] Alessandrod. [n.d.]. Snuffy. https://github.com/alessandrod/snuffy Accessed on 12-08-2023.

[2] Alessandrod. 2020. Intercepting Zoom's encrypted data with BPF. https://confused.ai/posts/intercepting-zoom-tls-encryption-bpf-uprobes Accessed on 12-08-2023.

[3] MITMproxy's authors. [n.d.]. MITMproxy. https://mitmproxy.org/ Accessed on 12-08-2023.

[4] Avast. [n.d.]. Avast. https://www.avast.com Accessed on 12-08-2023.

[5] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2010. Efficient Detection of Split Personalities in Malware. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*. The Internet Society. https://www.ndss-symposium.org/ndss2010/efficient-detection-split-personalities-malware

[6] Alex Birsan. 2021. Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies. https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610 Accessed on 06-02-2023.

[7] Cloudflare. [n.d.]. Cloudflare Radar. https://radar.cloudflare.com/adoption-and-usage Accessed on 12-08-2023.

[8] Alex Computer. 2014. How does Akamai's 'secure heap' patch to OpenSSL work? https://blog.nullspace.io/akamai-ssl-patch.html Accessed on 12-08-2023.

[9] Xavier de Carné de Carnavalet. 2019. *Last-Mile TLS Interception: Analysis and Observation of the Non-Public HTTPS Ecosystem*. Ph.D. Dissertation. Concordia University. https://spectrum.library.concordia.ca/id/eprint/985632/

[10] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. 2013. Tappan Zee (North) Bridge: Mining Memory Accesses for Introspection. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA, 839–850. https://doi.org/10.1145/2508859.2516697

[11] V. Dukhovni and W. Hardaker. 2015. *The DNS-Based Authentication of Named Entities (DANE) Protocol: Updates and Operational Guidance*. RFC 7671. RFC Editor.

[12] Zakir Durumeric, Zane Ma, Drew Springall, Richard Barnes, Nick Sullivan, Elie Bursztein, Michael Bailey, J. Alex Halderman, and Vern Paxson. 2017. The Security Impact of HTTPS Interception. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/security-impact-https-interception/

[13] David Eade. 2020. Avast Antitrack does not check validity of end web server certificates. https://www.davideade.com/2020/03/avast-antitrack.html

[14] Pavel Emelyanov. [n.d.]. Checkpoint/Restore In Userspace, or CRIU. https://criu.org/

[15] Pavel Emelyanov. 2013. mm: Ability to monitor task memory changes (v3). https://lwn.net/Articles/546966/ Accessed on 12-08-2023.

[16] C. Evans, C. Palmer, and R. Sleevi. 2015. *Public Key Pinning Extension for HTTP*. RFC 7469. RFC Editor. http://www.rfc-editor.org/rfc/rfc7469.txt

[17] Github. [n.d.]. Github Copilot - Your AI pair programmer. https://github.com/features/copilot Accessed on 12-08-2023.

[18] Google. [n.d.]. Further improving digital certificate security. https://security.googleblog.com/2013/12/further-improving-digital-certificate.html Accessed on 12-08-2023.

[19] Google. [n.d.]. Google Chrome Privacy Notice. https://www.google.com/chrome/privacy/ Accessed on 12-08-2023.

[20] Google. 2022. HTTPS encryption on the web. https://transparencyreport.google.com/https/overview Accessed on 12-05-2023.

[21] Brendan Gregg. [n.d.]. Linux Load Averages: Solving the Mystery. https://www.brendangregg.com/blog/2017-08-08/linux-load-averages.html Accessed on 12-08-2023.

[22] Toke Høiland-Jørgensen, Bengt Ahlgren, Per Hurtig, and Anna Brunstrom. 2016. Measuring Latency Variation in the Internet. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies* (Irvine, California, USA) *(CoNEXT '16)*. Association for Computing Machinery, New York, NY, USA, 473–480. https://doi.org/10.1145/2999572.2999603

[23] Internet Engineering Task Force (IETF). 2018. The Transport Layer Security (TLS) Protocol Version 1.3. https://datatracker.ietf.org/doc/rfc8446/ Accessed on 12-08-2023.

[24] Nubeva inc. [n.d.]. Nubeva Session Key Intercept, Supported Signatures. https://docs.nubeva.com/en/latest/files/Signatures.html#linux Accessed on 12-08-2023.

[25] Jana Iyengar and Martin Thomson. 2021. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000. https://doi.org/10.17487/RFC9000

[26] Masami Hiramatsu Jim Keniston, Prasanna S Panchamukhi. [n.d.]. Kernel Probes (Kprobes). https://docs.kernel.org/trace/kprobes.html Accessed on 12-08-2023.

[27] Linux kernel's authors. [n.d.]. Kernel TLS offload. https://docs.kernel.org/networking/tls-offload.html Accessed on 12-08-2023.

[28] Igibek Koishybayev, Aleksandr Nahapetyan, Raima Zachariah, Siddharth Muralee, Bradley Reaves, Alexandros Kapravelos, and Aravind Machiry. 2022. Characterizing the Security of Github CI Workflows. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 2747–2763. https://www.usenix.org/conference/usenixsecurity22/presentation/koishybayev

[29] P. Ladisa, H. Plate, M. Martinez, and O. Barais. 2023. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1509–1526. https://doi.org/10.1109/SP46215.2023.10179304

[30] Casey Lee. [n.d.]. Run your GitHub Actions locally. https://github.com/nektos/act

[31] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. 2008. Flicker: An Execution Infrastructure for Tcb Minimization. *SIGOPS Oper. Syst. Rev.* 42, 4 (apr 2008), 315–328. https://doi.org/10.1145/1357010.1352625

[32] Florent Moriconi, Axel Ilmari Neergaard, Lucas Georget, Samuel Aubertin, and Aurélien Francillon. 2023. Reflections on Trusting Docker: Invisible Malware in Continuous Integration Systems. In *17th IEEE Workshop on Offensive Technologies (part of proceedings of SPW)*. 219–227. https://doi.org/10.1109/SPW59333.2023.00025

[33] Mozilla. [n.d.]. NSS Key Log Format. https://firefox-source-docs.mozilla.org/security/nss/legacy/key_log_format/index.html Accessed on 12-08-2023.

[34] Tilo Müller, Felix C. Freiling, and Andreas Dewald. 2011. TRESOR Runs Encryption Securely Outside RAM. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association. http://static.usenix.org/events/sec11/tech/full_papers/Muller.pdf

[35] Netskope. [n.d.]. Netskope. https://www.netskope.com/ Accessed on 12-08-2023.

[36] Marc Ohm, Arnold Sykosch, and Michael Meier. 2020. Towards Detection of Software Supply Chain Attacks by Forensic Artifacts. In *Proceedings of the 15th International Conference on Availability, Reliability and Security* (Virtual Event, Ireland) *(ARES '20)*. Association for Computing Machinery, New York, NY, USA, Article 65, 6 pages. https://doi.org/10.1145/3407023.3409183

[37] Tavis Ormandy. 2016. Kaspersky: Local CA root is incorrectly protected. https://bugs.chromium.org/p/project-zero/issues/detail?id=989 Accessed on 12-08-23.

[38] Tavis Ormandy. 2016. Kaspersky: SSL interception differentiates certificates with a 32bit hash. https://bugs.chromium.org/p/project-zero/issues/detail?id=978 Accessed on 12-08-2023.

[39] Chris Palmer. [n.d.]. Intent To Deprecate And Remove: Public Key Pinning. https://groups.google.com/a/chromium.org/g/blink-dev/c/he9tr7p3rZ8/m/eNMwKPmUBAAJ

[40] IO Visor Project. [n.d.]. BPF Compiler Collection (BCC). https://github.com/iovisor/bcc Accessed on 12-08-2023.

[41] Aina Toky Rasoamanana, Olivier Levillain, and Hervé Debar. 2022. Towards a Systematic and Automatic Use of State Machine Inference to Uncover Security Flaws and Fingerprint TLS Stacks. In *Computer Security – ESORICS 2022*, Vijayalakshmi Atluri, Roberto Di Pietro, Christian D. Jensen, and Weizhi Meng (Eds.). Springer Nature Switzerland, Cham, 637–657.

[42] rclone's authors. [n.d.]. Rclone's source code - obscure.go. https://github.com/rclone/rclone/blob/313493d51b390d7f73f0780d15bf31698f2a919a/fs/config/obscure/obscure.go#L17 Accessed on 12-08-2023.

[43] Sudheesh Singanamalla, Muhammad Talha Paracha, Suleman Ahmad, Jonathan Hoyland, Luke Valenta, Yevgen Safronov, Peter Wu, Andrew Galloni, Kurtis Heimerl, Nick Sullivan, Christopher A. Wood, and Marwan Fayed. 2022. Respect the ORIGIN! A Best-Case Evaluation of Connection Coalescing in the Wild. In *Proceedings of the 22nd ACM Internet Measurement Conference* (Nice, France) *(IMC '22)*. Association for Computing Machinery, New York, NY, USA, 664–678. https://doi.org/10.1145/3517745.3561453

[44] sslmate. [n.d.]. Timeline of Certificate Authority Failures. https://sslmate.com/resources/certificate_authority_failures Accessed on 12-08-2023.

[45] Benjamin Taubmann, Omar Alabduljaleel, and Hans P. Reiser. 2018. DroidKex: Fast extraction of ephemeral TLS keys from the memory of Android apps. *Digit. Investig.* 26 Supplement (2018), S67–S76. https://doi.org/10.1016/j.diin.2018.04.013

[46] Benjamin Taubmann, Christoph Frädrich, Dominik Dusold, and Hans P. Reiser. 2016. TLSkex: Harnessing virtual machine introspection for decrypting TLS communication. *Digit. Investig.* 16 Supplement (2016), S114–S123. https://doi.org/10.1016/j.diin.2016.01.014

[47] Wikipedia. [n.d.]. Comparison of TLS implementations. https://en.wikipedia.org/wiki/Comparison_of_TLS_implementations Accessed on 12-08-2023.

7 APPENDICES

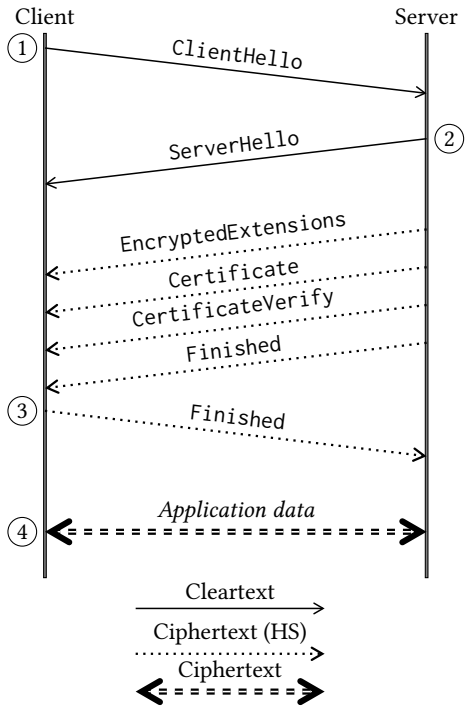

Figure 8: Illustration of TLS 1.3 handshake. TLS 1.3 requires only one round-trip for cold-start compared to TLS 1.2.

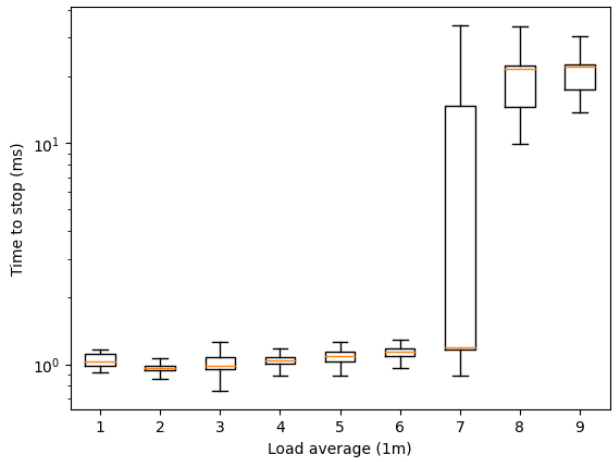

Figure 9: Evolution of the time needed to stop a target program (T_{dump}) concerning system 1-minute load average.

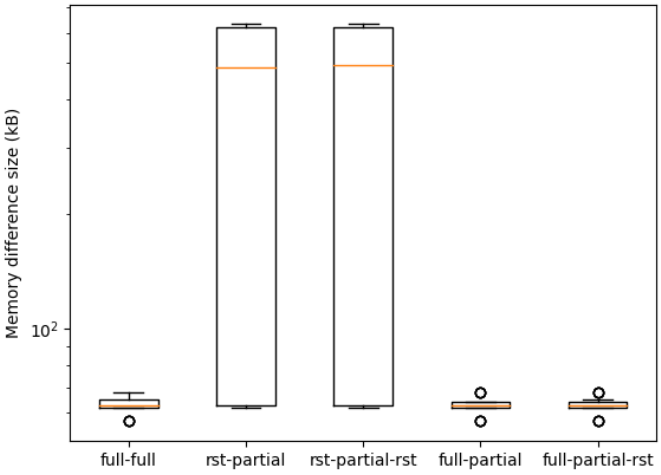

Figure 10: Illustration of memory difference size for each memory snapshot strategy for curl with 1 TLS session. Strategies can be divided into two groups: strategies with a full dump allow intra-page granularity, while strategies without a full dump are limited to page-by-page differences.

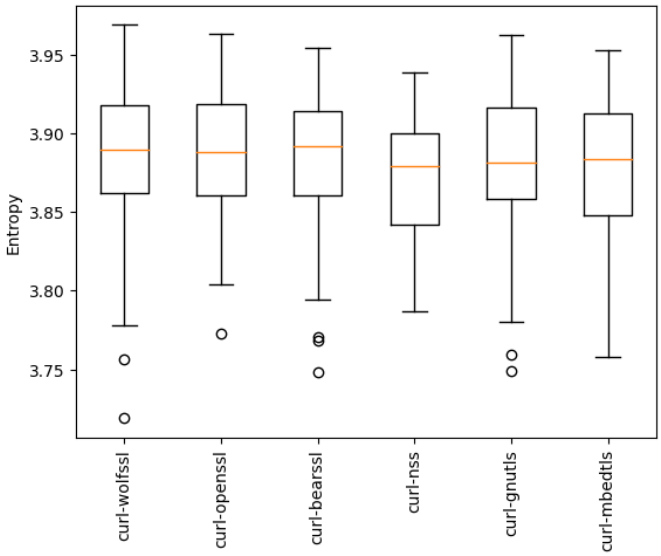

Figure 11: TLS secrets entropy per TLS library (computed on hexadecimal representation, thus maximum entropy is 4). We see a threshold (3.75) that can be used for entropy filtering.