Waterfall: A Capsule-Based Framework for Evaluating Traffic Watermarking in Anonymity Systems

Dimitri Mankowski
Ruhr University Bochum
dimitri.mankowski@ruhr-uni-bochum.de

Eduard Marin
Telefonica Research
eduard.marinfabregas@telefonica.com

Nuno Santos
INESC-ID / IST, University of Lisbon
nuno.m.santos@tecnico.ulisboa.pt

Veelasha Moonsamy
Ruhr University Bochum
email@veelasha.org

Abstract

Traffic watermarking is a powerful traffic-analysis technique and remains a practical threat to anonymous communication systems, yet its real-world feasibility is still poorly understood. Prior schemes are typically evaluated in narrow, single-scenario setups with ad hoc implementations and implicit assumptions about flow structure, coordination between embedder and detector, and how perturbations survive network noise. This makes results hard to reproduce, compare across contexts, and stress-test against modern network protocol features. This paper introduces Waterfall, a capsule-based framework for implementing and evaluating watermarking techniques on real traffic. Waterfall captures common semantic foundations of watermarking and exposes them via a unified capsule abstraction with a programmable API and declarative configuration. We implement Waterfall and validate it by reproducing and stress-testing representative schemes from the literature, and by deriving self-adaptive variants that tune their behavior using live traffic measurements. We further use Waterfall to analyze Tor's deployed Conflux protocol and show that its traffic-splitting design can unintentionally amplify watermark detectability. Finally, we demonstrate that Waterfall is also suitable for implementing and evaluating defenses against watermarking attacks.

Keywords

Traffic analysis, watermarking, anonymity networks, Tor

1 Introduction

Anonymous communication systems such as Tor [11] play a crucial role in protecting users from pervasive surveillance and traffic monitoring by governments, corporations, and other third parties [9, 12, 32, 38, 43]. A core privacy goal of these systems is *unlinkability*: preventing an adversary from associating communication sources with their destinations. Yet, unlinkability remains vulnerable to traffic analysis attacks that exploit patterns in observable metadata, even when payloads are encrypted. Among the most powerful such techniques are *watermarking attacks* [18, 20], where an adversary embeds distinctive, low-level perturbations—packet delays, losses, or header manipulations—into one segment of a communication flow and later detects these patterns at another vantage point. Watermarking has been repeatedly demonstrated as a credible deanonymization threat, but its true feasibility under modern networks is not well understood.

A major obstacle is that the watermarking literature relies on a set of implicit assumptions that have rarely been examined systematically. Prior schemes are typically designed and evaluated for a single, fixed deployment scenario and often presuppose stable network timing, predictable congestion behavior, or simple single-path routing. Evaluations are typically conducted in narrow experimental setups with fixed parameters, static designs, and ad hoc implementations [15, 17, 18, 20, 27, 36]. As a result, the community lacks a clear understanding of how watermarking behaves across diverse traffic patterns, protocol behaviors, or modern anonymity designs such as Tor onion services or Tor's emerging Conflux protocol. Even Tor developers have noted that the lack of standardized methodologies hinders rigorous evaluation of watermarking risks [30].

Our key insight is that all watermarking schemes, despite differing mechanisms and threat models, share three semantic foundations that govern their feasibility: (*i*) *flow-group semantics*, which determine the flows that must be jointly perturbed and observed; (*ii*) *coordination semantics*, which specify the information that embedders and detectors must agree on and exchange (e.g., active scheme/configuration, keys, and any required alignment metadata); and (*iii*) *perturbation semantics*, which define how modifications (e.g., drops, delays, or header tweaks) influence the observable watermark under network noise. We argue that a principled understanding of watermarking requires a unified execution model capturing these semantics. Without such a model, it is difficult to reason about the robustness of existing attacks, compare schemes across contexts, or discover their hidden assumptions.

In this paper, we introduce Waterfall, a *capsule*-based evaluation framework that realizes this unified execution model. Capsules are self-contained, event-driven environments that encapsulate the flow-group, coordination, and perturbation semantics of a watermarking scheme. Each scheme is expressed as a pair of capsules—one for embedding and one for detection—bound to specific flow groups, coordinated via runtime services and driven by a common API. This design provides a high-level yet expressive programming model for watermarking, enabling reproducible implementations,

Proceedings on Privacy Enhancing Technologies 2026(4), 1022–1040

https://doi.org/10.56553/popets-2026-0157

rapid exploration of new variants, and systematic evaluation under diverse conditions. Waterfall further provides runtime introspection, traffic monitoring, and dynamic adaptation capabilities, allowing schemes to adjust their behavior to real-time congestion patterns or throughput variations.

While Waterfall offers a unifying execution model for expressing, embedding, and detecting watermarks, it is not a full-fledged network simulator or traffic generator. Waterfall does not attempt to synthesize background traffic, emulate complete anonymity networks, or provide deterministic replay of network conditions. Instead, it focuses on the aspects that developers and analysts most frequently need but currently lack: a principled abstraction for representing watermarking logic, a runtime substrate for executing that logic on real traffic, and a standardized interface for observing its effects. We implemented a prototype of Waterfall in Go and Lua, conducting extensive evaluations to demonstrate its effectiveness as an analysis framework for watermark-based attacks. Waterfall is intended for two primary audiences. First, it helps researchers evaluate the feasibility, robustness, and hidden assumptions of traffic watermarking attacks and packet-level defenses. Second, it supports anonymity-system designers to assess how protocol or deployment changes—for example, Tor features such as Conflux—affect susceptibility to watermarking.

Using Waterfall, we replicate and stress-test three representative watermarking schemes—Inflow [20], the SDN-based traceback technique of Ling et al. [27] (*SDNt*), and RAINBOW [18]—and show how the same execution model supports both principled refinements and new scenarios. For Inflow, we show that its default configuration is more disruptive than necessary and that a throughput-aware variant reduces disruption by adapting the watermark to live flow measurements. For SDNt, we find that the original design is fragile under realistic network assumptions; we restore robustness with an adaptive throttling strategy that tracks current path conditions, yielding strong results on VPN traffic and noticeably improved detectability on Tor. We then use Waterfall to analyze Tor's recently deployed Conflux protocol [5] and show that its multi-leg traffic-splitting design, while improving performance, can unintentionally amplify watermark detectability. Finally, for RAINBOW we reproduce timing-based watermarking in its original setting and demonstrate end-to-end evaluation of defenses: Waterfall supports both known-flow detection and watermark removal, allowing us to quantify how defenses can flag or erase timing watermarks in realistic VoIP/SRTP traffic. A central insight emerging from our evaluation is that the effectiveness of traffic watermarking schemes is highly sensitive to deployment conditions. Network variability, traffic diversity, and protocol behavior can substantially alter performance, suggesting that results obtained under idealized assumptions may overestimate practical effectiveness.

In summary, our contributions are as follows:

- We identify the core semantic foundations underlying watermarking (flow-group, coordination, and perturbation semantics) and introduce capsules as the first abstraction that unifies them.
- We develop Waterfall, a capsule-based framework that provides a principled execution model and developer-friendly API for building, adapting, and evaluating watermarking schemes.
- Using Waterfall, we uncover previously unknown weaknesses in SDNt [27] and Inflow [20], and we develop improved variants enabled by runtime introspection.
- We demonstrate that Tor's Conflux protocol [5] unintentionally shifts the watermarking landscape: its multipath design hinders some single-leg attacks, but increases feasibility for adversaries that can observe, coordinate across, or combine both legs.
- We show that Waterfall supports end-to-end evaluation of defenses by reproducing known-flow detection and watermark removal against timing watermarks (RAINBOW [18]).

We make Waterfall publicly available at https://github.com/try0S/waterfall-framework. Our findings were responsibly disclosed to and acknowledged by the Tor Project team.

2 Motivation & Challenges

2.1 Motivation

Watermarking attacks continue to threaten anonymity systems [19–21, 27, 51], yet developers still struggle to assess practical feasibility. The core difficulty is combinatorial: watermarking spans a broad set of techniques and parameterizations [15, 18, 20, 27, 37], producing a large configuration space that must be explored systematically. Moreover, attack success is highly sensitive to runtime conditions (traffic patterns, congestion, buffering, routing), so conclusions drawn from a single setup often do not transfer.

The complexity of the aforementioned challenge is further exacerbated by new features and protocols designed to improve network stability, reduce latency, and increase throughput in anonymity systems [5, 34]. While these improvements are important for attracting a larger and more diverse user base, which contributes to stronger anonymity, they may also inadvertently increase exposure to watermarking attacks, as more stable and predictable traffic conditions improve their persistence. This tradeoff reveals a fundamental tension: *as anonymous communication systems are enhanced for better performance and usability, they may also become more susceptible to traffic analysis techniques*, such as watermarking.

In practice, developers typically implement schemes ad hoc, tailor them to their own systems, and build evaluation setups from scratch. This process is time-consuming, error-prone, and difficult to reproduce. A dedicated testbed would provide a common environment for implementing and evaluating a wide range of watermarking techniques, re-running experiments, and supporting mitigation studies. Currently, no such platform exists [30].

2.2 Requirements for a Traffic Watermarking Testbed

The design of a watermarking testbed involves the following stages and corresponding challenges:

Traffic Filtering: Before embedding or detecting a watermark, it is necessary to carefully select relevant traffic flows. Different watermarking schemes are suitable for different traffic types, including anonymity systems, transport protocols or traffic patterns. During the filtering stage, a watermarking testbed must therefore

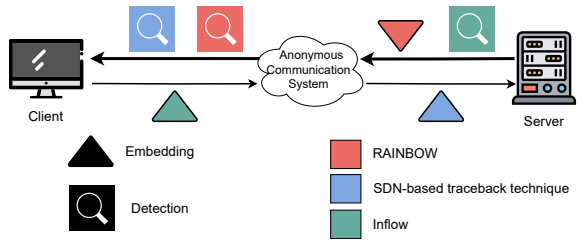


Figure 1: Constellation of existing watermarking attacks.

decide which watermarking schemes to apply to which flows, while ensuring that multiple embedding schemes executing concurrently do not interfere with each other.

For many watermarking schemes, selecting flows in a traditional way via IP addresses and ports, is often insufficient. Watermarking may require *groups of related flows*, such as the pair of client–server directions of a TCP session, multiple legs of a Tor Conflux circuit, or parallel connections belonging to the same application. Filtering must therefore enforce two constraints: (i) *exclusion*, i.e., interdependent flows must not be modified by incompatible embedding schemes, and (ii) *co-allocation*, i.e., schemes that operate on flow groups must be able to bind and process all required flows jointly.

Since some schemes are only meaningful for certain traffic patterns such as low-latency VoIP [42] or high-throughput file transfers [27], the filtering may also depend on runtime measurements such as rate or throughput. Figure 1 illustrates that embedder and detector filters may also differ substantially across attack constellations, where embedding and detection operate along the same communication path [18] or target control flows to influence interdependent data flows [20, 27].

Challenge: Provide a versatile yet easy-to-configure filtering abstraction that (i) selects the right flow groups (ii) maps them unambiguously to watermarking schemes in both embedding and detection roles with their custom traffic characteristics, and (iii) prevents conflicts between concurrently active schemes.

Coordination: Once an embedder and a detector are deployed, they must remain synchronized on a range of scheme-specific information: the active watermarking scheme and its parameters, the watermarking timeline (start/end and alignment), the watermarking key, and any feedback such as detection outcomes or metrics. In practice, this means that when an embedder selects a particular scheme and configuration for a given flow group, the corresponding detectors must be informed which scheme is active and how to correctly interpret the resulting traffic.

In simple scenarios, some schemes assume that watermarking begins with the first packet of a flow and that both sides observe packets with negligible offset. In more complex systems, especially anonymity systems such as Tor, buffering, circuit setup, and background traffic introduce unknown offsets and flow-dependent delays. As a result, several schemes require explicit alignment between embedder and detector using events or exchanged timing metadata (e.g., SDNt [27], Inflow [20]). Beyond timing alignment, many schemes additionally depend on exchanging traffic characteristics such as inter-packet delays (IPDs) [17, 18], and nearly all require a shared key or seed. In a benchmarking context, the runtime may also transmit detection outcomes back to the embedder to log evaluation metrics or to adapt parameters or algorithmic variants.

Coordination becomes even more involved when multiple schemes or detectors are active. If a single embedder can apply several watermarking schemes, it must ensure that detectors are aware of which scheme is currently active for a given flow group and which detection logic they should apply. In settings with multiple detectors and unknown egress points (e.g., due to Tor's anonymous routing), the embedder may need to distribute keys or timing metadata to several detectors, which must then attempt detection independently and report their results.

Challenge: Provide a scalable, real-time communication channel between embedders and detectors that (i) distributes timing information, scheme identifiers, keys, and traffic-related metadata, (ii) supports heterogeneous schemes with different coordination needs, and (iii) scales to many concurrent watermark instances without becoming a bottleneck.

Watermark Embedding and Detection: Embedding perturbs traffic to encode a key, while detection attempts to reconstruct that key from noisy and reshaped observations. The two processes may operate on the same flows or on different but interdependent flows, and they often rely on closely related logic.

Embedding mechanisms rely on low-level perturbations such as modifying packet headers, delaying packets or dropping packets. For example, several schemes embed watermarks into inter-packet delays (IPDs) [15, 17, 18, 36], others selectively drop packets to create silent periods or irregular patterns [20, 21], and a third class modifies flow-control parameters such as congestion windows or transmission rates to indirectly modulate throughput [27, 51]. In practice, these operations are executed through event-driven chains with state machines and precise timers (e.g., bursts of drops/delays triggered by packet arrivals or timeout events), and often need to adapt at runtime, which requires continuous traffic measurements and the ability to adjust parameters on the fly.

Detection is the counterpart: it relies on measurements to infer watermark presence and recover a candidate key. Detectors may see traffic after it has been reshaped by buffering, congestion control, or multi-path routing (e.g., Tor and Conflux). This requires detection logic that robustly extracts features such as long IPDs, throughput modulations, or header patterns under noise, jitter and reordering.

Both embedding and detection must additionally cope with heterogeneous key representations. Keys can be binary vectors, sequences of relative or absolute timestamps, or seeded constructions. Typically, keys are generated or selected offline for reproducibility, for example by sampling bit strings or timing sequences from a given distribution or using fixed seeds. Online, the embedder obtains a key to drive the perturbation pattern, while the detector attempts to extract a candidate key from the observed traffic. After extraction, the key often undergoes further offline reconstruction (e.g., thresholding or error-correcting codes) and is finally matched against the set of keys that were used for embedding.

Challenge: Provide a unified yet expressive programming and execution model for embedding and detection that supports heterogeneous traffic perturbations and flexible feature extraction under noise, jitter, and reshaping, without forcing scheme re-implementation for each environment.

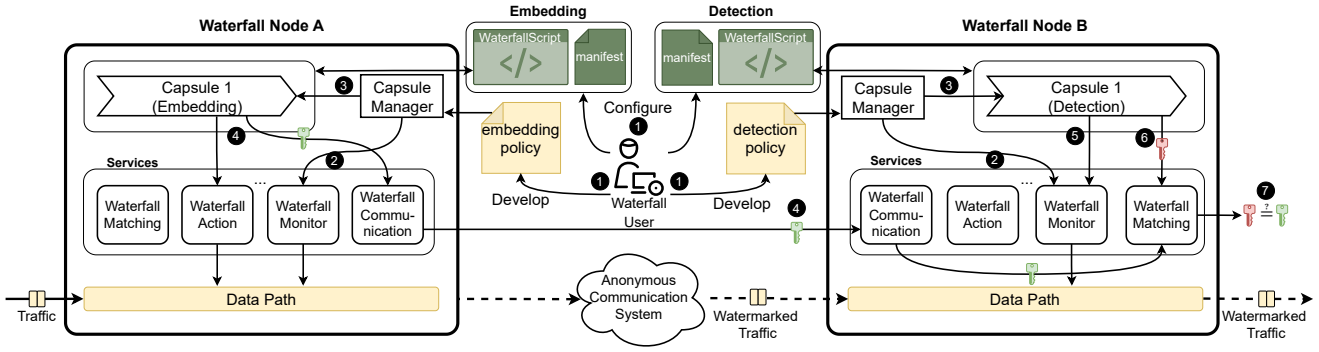


Figure 2: Overview of the Waterfall framework.

2.3 Design Goals

Our goal is to design and implement a framework that offers a flexible, scalable, and easy-to-use environment to evaluate watermarking schemes for different anonymity systems. In particular, the following criteria should be met:

Programmability. The framework should let developers implement, modify, and test a wide range of watermarking schemes without re-engineering low-level packet-processing pipelines. To achieve this, it should lift scheme logic from per-packet handlers to a flow- and flow-group-centric model with event-driven primitives, while abstracting away OS- and hardware-specific details.

Scalability. The framework should support (i) *intra-node* concurrency, i.e., many concurrent watermark instances (per flow/flow-group) and even different schemes on the same runtime instance (node), while enforcing non-interference between embedding processes on related flows; and (ii) *inter-node* scalability, i.e., distributed deployments with multiple embedding and detection nodes at different network vantage points.

Benchmarking. The framework should support automated experimentation that is reproducible at the configuration and workflow level. It should let developers run experiments over arbitrary traffic sources (custom applications, traces, or external generators) while specifying which flows are watermarked by which schemes and parameters. The same experiment specification should be executable repeatedly, with the framework automatically orchestrating embedding and detection, recording metadata and metrics, and producing comparable outputs across runs. Schemes and configurations should also be easy to package and deploy across nodes with heterogeneous environments and hardware capabilities. Such a testbed is not intended to be a network simulator such as Shadow, but rather to focus on packet-level watermarking evaluation; we revisit this distinction in Appendix B.4.

3 Waterfall

In this section, we describe our threat model and provide details about the various components of Waterfall.

3.1 Threat Model & Assumptions

Waterfall is not a general-purpose passive traffic-analysis system, but a specialized framework for controlled end-to-end evaluation of active perturbation. Accordingly, it is designed to support accurate modeling and evaluation of both attacks and defenses for packet/flow-based anonymity settings (e.g., Tor and tunneling/proxy systems) where watermarking perturbs packet timing/loss/order/rate and observes the resulting packet-level traces.

For attacks, we model and evaluate adversaries that control specific on-path vantage points on the communication path, typically one embedding point and one detection point. Depending on the scheme and scenario, these vantage points may correspond to the client-side access network (e.g., on-device firewall, ISP or gateway), a VPN or SOCKS proxy, a server or reverse proxy, an onion service, or Tor relays. Waterfall can express broader placements in principle, but the results in this paper should be interpreted with respect to these concrete vantage-point assumptions rather than as evaluations of a fully global passive-active adversary. Attacks that require endpoint compromise, access to cryptographic secrets, payload decryption, or protocol-internal modification are out of scope [19, 39].

For defenses, we assume that watermarks have already been embedded in network flows by adversaries. The defender's goal is to disrupt these watermarks to prevent successful deanonymization. We assume defenders operate at a vantage point located before the adversary's detector, allowing them to interfere with the watermark before it can be observed. The defender is restricted to the same packet-level operations supported by Waterfall. While Waterfall is designed to support both attacks and defenses, the case studies in this paper are primarily attack-centric, and the defense evaluation is correspondingly limited to defenses expressible within the same packet-level runtime model.

3.2 Overview of Waterfall Architecture

Figure 2 illustrates the general architecture of Waterfall, an embedding node and a detecting node, each running the Waterfall runtime. To evaluate a watermarking scheme on a target anonymity system, the developer follows the numbered steps shown in the figure. In Step ❶, the developer provides: (i) separate Waterfall scripts implementing the embedding and detection logic of the watermarking scheme using the Waterfall API, (ii) manifests specifying runtime parameters, filters, and configuration details for the capsules, and (iii) policies defining scheme-specific filters and parameters for instantiating the capsules. Waterfall scripts run within independent

capsules, which are self-contained environments enabling multiple watermarking schemes to operate in parallel and in isolation on Waterfall nodes. The embedding capsule and its associated files are deployed on one node (e.g., Node A), and the detection capsule and its corresponding files on another node (e.g., Node B).

Using the provided policies, both Waterfall nodes monitor the network flows passing through them to identify events that trigger the instantiation of the embedding and detection capsules (Step ❷). When a flow matches one or more pre-defined filtering conditions in the policy, it is identified as relevant for processing in the context of a specific watermarking scheme, either for embedding or detecting a watermark according to that scheme. This triggers the instantiation of the respective capsule on the corresponding node (Step ❸). Once instantiated, the embedding capsule on Node A introduces traffic perturbations to embed a *watermark key*, i.e., a scheme-specific unique identifier, into the flows and transmits the watermark key to Node B (Step ❹) using a communication channel for coordination. The detection capsule on Node B then analyzes the observed traffic to extract the embedded watermark (Step ❺). After extracting the key (Step ❻), represented as the red key in Figure 2, the detection capsule sends it to the matching service, which reconstructs it and matches the reconstructed candidate key with the key sent by Node A, thereby confirming the watermark's presence (Step ❼).

3.3 Capsules

In Waterfall, a *capsule* is an isolated execution unit that packages the three semantic foundations introduced above into a single runtime object. Concretely, a capsule specifies (i) the *flow-group semantics* of a scheme, i.e., which flows must be observed or manipulated jointly, (ii) the *coordination semantics*, i.e., what information embedder and detector must share or agree on, and (iii) the *perturbation semantics*, i.e., how packet-level modifications and measurements implement embedding and detection. Capsules maintain local state and timers, and interact with the runtime via events and Waterfall services.

At runtime, capsules receive intercepted packet and flow-level events with tight timing control and apply scheme logic without interfering with other schemes. To preserve watermark integrity, Waterfall assigns each flow to at most one *embedding* capsule at a time. In contrast, multiple *detection* capsules may attach to the same flow group concurrently, enabling parallel detectors (e.g., different hypotheses about the embedded scheme or parameters) to operate over identical traffic.

To develop and deploy a capsule-based watermarking scheme in Waterfall, the developer must provide three components: *capsule manifests*, *capsule scripts*, and *capsule policies*. Next, we illustrate the use of these components to implement the Inflow watermarking scheme on Waterfall. This scheme embeds a recognizable pattern by selectively dropping packets in the client-side control flow, which is then identified at the server-side data flow during detection (§4.2 describes the algorithms of this scheme in more detail).

1. Capsule manifests: The first step in deploying a watermarking scheme in Waterfall is to define the capsule manifests for both embedding and detection. These YAML files specify capsule properties and control runtime behavior. Figure 3 shows the Inflow manifests, which we use here to explain the semantics of each field.

Manifest Embedding

```yaml
name: InflowEmbedding
capsule_type: Embedding
exclude:
  - Reverse
flow_name: ControlFlow
parameters:
  - DropInterval: float
  - KeyPath: string
  - WatermarkLen: int
```

Manifest Detection

```yaml
name: InflowDetection
capsule_type: Detection
exclude:
  - Reverse
flow_name: DataFlow
parameters:
  - DetectionWindow: float
  - WatermarkLen: int
```

Figure 3: Capsule manifests for Inflow implementation.

Each manifest begins with a `name` field, assigning a unique identifier to the capsule. The `capsule_type` distinguishes between `Embedding` and `Detection` capsules, while `flow_name` binds the capsule to a logical flow, in this case, the control flow for embedding and the data flow for detection. The optional `exclude` section prevents interference by defining flows that no other embedding capsule may modify. For example, `Reverse` excludes the reverse direction of the bound flow, allowing the capsule to treat both directions as a single flow group while remaining isolated from other embedders. Exclusions can also be expressed using header fields (e.g., `SrcIP`, `DstIP`, `sport`) to define finer-grained non-interference boundaries. Finally, `parameters` declares the scheme-specific configuration interface that the capsule logic will consume at runtime. For Inflow, the embedding manifest exposes `DropInterval`, which defines the duration between packet drops, `KeyPath`, which indicates the keys to be embedded, and `WatermarkLen`, which specifies the total duration of the watermark. On the detection side, the manifest includes `DetectionWindow`, setting the minimum length of silent intervals to be interpreted as intentional drops, and also `WatermarkLen` for consistency in decoding.

2. Capsule scripts: After defining the manifests, developers implement the scheme logic as Lua scripts. The scripts express the core watermarking algorithm and use the Waterfall API for configuration, key handling, timing, monitoring, and coordination. This integration with the Waterfall environment supports flexible and adaptive behavior within capsules. Figure 4 shows the scripts for the Inflow scheme.

The embedding script begins by loading the `waterfall` module and retrieving configuration parameters (lines 1–6). It then obtains the Inflow key from the Waterfall Key service (line 9), and broadcasts the key to detection capsules (line 12), ensuring that both embedding and detection capsules use the same key. The watermark embedding logic is defined in the `embed_watermark` callback (lines 15–21), which drops packets on the control flow for a specified duration and schedules the next drop based on the remaining key timings. The script also schedules a termination call to `wf.stop()` after the watermark duration ends (lines 27–29). Calls to `wf.on_time()` associate each callback with a named event, e.g., "`Inflow`" or "`Stop`".

The detection script follows a similar structure. It loads parameters and invokes `wf.detect_silent_interval()` (lines 10–12) to monitor the data flow and identify silent intervals, which are timing gaps in packet arrivals that signal dropped packets. The start times of these intervals are collected into an `extracted_keys` list. Once the watermark duration elapses, the script submits the extracted key and stops the capsule (lines 15–18). These scripts

Inflow Embedding WaterfallScript

```lua
local wf = require("waterfall")

-- Retrieve configuration parameters
local key_path = wf.get_parameter("KeyPath")
local drop_interval = wf.get_parameter("DropInterval")
local watermark_len = wf.get_parameter("WatermarkLen")

-- Retrieve the watermarking key from the Key Service
local key = wf.get_key("Inflow", key_path)

-- Broadcast the key to the Matching Service
wf.broadcast_key("Inflow", key)

-- Function to embed watermark based on key timings
local function embed_watermark()
    wf.set_action_drop("ControlFlow", {duration = drop_interval *
          wf.SECOND})
    table.remove(key, 1)
    if #key > 0 then
        wf.on_time("Inflow", key[1] * wf.SECOND, embed_watermark)
    end
end

-- Schedule the first packet drop
wf.on_time("Inflow", key[1] * wf.SECOND, embed_watermark)

-- Schedule stopping the watermarking process
wf.on_time("Stop", watermark_len * wf.SECOND, function()
    wf.stop()
end)
```

Inflow Detection WaterfallScript

```lua
local wf = require("waterfall")

-- Retrieve configuration parameters
local detection_window = wf.get_parameter("DetectionWindow")
local watermark_len = wf.get_parameter("WatermarkLen")

local extracted_keys = {}

-- Detect silent intervals and extract keys
local detector = wf.detect_silent_interval("InflowDetect",
      detection_window * wf.SECOND, 0, "DataFlow",
    function(start_time)
        table.insert(extracted_keys, start_time)
    end)

-- Schedule key submission and stop after watermark_len seconds
wf.on_time("InflowDetect", watermark_len * wf.SECOND, function()
    wf.submit_key("Inflow", extracted_keys)
    wf.stop()
end)
```

Figure 4: Capsule scripts for Inflow implementation.

Policy Configuration: Embedding

```yaml
capsules:
  - name: InflowEmbedding
    parameters:
      - DroppingInterval: 3
      - KeyPath: "Keys3sec"
      - WatermarkLen: 200
    start_events:
      - dst_ips:
          - <tor_ip1>
          - <tor_ip2>
          ...
```

Policy Configuration: Detection

```yaml
capsules:
  - name: InflowDetection
    parameters:
      - DetectionWindow: 2.5
      - WatermarkLen: 200
    start_events:
      - dst_ips:
          - <tor_ip3>
          - <tor_ip4>
          ...
```

Figure 5: Capsule policies for Inflow deployments.

define watermarking logic using event callbacks and coordinate with the system using the Waterfall API (detailed in §3.4).

3. Capsule policies: While manifests define the static properties of a capsule (its name, type, parameter names and types, and exclusion conditions), capsule policies specify the runtime conditions and filters under which Waterfall instantiates capsules. Policies give researchers fine-grained control over when and how watermarking is applied, enabling reproducible experimentation and flexible evaluation of scheme behavior under varying network conditions.

Figure 5 shows policy configurations for the embedding and detection capsules of the Inflow scheme. These files associate a capsule name with its configuration parameters, as declared in the manifest. Each policy provides concrete values for the various parameters of the scheme—for example, specifying a dropping interval of 3 seconds, a key path name "`Keys3sec`", and a watermarking length of 200 seconds for the embedding side. Each policy also includes a `start_events` section, which defines the conditions that trigger capsule instantiation. In the Inflow example, embedding is triggered for flows targeting specific IP addresses (e.g., Tor relays), while detection is triggered for flows going to other Tor IPs.

In general, developers can also specify source and destination ports, and further constrain capsule activation using traffic metrics such as the number of packets seen (`count`) or the volume of data transferred over a given time window (`aggr`), which are not illustrated in this example. These options allow for precise and context-aware capsule deployment, depending on network-specific conditions.

Expressiveness and limits of the capsule abstraction: Capsules capture a broad class of packet-level watermarking schemes whose embedding and detection logic can be expressed via observable timing, loss, ordering, rate, or mutable-header perturbations at on-path vantage points. While this scope is sufficient for the schemes evaluated in this paper, it is, however, not a formal completeness claim. Waterfall enforces isolation and lifecycle management for capsule instances, but still relies on the developer to specify the correct flow groups, parameters, and coordination metadata, and does not automatically infer these for arbitrary protocols or traffic patterns. The abstraction is intentionally out of scope for schemes or defenses that require endpoint compromise, access to decrypted payload or application semantics, or protocol-internal actions such as generating Tor cells. More generally, we view capsules as an architectural synthesis informed by prior watermarking designs rather than as a claim that all watermarking schemes can be reduced to this model. Our results should therefore be interpreted as applying to this well-defined class of packet-level watermarking schemes rather than to all possible watermarking designs.

3.4 Waterfall API

The Waterfall API is a generic, versatile, and simple programming interface that enables developers to program watermarking actions and event-driven behaviors. With this API, developers can create Waterfall scripts that encapsulate the logic required for both embedding and detection of a watermarking scheme. It is organized into four main categories of functions presented briefly below. Appendix E provides a complete list and detailed documentation.

(a) Action-based functions: Embedding capsules enact traffic perturbations through the Action API, which applies modifications with explicit scope (packets/flow/group), and bounded duration or packet limits. For instance, the embedding script in Figure 4 demonstrates the use of `set_action_drop()` to drop packets on a flow for a duration according to Inflow's algorithm.

(b) Timing-based functions: Capsules can schedule callbacks with precise timers (e.g., interval-based embeddings, windowed detection), independent across capsules. For example, Figure 4 shows how `on_time()` is used to trigger a callback at a specific time as part of INFLOW's embedding logic.
(c) Monitor-based functions: Both embedding and detection capsules can query and subscribe to runtime traffic measurements (rate, throughput, silence periods, bursts, etc.) to drive adaptive behavior. As shown in Figure 4, the `detect_silent_interval()` function is used by the INFLOW detection capsule to identify silent intervals in the traffic and trigger callbacks accordingly, enabling timing-based watermark detection.
(d) Communication-based functions: To enable coordination, capsules exchange scheme identifiers, key material, timing metadata, and outcomes across Waterfall nodes via a low-latency pub/-sub control plane. The same interface can expose events and measurements to external components via a gRPC endpoint, enabling integrations such as ML-driven parameter selection or closed-loop adaptive watermarking. For example, Figure 4 shows how an embedding capsule uses `broadcast_key()` to transmit the INFLOW key to detection nodes.

3.5 Waterfall Runtime Model

As shown in Figure 2, the Waterfall runtime ties together capsules, policies, and services into a coherent execution model. It is responsible for both orchestrating capsule lifecycles and realizing the flow-group, coordination, and perturbation semantics introduced above. The runtime adopts an event-driven architecture in which capsule instances are managed by the Capsule Manager and interact with auxiliary services through a common brokered interface.
Capsule Manager: The capsule manager dynamically manages the lifecycle of capsule instances, including creation, execution, and termination, according to the policy files associated with each capsule, which specify start conditions expressed as filters. To enforce these, the Capsule Manager leverages the `WaterfallMonitor` service, which installs active monitors to continuously inspect traffic. When a monitor detects a flow that satisfies a policy's start condition, the Capsule Manager spawns a corresponding capsule instance and assigns the detected flow as its initial flow. A given set of policies and parameters thus defines an experiment configuration: replaying the same workload under the same configuration instantiates the same capsules and executes the same embedding/detection logic, enabling reproducible benchmarking. To preserve watermark integrity and avoid interference, the capsule manager ensures each flow is assigned to at most one embedding capsule. During execution, capsules may request to bind additional related flows (e.g., session counterparts or flows sharing addresses). Bind requests are granted only if the target flow is unassigned, after which it is added to the capsule by triggering an event. Once the capsule completes its task, the Capsule Manager terminates the instance and removes its flows from the exclusion list.
Waterfall Services: To help developers focus on watermarking logic without dealing with low-level or repetitive tasks, Waterfall exposes a set of auxiliary services via its API. These services support complex behaviors and enable capsules to adapt dynamically to changing runtime conditions, including network flow dynamics, packet-level statistics, or contextual information from previous executions. Table 6 in Appendix B.3 summarizes the services and their capabilities. For instance, the `WaterfallMonitor` service lets developers define monitoring criteria at runtime and receive immediate feedback, allowing capsule behavior to adjust fluidly to varying traffic patterns, reducing the need for manual configuration.

From Challenges to Runtime Model: Waterfall organizes watermarking schemes along three core semantics: *flow-group semantics*, which capture how flows are selected and grouped; *coordination semantics*, which capture how embedder and detector remain aligned; and *perturbation semantics*, which capture how packet-level modifications become observable watermarks under noise. This decomposition provides a common structure for modeling diverse schemes uniformly and enables consistent comparison across workloads, protocols, and attacker vantage points. In the runtime, these semantics are made explicit through concrete mechanisms. Flow-group semantics are realized through manifests and policies, which specify which flows belong together, while exclusion rules and capsule assignment prevent conflicting embedders from modifying overlapping flow groups. Coordination semantics are realized through Waterfall's communication and matching services, which make scheme identifiers, keys, alignment metadata, and reconstruction logic explicit rather than leaving them implicit in ad hoc code. Perturbation semantics are realized through the Action, Timing, and Monitor APIs, which provide a common execution model for expressing packet-level modifications and corresponding detector logic under noise, jitter, and reshaping.

4 Evaluation

In this section, we present an extensive evaluation of Waterfall, demonstrating its versatility in modeling and analyzing watermarking attacks and defenses for anonymized communications. We structure the evaluation around four key questions:

- **Q1:** Can Waterfall reproduce existing watermarking schemes and match their reported results? (§4.2)
- **Q2:** Can Waterfall enhance watermarking schemes, adapting their behavior to traffic conditions? (§4.3)
- **Q3:** Can Waterfall evaluate the resilience of new network protocols against watermarking attacks? (§4.4)
- **Q4:** Can Waterfall support the development and testing of defenses against watermarking schemes? (§4.5)

Before addressing these questions in detail, we first describe our evaluation methodology and metrics.

4.1 Evaluation Methodology, Traffic Classes & Metrics

To answer the research questions above, we implemented a full prototype of Waterfall in Go and Lua and conducted extensive experiments. For packet interception/manipulation we use `iptables` with `NetfilterQueue` [1]. For passive monitoring, we rely on `libpcap` via the Go `pcap` package [3]. A runtime overhead characterization is provided in Appendix C.3. Each experiment embeds and detects watermarks in live traffic. The methods vary per experiment, as described in their respective sections. For each active scheme, we

generate 10 random candidate keys following the method used in the original paper. In each run, one key is embedded and the detector attempts to extract and match it. We repeat each configuration multiple times, with all run counts summarized in Table 11. We reference each experiment configuration by an ID, E01–E42.

Traffic classes covered: Our choice of traffic classes and deployment settings is guided by two complementary objectives. First, we aim to reproduce prior results as closely as possible. Second, we aim to extend evaluation to new or previously untested workloads in order to assess how strongly reported performance depends on the underlying traffic model. Concretely, our evaluation spans several traffic classes rather than a single application family. For RAINBOW, we evaluate both replayed SSH traces and live VoIP/S-RTP traffic. For SDNt, we evaluate VPN traffic as well as Tor traffic with file download. For Inflow, we use Tor onion-service traffic with file download, MP4 video, and HLS live-streaming workloads. In addition, to move beyond negatives taken only from the same traffic workload setting as the watermarked traffic, Section §C.4 introduces mixed-background Tor workloads that combine concurrent browser sessions with background transfer activity. These workloads are intended to cover a variety of distinct settings: bulk transfer, replayed traces, streaming, periodic real-time traffic, and mixed concurrent background activity, yet they are not exhaustive. This diversity is important because it reveals how strongly watermarking performance depends on traffic type, rather than relying on homogeneous workloads that may obscure such effects.

Coordination assumptions: Although coordination requirements differ by scheme, in our work, we assume that the embedder and detector pre-agree on the active watermarking scheme and its parameters, unless stated otherwise. In all embedding experiments, the embedder shares the watermark key material. Some schemes require further coordination metadata: for RAINBOW, the detector additionally relies on a reference IPD sequence; for SDNt and Inflow, alignment requires timing metadata such as start timestamps. We assume that keys and coordination metadata arrive reliably and in time over the coordination channel. While these conditions are somewhat idealized, they primarily serve to faithfully reproduce the attacker models from prior work rather than to claim that such coordination is equally easy in deployment.

Testbed setup: Experiments were conducted using hosts on our campus network (Ubuntu 24.04 [8]) and a remote Hetzner vCPU server [2] to introduce WAN latency. Hosts act as clients, proxies (e.g., VPN, SIP, or SOCKS), or servers, depending on the scenario. Tor experiments use Tor v0.4.8.13 with traffic routed via SOCKS. For standard client-driven workloads, we use `curl`; for browser-driven workloads, we automate Firefox/Tor Browser using `tbselenium` [4, 31]. We refreshed circuits between runs and repeated tests on different days to reduce the impact of transient Tor conditions. VPN experiments use OpenVPN v2.6.12 [50]. VoIP experiments use SRTP traffic between an Asterisk PBX (v20.6.0) [41] and `baresip` agents [14] controlled via Python. For a socks Proxy, we use a Dante (v1.4.3) [22].

Metrics: We report the following three metrics, which capture both detection accuracy and the scheme's reliability:

Table 1: INFLOW performance on onion-service traffic.

Traffic type	TPR (%)	FPR_1 (%)	FPR_2 (%)
File download[a] (E01)	97	0.0	0.0
MP4 video[b] (E03)	78	0	0.5
HLS live stream[b] (E02)	84	0.8	1

[a] Gap-pattern detector (original INFLOW).
[b] Large-IPD alignment matcher.

- **TPR** (True Positive Rate): On watermarked traffic, the fraction of runs in which the detector matches the true embedded key.
- **FPR_1** (False Positive Rate 1): On unwatermarked traffic, the average fraction of candidate keys that are falsely matched.
- **FPR_2** (False Positive Rate 2): On watermarked traffic, the average fraction of incorrect candidate keys that are falsely matched. This lets us assess whether the scheme can reliably encode more than a single bit (watermarked vs. unwatermarked); which is sometimes referred to as fingerprinting [17].

We compute these metrics as follows: For each unwatermarked run, we apply the detector to all K candidate keys. If n of these K keys match, the run contributes n/K to FPR_1. For each watermarked run, we apply the detector to the true embedded key and to the remaining K-1 candidate keys. If the true key matches, the run contributes one true positive; otherwise, it contributes zero. If n of the K-1 incorrect keys also match, the run contributes n/(K-1) to FPR_2. We average these per-run quantities over all runs and, to reduce key-specific bias, over experiments in which each key is used as the embedded key.

4.2 Reproducing Existing Watermarking Schemes

In this section we address Q1, investigating whether Waterfall can faithfully reproduce existing watermarking schemes. We implemented the schemes as closely as possible to the way they are described in their original papers, and compared our results to the reported ones. One-to-one comparisons are non-trivial, as prior work uses different evaluation metrics and omits key details essential for accurate reproduction; Appendix B.1 discusses these issues in more detail. We first summarize the schemes and then report our findings from implementing and running them in Waterfall.

Selected watermarking schemes: We evaluate three established, representative schemes, each employing a distinct approach to embed watermarks for different communication systems. Inflow [20] performs client-side watermarking by dropping bursts of TCP ACKs (control flow) over a 200-second period, triggering TCP and Tor's congestion control and creating server-visible silent intervals. SDNt [27] implements server-side watermarking by embedding a watermark key by modulating the advertised window size (AWND) in the TCP packets sent from the proxy to the server. RAINBOW works by slightly delaying packets, creating a recognizable pattern that encodes the selected watermark key, applicable at either endpoint. Inflow targets Tor onion services, SDNt targets VPN, SSH and Tor traffic and RAINBOW is mostly designed for SSH traffic.

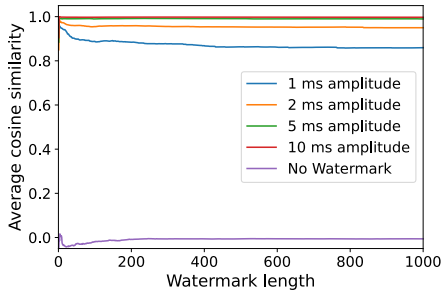


Figure 6: RAINBOW TPR/FPR similarity over key lengths.

Table 2: RAINBOW on VoIP traffic with different amplitudes.

	1 ms (E24)			2 ms (E25)			5 ms (E26)		
WM len.	TPR	FPR_1	FPR_2	TPR	FPR_1	FPR_2	TPR	FPR_1	FPR_2
50	97.0	1.5	4.7	99.0	0.5	1.1	100.0	0.0	0.1
100	98.0	0.8	3.4	100.0	0.0	0.0	100.0	0.0	0.0
200	99.5	0.5	0.6	100.0	0.0	0.0	100.0	0.0	0.0
500	100.0	0.0	0.0	100.0	0.0	0.0	100.0	0.0	0.0

1. Reproducing the original schemes faithfully: Inflow and RAINBOW can be replicated exactly as described in their original papers, without requiring modifications. Using Waterfall, we reproduced their reported detection rates while applying even stricter false-positive tests and a stronger reproducibility discipline.

For Inflow, which targets Tor onion-service flows, the embedder drops bursts of ACK packets according to a key, i.e., a list of drop times. On average, Inflow uses ten 3s dropping intervals within a 200s window; these drops throttle Tor's congestion control, creating long inter-packet delays (IPDs) (silent-intervals) observable at the server. The detector flags a flow once a sufficient number of IPDs exceed 2.5 s and their gaps align with the key. Iacovazzi et al. [20] reported 96% TPR and 0% FPR with four matching silent-intervals. With Waterfall, we bind an embedding capsule to the client's control flow and a detection capsule to the server's data flow to perform the watermarking (E01). We also employed a more tightened detection rule so that a flow is only declared watermarked if at least 8 of those 10 silent-interval positions are present instead of the 4-out-of-10 test that Inflow's authors used. Even under this stricter test, Waterfall attains 97% TPR and 0% FPR (Table 1, "File Download"), reproducing the original result within one percentage point.

RAINBOW operates differently, embedding the watermark by delaying each packet according to the cumulative sum of a sequence of $\pm a$ms amplitudes that constitute the key. For example, $[+10, +10, +10, -10, +10]$ yields delays of $[10, 20, 30, 20, 30]$ ms. The embedder records baseline IPDs, applies the pattern, and provides baseline IPDs plus the key to the detector, which correlates the induced IPD differences with the key via cosine similarity. Houmansadr et al. [18] reported a crossover error rate (COER) of 10^{-6} with fewer than 400 packets on replayed SSH traces, implying near-perfect TPR and FPR. Using Waterfall, we replay CAIDA-2019 SSH timings between a campus host and a remote node, apply the delay pattern in an embedding capsule, and stream IPDs and keys to the detector via `WaterfallCommunication`. Across four amplitudes (10, 5, 2, 1 ms) and a 1000-IPD key (E20–E23), we observe COER = 0 for all amplitudes once the key length reaches 50 packets, with 100% TPR and 0% FPR under both false-positive tests (Figure 6).

2. Discovering hidden design or network assumptions: Running legacy schemes in Waterfall, under realistic workloads exposed two fragile assumptions that are not apparent from the original papers and that Waterfall helped us diagnose and patch.

SDNt [27] encodes each bit by shrinking the *advertised window* (AWND) within 0.8 s intervals split into two 0.4 s sub-intervals (first half=1, second half=0). A 24-bit key, repeated six times for error correction is written over 116s, and the detector extracts the bit-string by measuring throughput per sub-interval and applying a Hamming-distance threshold of seven. Although the original paper reports 100% TPR / 0% FPR on VPN/SSH and 95% / 0% on Tor, replaying SDNt in Waterfall produced only 2% TPR (and comparable FPR) (E11). System logs showed that real-world links often have a *congestion window* (CWND) smaller than AWND, so shrinking AWND no longer throttles traffic [40]. Thus, SDNt implicitly relies on AWND ≤ CWND. Waterfall helped make the mismatch clear; §4.3 shows that a more sophisticated approach with traffic measurements and adaptive offsets restores TPR to 100% on stable VPN (E12) above 60% on Tor (E19) while keeping FPR < 2%.

Inflow revealed a different fragility. Its detector treats any IPD > 2.5 s as a silent interval—an assumption that holds for bulk downloads but breaks for streaming media. On MP4 downloads (E03) and HLS live streams (E02), natural buffering introduces similar gaps: TPR drops to 54%–50% and FPR_2 rises to 0.8%–0.9%. Because Waterfall separates the timing extraction (online) from the reconstruction and matching (offline), we swapped the gap-pattern test for a coarse large-IPD-alignment matcher that counts how many key elements match with silent intervals, as described in Appendix D.1. Table 1 shows the results. Setting the alignment threshold to six gaps for MP4 and eight for HLS lifts TPR to 78% and 84%, respectively, while keeping FPR below or at 1%. This suggests that our interval matcher absorbs the timing jitter of streaming media with many natural silent intervals better than the gap-pattern detector proposed by Inflow. The watermarking algorithm itself remains unchanged; only the reconstruction logic is replaced, showing how Waterfall helps pinpoint and remedy these weaknesses.

3. Porting schemes to new workloads & timing regimes: We also show that a Waterfall configuration can transplant a watermark in a new application and push its timing granularity beyond what prior work evaluated, without changing the watermark's core logic.

We first demonstrate portability by applying RAINBOW to live VoIP calls instead of the SSH-like traces used in the original paper. Two soft-phone clients exchange SRTP traffic; an embedding capsule delays packets according to the cumulative pattern, and a detection capsule extracts the key from IPDs (E24–E26). As Table 2 shows, for amplitudes of 5, 2, and even 1 ms all achieve 100% TPR and 0% FPR at a key length of 500 packets. This requires no algorithmic changes, only rebinding the capsules to the VoIP stream.

The same experiment validates sub-millisecond precision. With a 1 ms amplitude, smaller than typical WAN jitter, Waterfall consistently injects and detects the delays. Across 200 trials per amplitude the crossover error rate remains zero, confirming that Waterfall

preserves timing accuracy down to the 1 ms level and can therefore evaluate schemes that rely on sub-millisecond timing modulations.

Portability also revealed interesting limitations: For demonstration, we applied RAINBOW with a conservative 10 ms amplitude to traffic replayed through a Tor SOCKS/TCP path to test whether the same timing-based detector transfers to a modern anonymity network (E27). Unlike the VoIP setting, detection did not carry over. Under the original cosine-based detector, the true key beat all wrong keys in only 21/100 runs. Using the detector threshold that approximately equalizes false negatives and wrong-key false matches, we obtain 26% TPR and 15% FPR_2. We then allowed an exhaustive two-dimensional alignment search over up to 100 sender-side and 100 detector-side IPD prefixes to account for startup misalignment, but this still did not recover robust separability: the mean best true-key cosine-correlation score was 0.0852, while the mean best wrong-key score was 0.1046, and the best true-key score exceeded the best wrong-key score in only 20/100 runs.

The trace statistics indicate that this failure is not merely a synchronization issue. Relative to the sender-side sequence, the detector-side Tor trace was about 16% shorter on average, short IPDs below 10 ms nearly disappeared (7.2% at the sender versus 0.9% at the detector), and the median absolute IPD distortion rose to 82.0 ms, far above the 10 ms watermark amplitude. This experiment therefore acts as a stress test of RAINBOW's implicit timing assumptions: its detector relies on the watermark's temporal structure surviving transmission well enough for cosine-based matching, but Tor's stream-level buffering, coalescing, and repacketization reshape the detector-side IPD sequence, breaking that assumption.

4. Evaluating related passive baselines: Waterfall also allows us to implement and evaluate passive baselines alongside active watermarking schemes. To contextualize the benefit of RAINBOW's active perturbation, we implement a passive RAINBOW-style matching baseline in which no watermark is embedded. Instead of correlating an IPD difference signal with a known watermark key, we directly compare upstream and downstream IPD sequences. Concretely, for each downstream flow, we score the true upstream peer and a set of 10 non-matching upstream candidates using cosine similarity over z-score-normalized IPD windows of length 500, which is equivalent to Pearson correlation. While it is not an exhaustive model of passive traffic analysis, and more sophisticated passive methods exist, this baseline captures a common class of passive timing-correlation attacks based on inter-packet delays. Thus it provides a representative reference point for contrasting passive correlation with active watermarking under identical conditions.

On replayed CAIDA SSH traces, passive matching is already highly effective: across 200 positive and 2000 negative pairs, the true-pair and false-pair scores are cleanly separated, yielding 100% TPR at no observed false positives in our sample (E28). This is plausible because the SSH replay traces retain highly distinctive timing structure that remains strongly correlated across observation points. In contrast, passive matching on SIP/VoIP traces is much weaker: true and false match scores overlap substantially, and at approximately 1% FPR, TPR is only about 0.5% (E29). A likely reason is that the G.711 μ-law codec emits one SRTP packet every 20 ms, producing a highly periodic 50 packets/s stream, so many unrelated calls share similar timing patterns. Thus, Waterfall not only reproduces RAINBOW's active detector, but also helps compare it against a closely related passive correlation baseline, showing that the benefit of active perturbation depends strongly on the workload.

Table 3: Different dropping intervals for INFLOW on Tor.

Dropping Interval (s)	Detection Window (s)	TPR (%)	FPR_1(%)	FPR_2(%)	Duration (s)
3 (E01)	2.5	97	0	0	200
2 (E04)	1.7	100	0	0	133
1 (E05)	0.8	100	0	0.4	66
0.5 (E06)	0.4	97	0.4	0.9	33

Takeaway: Waterfall advances experimental methodology for watermarking by providing one framework that: (*i*) faithfully reproduces prior results of existing watermarking schemes, (*ii*) exposes unarticulated network or detector assumptions, (*iii*) is able to port existing schemes to new workloads and, (*iv*) contextualizes the benefit of active perturbation by enabling direct comparison against closely related passive correlation baselines.

4.3 Enhancing Existing Watermarking Schemes

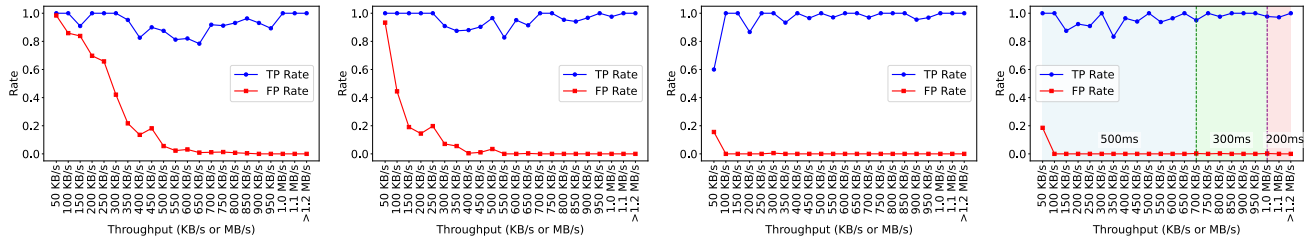
Waterfall is not limited to reproducing prior schemes. While most schemes rely on static workflows and parameters that are fixed regardless of network conditions, Waterfall's monitors and per-capsule APIs enable us to *enhance* legacy schemes so they dynamically adapt to runtime conditions along two axes:

- *Lower disruption*, i.e., the amount of interference introduced by the embedder during watermark insertion, such as packet drops, added latency, or throughput degradation. Lower disruption yields a stealthier watermark with less impact on transmission quality, without necessarily changing the detection algorithm.
- *Higher robustness*, i.e., the likelihood that the watermark remains detectable by the decoding party when real-world conditions deviate from the lab setting: higher jitter, different congestion windows, variable cross-traffic, or active countermeasures. Higher robustness means the detector maintains high TPR and low FPR across a broader range of network conditions.

To show Waterfall's adaptive capabilities, we present: (i) a low-disruption INFLOW variant that shortens drop bursts while preserving accuracy, and (ii) a bandwidth–delay-aware SDNt variant that restores effectiveness in networks where the original design fails.

1. Low-disruption INFLOW: Original INFLOW drops 3 s bursts over a 200 s window, which is effective but highly disruptive for short or latency-sensitive flows. To explore whether Waterfall can reduce this disruption by shortening the watermarking duration while preserving the TPR and FPR reported in the original paper, we progressively squeezed the dropping interval down to 0.5s and retuned the detector (smaller window and a 10-packet threshold) to stay aligned with the watermark (see Table 3). Across 250 trials per setting (E04–E06), we find that the 3 s bursts are unnecessary: a 0.5 s interval still achieves 97% TPR with only a small FPR increase, while reducing disruption and the watermark duration to 33 s.

Nevertheless, shorter dropping intervals and detection windows tend to increase FPR. To assess noise levels and identify conditions



(a) 200ms drop interval (b) 300ms drop interval (c) 500ms drop interval (d) Self-adjusting drop interval with a 338ms avg. interval

Figure 7: True positive rate and false positive rate vs. throughput for Inflow and small drop intervals.

Table 4: Small-drop and adaptive Inflow on Tor.

Average Dropping Interval (ms)	TPR (%)	FPR_1 (%)	FPR_2 (%)
200 (E07)	91.5	-	12.1
300 (E08)	95.7	-	2.4
500 (E09)	98.3	-	0.2
338 (E10)	97.3	0.1	0.2

under which smaller intervals can still be reliably detected, we fix 66s flows (1 s key intervals) and evaluate 200/300/500 ms drops with matched detection windows (130/180/400 ms) determined experimentally (E07–E09). Since watermarking was always applied, we focused exclusively on FPR_2, the stricter and typically higher of the two false-positive measures (see Table 3). Figure 7 shows that for 200 ms and 300 ms, FPR_2 is high at low throughput and decreases as throughput increases, while TPR varies nonlinearly; 500 ms remains stable across throughput.

These findings led us to design a more effective and adaptive variant of Inflow that reduces disruption without sacrificing detectability. By profiling flows, we identified optimal dropping and detection intervals and enabled the embedder and detector capsules to synchronize them dynamically based on real-time throughput, rather than fixed parameters. The similarity in data flow properties on the client and server sides enables consistent watermarking behavior across both ends. Figures 7a, 7b, and 7c show that the 200ms interval becomes stable above 1MB/s, while the 300ms interval stabilizes at 700KB/s. The 500ms interval performs consistently across all throughput levels. Based on these results, we implemented an optimized variant of Inflow (Figure 7d) that achieves performance comparable to the 500ms interval while reducing the average dropping interval by 32%, down to just 338ms (E10) (see Table 4).

2. BDP-aware SDNt. As mentioned in §4.2, the SDNt watermark [27] targets proxy-mediated traffic (VPN, SSH, Tor) by modulating TCP's advertised window (AWND) on packets from the proxy, thereby throttling the sender's effective sending rate. The detector decodes bits from receiver-side throughput changes. The original SDNt design uses a fixed AWND modulation pattern, implicitly assuming that AWND is the active bottleneck. However, as shown in §4.2, this often fails in modern deployments because the sender is frequently congestion-window limited (CWND < AWND). In that case, reducing AWND has no effect as long as AWND ≥ CWND, and the watermark vanishes. In short, SDNt requires that the embedder drive AWND below the sender's effective congestion-limited window so that AWND becomes throughput-limiting.

To make this scheme more robust, we designed an SDNt variant that dynamically adjusts AWND based on real-time estimates of the Bandwidth–Delay Product (BDP). We compute BDP_{est} = throughput × RTT, which approximates the in-flight data needed to saturate the path. During embedding, we choose $\text{AWND}_{\text{adjusted}}$ as a fraction of this estimate, so that AWND reliably constrains the sender, regardless of CWND fluctuations:

$$\text{AWND}_{\text{adjusted}} = \frac{BDP_{est} \times \text{ratio}}{2^S}$$

where ratio < 1 controls throttling strength and S is the TCP Window Scale (RFC 7323) [7] extracted by the capsule manager from the TCP handshake and passed to the capsule. In Waterfall, a capsule binds both data and control flow to estimate these quantities: throughput is computed once per second, and RTT is estimated from ACK timing averaging five samples. We use this estimate to set $\text{AWND}_{\text{adjusted}}$ during the watermarking phase and then release the window afterward.

On a controlled VPN setup, with the original paper's parameters, our BDP-based method achieves 100% TPR, 0% FPR_1, and 0% FPR_2 with a stricter Hamming threshold of 5 vs. 7; (E12). Interestingly, shorter intervals (0.4/0.2/0.1 s) provide the same results (E13–E15). Subsequently, we tested our method on Tor, comparing it to the original AWND-based method. To match the original setup, we disabled Conflux and evaluated detection at multiple extraction offsets (20ms, 50ms, 100ms, 250ms, 500ms) to account for Tor's natural variability. Under the original parameters, i.e., 0.8s modulation intervals and an AWND ratio of $\frac{3}{4}$, we were unable to detect any watermark, using either method. This reflects the high jitter and buffering variability present in today's Tor network and suggests that the original paper omitted key contextual information. We then tested two alternate configurations, varying the modulation interval and AWND ratio (see Table 5). Our BDP-based SDNt variant outperforms the original AWND-based method across both configurations. We achieved a TPR of 29% with a 1.3% FPR_2 in Conf. 1, and a significantly higher TPR of 65% with a minimal FPR_2 of 0.5% in Conf. 2. However, a TPR of 65% remains modest for practical applications, highlighting the inherent challenges of performing server-side watermarking attacks in Tor, primarily due to high disruption and buffering. In

Table 5: Comparison of AWND- and BDP-based SDNt methods for Tor without Conflux.

Config.	Method	TPR %	FPR_1 %	FPR_2 %
Conf 1: 800ms, $\frac{1}{2}$	AWND-based (E16)	3	1.6	1.9
Conf 1: 800ms, $\frac{1}{2}$	BDP-based (ours) (E18)	29	1.6	1.3
Conf 2: 1600ms, $\frac{1}{4}$	AWND-based (E17)	7	1.1	1.6
Conf 2: 1600ms, $\frac{1}{4}$	BDP-based (ours) (E19)	65	1.1	0.5

the next section, we show how the results compare against Tor with Conflux, which reveals a significant performance increase.

Takeaway: Waterfall's live traffic introspection and per-capsule parameter APIs let practitioners *reduce disruption* (short-burst Inflow) and *restore robustness* (BDP-aware SDNt) without touching the watermark algorithms, enabling principled, automated improvement beyond replication.

4.4 Watermarking New Anonymization Protocols

In this section, we demonstrate how Waterfall can serve as a testbed for evaluating watermarking attacks on emerging anonymity protocols. As a case study, we target *Conflux*, a recent traffic-splitting extension to Tor, now enabled by default. It improves performance by routing traffic across two independent circuits simultaneously, rather than through a single one, improving throughput and latency. While beneficial for users, Conflux changes the watermarking surface: splitting a connection into two legs complicates both embedding and detection, but the increased performance and reduced buffering may make watermarks more resilient and easier to extract. To explore these dynamics, we examine two complementary scenarios: (*i*) *multi-flow watermark embedding*, here the adversary embeds across both legs and detects downstream; and (*ii*) *multi-flow watermark detection*, where embedding happens upstream and detection aggregates one or both legs near the client. Our goal is to leverage Waterfall to assess whether Conflux strengthens or weakens Tor's resistance to watermarking-based attacks.

1. Multi-flow watermark embedding: Here, watermarking is embedded at the client across both Conflux legs but detected on a single server-side flow. Since Tor onion services do not yet support Conflux, we combined standard Tor with the Inflow technique for our experiments. A naive strategy, i.e., dropping packets on only one leg, fails because Conflux shifts load to the other leg. Thus, the attack surface for one-leg embedding positions is reduced. To keep the watermark visible, we bind both legs to one capsule and apply synchronized drops on both. Table 9 in Appendix C.2 reports results across dropping intervals (E30–E33). Compared to Tor onion services, TPR decreases (for reference, see Table 3), but FPR values remain negligible, and watermark duration shrinks with the dropping interval. The lower TPRs may be due to increased stability in the presence of Conflux, resulting in a reduced response to disruptions.

2. Multi-flow watermark detection. In this scenario, the watermark is embedded in a single server-side flow but may arrive at the client split across Conflux legs. We use our BDP-aware SDNt variant with the best parameters from §4.3 (1.6 s interval, ratio $\frac{1}{4}$; E34). We evaluate two configurations: (*i*) both Conflux legs are received and processed together by the same capsule, and (*ii*) each leg is observed independently. Merging both legs before detection yields 87.6% TPR, representing an improvement of 22% over the same attack without Conflux. This increase suggests that Conflux's reduced buffering and higher throughput make the watermark easier to detect. Interestingly, in 90% of cases where a watermark is found in the merged flow, it is also reconstructable on both individual legs. In 7.3% of cases, it appears in only one leg, and in 2.7%, it is detectable only after merging. This behavior highlights an expanded attack surface resulting from the geographical dispersion of Conflux legs, giving adversaries greater flexibility. A watermark can now be extracted from either individual leg or from the combined flow, depending on what vantage points are available. This improves coverage, as more vantage points become viable for detection, and it shows robustness, since the watermark remains detectable even after the transformations introduced by Conflux's traffic-splitting behavior.

Takeaway: Waterfall enables quick adaptation of watermarking schemes to new anonymization protocols like Conflux, revealing that Conflux affects watermarking differently depending on attacker vantage point: (*i*) it can hinder single-leg attacks, but (*ii*) it can facilitate attacks that can observe, coordinate across, or combine both legs.

4.5 Defending Against Watermarking Attacks

While Waterfall primarily targets evaluation of watermarking attacks, it also supports building and validating defenses. We demonstrate two standard strategies from prior work: *watermark presence detection* [24, 25, 29] and *watermark removal* [24, 52]. We focus on RAINBOW, a popular watermarking scheme for which detection and removal techniques have been studied [25, 29, 52].

1. Watermark presence detection: We replicate the Known-Flow Attack (KFA) detector introduced by Lin et al. [25]. This method assumes that the receiver has access to the unmodified IPD sequence, allowing it to flag deviations caused by a watermark. To set up this testbed, we use VoIP traffic carried via SIP signaling. A caller initiates a call to a callee through a SIP server, and we place Waterfall instances at each endpoint and at the server. The SIP server applies RAINBOW watermarking into the SRTP traffic using an active embedding capsule, while the caller sends the callee periodic unwatermarked IPD beacons every 200 packets as a baseline reference. After 200 trials with a 1ms watermark amplitude and 200 control trials without watermarking, we detected the watermark in all cases with no false alarms (E35).

2. Watermark removal: Next, we demonstrate a lightweight removal defense inspired by DeMarking [52]. DeMarking trains a neural converter to transform an input IPD sequence into a perturbed sequence that erases the watermark. Instead of training a neural converter, we take advantage of the fact that our VoIP codec (G.711 ulaw) emits one SRTP packet every 20ms resulting in a perfectly periodic 50 packets/second stream. Consequently, our simplified defense enforces constant 20ms spacing on the SIP server. Specifically, the SIP server buffers two incoming RTP packets in a queue. Then, it forwards packets at fixed 20ms intervals, regardless of inter-arrival variance or embedded watermarks (E36). Table 8

shows that the detector's true and false positive rates converge, indicating that the constant-delay scheduler masks the watermark's timing signal. Compared to the original performance without defense (see Table 2), detection is neutralized across all watermark lengths, rendering the attack ineffective.

3. Attacker adaptation. To test whether the defense remains effective against an adaptive attacker, we implement a modified RAINBOW variant that randomizes the delay amplitude. Concretely, instead of using one fixed amplitude, the embedder samples a bounded amplitude between 1 and 5 ms for each key element while preserving the original key structure and the same detection pipeline. This is a stronger attacker model in the sense that the attacker is no longer constrained to a single fixed delay amplitude, but can vary the amplitude across key elements, removing a deterministic pattern that a timing defense could otherwise exploit. Without defense, the randomized-amplitude variant remains effective on periodic VoIP traffic, but is less stable than the fixed-amplitude baseline for short watermark lengths, as shown in Table 7 (E37) in Appendix C.1. Thus, the modified attack remains viable in the undefended setting while representing a more challenging and less idealized operating point than the originally evaluated fixed-amplitude case.

We then evaluate the same attack against the watermark removal defense. As before, the SIP server buffers incoming RTP packets and forwards them at fixed 20 ms intervals. Under this defense, the randomized-amplitude attack collapses to near-chance performance across all watermark lengths, as shown in Table 8 (E39) in Appendix C.1. Thus, even after attacker-side adaptation, the constant-rate pacing still suppresses the timing signal. More broadly, this experiment illustrates how Waterfall supports systematic attacker–defender evaluation. By expressing both the adapted attack and the defense within the same framework, we can assess how small changes in attacker strategy affect robustness under identical conditions rather than through isolated, scheme-specific implementations. Finally, we also tested the known-flow detector in the same VoIP setting. As expected, because it compares the clean and transformed timing traces directly, it continues to flag the presence of watermarking for the adapted variant (E38).

Takeaway: Waterfall not only supports extensive testing of watermarking schemes, but also enables the development and validation of defenses. By replicating detection and removal strategies, Waterfall offers a practical tool to harden anonymization systems against watermarking threats.

5 Limitations and Future Directions

Waterfall is scoped to packet- and flow-level watermarking schemes that operate from on-path vantage points and express their behavior through observable timing, loss, ordering, rate, or mutable-header perturbations. Our experiments assume that the relevant vantage points can observe and, where needed, modify packet streams, and that required coordination metadata such as keys, parameters, reference traces, or alignment information is available as assumed by the original attacker models. Waterfall also relies on developers to specify correct flow groups, policies, and parameters; it does not infer arbitrary protocol semantics or guarantee correct configuration in all deployments.

Techniques requiring endpoint compromise, decrypted payloads, application semantics, cryptographic secrets, or protocol-internal actions such as Tor-cell generation or padding-cell injection are outside this abstraction. Future work could combine Waterfall with endpoint instrumentation, protocol-specific hooks inside anonymity systems, or simulators such as Shadow [23] to evaluate watermarking and defenses that operate beyond observable packet flows.

6 Related Work

Prior research on traffic watermarking attacks has largely focused on timing-based schemes that embed patterns by modifying IPDs. Early work by Wang et al. [48] demonstrated stepping-stone detection via IPD manipulation, followed by many timing-based designs [15–17, 28, 33, 35, 36, 45, 47, 53]. RAINBOW [18] introduced a non-blind variant that requires access to the original IPD data at the detector. Wang et al. [46] further explored the approach by embedding watermarks through timed packet delays. Recent work has explored neural network-based timing watermarks [37]. Waterfall can reproduce such timing-based schemes, as shown by our RAINBOW implementation.

Beyond timing modulation, prior work has proposed packet-level watermarks that modify contents, sizes, or inject additional traffic [6, 10, 26, 44, 49]. For instance, in 2001, Wang et al. [49] introduced a watermarking framework which acts as an intrusion detection system. Other related works inject JavaScript into victim browsers, causing identifiable packet patterns [6, 44]. Some approaches additionally assume compromised endpoints, clients, or relays (e.g., cell injection) [19, 39]. Because Waterfall targets encrypted anonymity settings without endpoint compromise, content-based schemes under these assumptions are out of scope.

Additionally, rate-based schemes embed watermarks by modulating the traffic rate, for example by using a direct-sequence spread spectrum (DSSS) signal within the sender's traffic [51], or by adjusting the TCP congestion window on the server side to control the sending rate [27]. We demonstrate that Waterfall can replicate and evaluate rate-based watermarking schemes.

Finally, some techniques rely on deliberate packet drops to induce recognizable traffic patterns by exploiting TCP's congestion control and retransmission mechanisms. DropWat [21] selectively drops server-to-client packets to trigger extended silent intervals caused by retransmissions. We show that Waterfall can effectively support dropping-based methods by reproducing a more recent scheme presented in Inflow [20].

7 Conclusion

We present Waterfall, a capsule-based framework for the development, deployment, and evaluation of network watermarking schemes in anonymous communication systems. Our framework not only enhances the ability to embed and detect watermarks effectively but also provides valuable insights into the resilience of anonymous communication systems. We show how analyzing network traffic properties can guide the design of more effective watermarking strategies. Additionally, we demonstrate how watermarking schemes can be adapted to emerging network protocols. Finally, we show that Waterfall also supports the development of defenses against watermarking attacks.

Acknowledgments

The authors used generative AI-based tools to revise the text, improve flow, and correct typos, grammatical errors, and awkward phrasing. This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972, and NLnet Foundation (Project number: 2024-12-326), by Portuguese national funds through Fundação para a Ciência e a Tecnologia, I.P. (FCT) under projects UID/50021/2025 (DOI: https://doi.org/10.54499/UID/50021/2025) and UID/PRR/50021/2025 (DOI: https://doi.org/10.54499/UID/PRR/50021/2025), by IAPMEI under grant C6632206063-00466847 (PT Smart Retail), by the Smart Networks and Services Joint Undertaking (SNS JU) under the European Union's Horizon Europe research and innovation programme through the ORIGAMI (GA#101139270) and PAISES-6G (GA#101292896) projects, and by the CoEvolution project under the European Union's Horizon Europe research and innovation programme (GA#101168560).

References

[1] 2024. Netfilterqueue. https://github.com/oremanj/python-netfilterqueue.
[2] 2025. Dedicated Server, Cloud, Storage & Hosting — hetzner.com. https://www.hetzner.com.
[3] 2025. pcap package - github.com/google/gopacket/pcap - Go Packages — pkg.go.dev. https://pkg.go.dev/github.com/google/gopacket/pcap. Accessed 16-01-2025.
[4] Gunes Acar, Marc Juarez, and individual contributors. 2023. tor-browser-selenium - Tor Browser automation with Selenium. https://github.com/webfp/tor-browser-selenium.
[5] Mashael AlSabah, Kevin Bauer, Tariq Elahi, and Ian Goldberg. 2013. The Path Less Travelled: Overcoming Tor's Bottlenecks with Traffic Splitting. In *Privacy Enhancing Technologies*, Emiliano De Cristofaro and Matthew Wright (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 143–163.
[6] Daniel Arp, Fabian Yamaguchi, and Konrad Rieck. 2015. Torben: A Practical Side-Channel Attack for Deanonymizing Tor Communication. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (Singapore, Republic of Singapore) *(ASIA CCS '15)*. Association for Computing Machinery, New York, NY, USA, 597–602. https://doi.org/10.1145/2714576.2714627
[7] David Borman, Robert T. Braden, Van Jacobson, and Richard Scheffenegger. 2014. TCP Extensions for High Performance. RFC 7323. https://doi.org/10.17487/RFC7323
[8] Canonical Ltd. 2024. Ubuntu 24.04 LTS "Noble Numbat". https://releases.ubuntu.com/24.04/ Long-Term Support release.
[9] Ronald Deibert, John Palfrey, Rafal Rohozinski, and Jonathan L. Zittrain (Eds.). 2010. *Access Controlled: The Shaping of Power, Rights, and Rule in Cyberspace*. The MIT Press, Cambridge, MA. https://direct.mit.edu/books/oa-edited-volume/1872/Access-ControlledThe-Shaping-of-Power-Rights-and
[10] Huijuan Deng, Xingming Sun, Baowei Wang, and Yuanfu Cao. 2009. Selective forwarding attack detection using watermark in WSNs. In *2009 ISECS International Colloquium on Computing, Communication, Control, and Management*, Vol. 3. 109–113. https://doi.org/10.1109/CCCM.2009.5268016
[11] Roger Dingledine, Nick Mathewson, Paul F Syverson, et al. 2004. Tor: The second-generation onion router.. In *USENIX security symposium*, Vol. 4. 303–320.
[12] Matthew Edman and Bülent Yener. 2009. On anonymity in an electronic society: A survey of anonymous communication systems. *ACM Comput. Surv.* 42, 1, Article 5 (Dec. 2009), 35 pages. https://doi.org/10.1145/1592451.1592456
[13] ESnet and Lawrence Berkeley National Laboratory. 2026. iperf3. https://software.es.net/iperf/
[14] Alfred E. Heggestad and Contributors. 2025. Baresip: Modular SIP User-Agent with Audio and Video Support. https://github.com/baresip/baresip
[15] Amir Houmansadr and Nikita Borisov. 2011. SWIRL: A Scalable Watermark to Detect Correlated Network Flows.. In *NDSS*. Citeseer.
[16] Amir Houmansadr and Nikita Borisov. 2013. BotMosaic: Collaborative network watermark for the detection of IRC-based botnets. *Journal of Systems and Software* 86, 3 (2013), 707–715. https://doi.org/10.1016/j.jss.2012.11.005
[17] Amir Houmansadr and Nikita Borisov. 2013. The need for flow fingerprints to link correlated network flows. In *Privacy Enhancing Technologies: 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings 13*. Springer, 205–224.
[18] Amir Houmansadr, Negar Kiyavash, and Nikita Borisov. 2009. RAINBOW: A robust and invisible non-blind watermark for network flows.. In *NDSS*, Vol. 47. Citeseer, 406–422.
[19] Alfonso Iacovazzi, Daniel Frassinelli, and Yuval Elovici. 2019. The DUSTER Attack: Tor Onion Service Attribution Based on Flow Watermarking with Track Hiding. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. USENIX Association, Chaoyang District, Beijing, 213–225. https://www.usenix.org/conference/raid2019/presentation/iacovazzi
[20] Alfonso Iacovazzi, Sanat Sarda, and Yuval Elovici. 2018. Inflow: Inverse Network Flow Watermarking for Detecting Hidden Servers. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. 747–755. https://doi.org/10.1109/INFOCOM.2018.8486375
[21] Alfonso Iacovazzi, Sanat Sarda, Daniel Frassinelli, and Yuval Elovici. 2017. DropWat: An Invisible Network Flow Watermark for Data Exfiltration Traceback. *IEEE Transactions on Information Forensics and Security* PP (05 2017). https://doi.org/10.1109/TIFS.2017.2779113
[22] Inferno Nettverk A/S. 2026. Dante: A Free SOCKS Server. https://www.inet.no/dante/.
[23] Rob Jansen, Jim Newsome, and Ryan Wails. 2022. Co-opting Linux Processes for High-Performance Network Simulation. In *USENIX Annual Technical Conference*. See also https://netsim-atc2022.github.io.
[24] Negar Kiyavash, Amir Houmansadr, and Nikita Borisov. 2008. Multi-flow attacks against network flow watermarking schemes. In *Proceedings of the 17th Conference on Security Symposium* (San Jose, CA) *(SS'08)*. USENIX Association, USA, 307–320.
[25] Zi Lin and Nicholas Hopper. 2012. New attacks on timing-based network flow watermarks. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Bellevue, WA) *(Security'12)*. USENIX Association, USA, 20.
[26] Zhen Ling, Xinwen Fu, Weijia Jia, Wei Yu, and Dong Xuan. 2011. A novel packet size based covert channel attack against anonymizer. In *2011 Proceedings IEEE INFOCOM*. 186–190. https://doi.org/10.1109/INFCOM.2011.5934988
[27] Zhen Ling, Junzhou Luo, Danni Xu, Ming Yang, and Xinwen Fu. 2019. Novel and Practical SDN-based Traceback Technique for Malicious Traffic over Anonymous Networks. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. 1180–1188. https://doi.org/10.1109/INFOCOM.2019.8737586
[28] Junzhou Luo, Xiaogang Wang, and Ming Yang. 2012. An interval centroid based spread spectrum watermarking scheme for multi-flow traceback. *Journal of Network and Computer Applications* 35, 1 (2012), 60–71. https://doi.org/10.1016/j.jnca.2011.03.003 Collaborative Computing and Applications.
[29] Xiapu Luo, Peng Zhou, Junjie Zhang, Roberto Perdisci, Wenke Lee, and Rocky K. C. Chang. 2011. Exposing invisible timing-based traffic watermarks with BACKLIT. In *Proceedings of the 27th Annual Computer Security Applications Conference* (Orlando, Florida, USA) *(ACSAC '11)*. Association for Computing Machinery, New York, NY, USA, 197–206. https://doi.org/10.1145/2076732.2076760
[30] Nick Mathewson and Mike Perry. 2018. *Towards Side Channel Analysis of Datagram Tor vs Current Tor*. Technical Report 2018-11-002. The Tor Project. https://research.torproject.org/techreports/side-channel-analysis-2018-11-27.pdf
[31] Mozilla Foundation and Mozilla contributors. 2026. Firefox Browser. https://www.mozilla.org/firefox/.
[32] NDR. 2024. Investigations in the so-called darknet: Law enforcement agencies undermine Tor anonymisation — ndr.de. https://www.ndr.de/fernsehen/sendungen/panorama/aktuell/Investigations-in-the-so-called-darknet-Law-enforcement-agencies-undermine-Tor-anonymisation,toreng100.html. Accessed 02-01-2025.
[33] Pai Peng, Peng Ning, D.S. Reeves, and Xinyuan Wang. 2005. Active timing-based correlation of perturbed traffic flows with chaff packets. In *25th IEEE International Conference on Distributed Computing Systems Workshops*. 107–113. https://doi.org/10.1109/ICDCSW.2005.30
[34] Mike Perry. 2022. Congestion control arrives in tor 0.4.7-stable!: Tor project. https://blog.torproject.org/congestion-contr0l047
[35] Y. J. Pyun, Y. H. Park, X. Wang, D. S. Reeves, and P. Ning. 2007. Tracing Traffic through Intermediate Hosts that Repacketize Flows. In *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications*. 634–642. https://doi.org/10.1109/INFCOM.2007.80
[36] Fatemeh Rezaei and Amir Houmansadr. 2017. TagIt: Tagging Network Flows using Blind Fingerprints. *Proc. Priv. Enhancing Technol.* 2017, 4 (2017), 290–307.
[37] Fatemeh Rezaei and Amir Houmansadr. 2021. FINN: fingerprinting network flows using neural networks. In *Annual Computer Security Applications Conference*. 1011–1024.
[38] Neil M. Richards. 2013. The Dangers of Surveillance. *Harvard Law Review* 126, 7 (2013), 1934–1965. https://harvardlawreview.org/print/vol-126/the-dangers-of-surveillance/
[39] Florentin Rochet and Olivier Pereira. 2018. Dropping on the Edge: Flexibility and Traffic Confirmation in Onion Routing Protocols. *Proceedings on Privacy Enhancing Technologies* 2018, 2 (2018), 27–46. https://doi.org/10.1515/popets-2018-0011
[40] Jan Rüth and Oliver Hohlfeld. 2018. Demystifying TCP Initial Window Configurations of Content Distribution Networks. In *2018 Network Traffic Measurement and Analysis Conference (TMA)*. 1–8. https://doi.org/10.23919/TMA.2018.8506549

[41] Sangoma Technologies Corporation. 2025. Asterisk: The Open Source Telephony Project. https://www.asterisk.org/

[42] Hemant Sengar, Zhen Ren, Haining Wang, Duminda Wijesekera, and Sushil Jajodia. 2010. Tracking Skype VoIP Calls Over The Internet. In *2010 Proceedings IEEE INFOCOM*. 1–5. https://doi.org/10.1109/INFCOM.2010.5462266

[43] Fatemeh Shirazi, Milivoj Simeonovski, Muhammad Rizwan Asghar, Michael Backes, and Claudia Diaz. 2018. A Survey on Routing in Anonymous Communication Protocols. 51, 3, Article 51 (June 2018), 39 pages. https://doi.org/10.1145/3182658

[44] Natalija Vlajic and Daniel Brown. 2023. Poster: Novel Client-Side Watermarking Technique for Tor User De-Anonymization. In *Proceedings of the 2023 ACM on Internet Measurement Conference* (Montreal QC, Canada) *(IMC '23)*. Association for Computing Machinery, New York, NY, USA, 720–721. https://doi.org/10.1145/3618257.3624997

[45] Xinyuan Wang, Shiping Chen, and Sushil Jajodia. 2005. Tracking anonymous peer-to-peer VoIP calls on the internet. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (Alexandria, VA, USA) *(CCS '05)*. Association for Computing Machinery, New York, NY, USA, 81–91. https://doi.org/10.1145/1102120.1102133

[46] Xinyuan Wang, Shiping Chen, and Sushil Jajodia. 2007. Network Flow Watermarking Attack on Low-Latency Anonymous Communication Systems. In *2007 IEEE Symposium on Security and Privacy (SP '07)*. 116–130. https://doi.org/10.1109/SP.2007.30

[47] Xiaogang Wang, Junzhou Luo, and Ming Yang. 2010. A Double Interval Centroid-Based Watermark for network flow traceback. In *The 2010 14th International Conference on Computer Supported Cooperative Work in Design*. 146–151. https://doi.org/10.1109/CSCWD.2010.5471985

[48] Xinyuan Wang and Douglas S. Reeves. 2003. Robust correlation of encrypted attack traffic through stepping stones by manipulation of interpacket delays *(CCS '03)*. Association for Computing Machinery, New York, NY, USA, 20–29. https://doi.org/10.1145/948109.948115

[49] Xinyuan Wang, Douglas S Reeves, S Felix Wu, and Jim Yuill. 2001. Sleepy watermark tracing: An active network-based intrusion response framework. In *Trusted Information: The New Decade Challenge 16*. Springer, 369–384.

[50] James Yonan and the OpenVPN Project. 2025. OpenVPN. https://openvpn.net/.

[51] Wei Yu, Xinwen Fu, Steve Graham, Dong Xuan, and Wei Zhao. 2007. DSSS-Based Flow Marking Technique for Invisible Traceback. In *2007 IEEE Symposium on Security and Privacy (SP '07)*. 18–32. https://doi.org/10.1109/SP.2007.14

[52] Yali Yuan, Jian Ge, and Guang Cheng. 2025. DeMarking: A defense for network flow watermarking in real-time. *Comput. Secur.* 152, C (April 2025), 11 pages. https://doi.org/10.1016/j.cose.2025.104355

[53] Ali Zand, Giovanni Vigna, Richard Kemmerer, and Christopher Kruegel. 2014. Rippler: Delay injection for service dependency detection. In *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*. 2157–2165. https://doi.org/10.1109/INFOCOM.2014.6848158

Inflow Embedding Adjustment

```
...
-- Get the control flow
local control_flow = wf.get_flows().ControlFlow

-- Swap IPs and ports to get the data flow, going in the opposite direction
local data_flow = {
    src_ip = control_flow.dst_ip,
    dst_ip = control_flow.src_ip,
    src_port = control_flow.dst_port,
    dst_port = control_flow.src_port,
    protocol = control_flow.protocol
}

-- Bind additional flows and monitor throughput
wf.bind_flow("InflowEmbedding", "DataFlow", data_flow, function()
    wf.monitor_data_throughput("InflowEmbedding", 1 * wf.SECOND, "DataFlow", function(throughput)
        if throughput > 1000000 then
            drop_interval = 200
        elseif throughput > 700000 then
            drop_interval = 300
        else
            drop_interval = 500
        end
    end)
end)
...
```

Inflow Detection Adjustment

```
...
-- Monitor throughput and adjust DetectionWindow
wf.monitor_data_throughput("InflowDetection", 1 * wf.SECOND, "DataFlow", function(throughput)
    local new_detection_window
    if throughput > 1000000 then
        new_detection_window = 130
    elseif throughput > 700000 then
        new_detection_window = 180
    else
        new_detection_window = 400
    end
    -- Reconfigure detection with new window duration
    detector.set_duration(new_detection_window * wf.SECOND)
end)
...
```

Figure 8: Waterfall Scripts Adjustments.

A Ethics Considerations

We conduct a substantial portion of our experiments on the publicly available anonymous privacy platform Tor [11]. Importantly, we exclusively used our own data and did not interfere with the data or activities of other Tor users. Our watermarking techniques remain within the bandwidth and data limits of both Tor and our network connection, avoiding any behavior that resembles traffic flooding or overload. As a result, our experiments do not impact the performance or privacy of the Tor network or its users. We therefore see no ethical concerns raised by our methodology. We have also responsibly disclosed our findings to the Tor Project team.

B Additional Framework Details

B.1 Challenges in Reproducing Prior Schemes

The evaluation methodologies used for the three schemes vary significantly. For Inflow, false positives were assessed by attempting to detect watermark keys different from those embedded in the flows. In SDNt, experiments were conducted to detect watermarks in unwatermarked flows. Meanwhile, the authors of RAINBOW define the false positive rate as the probability of incorrectly identifying a non-watermarked flow as watermarked. Additionally, the authors of Inflow omitted critical details, such as the number of alternative keys tested per flow or their distances. Similarly, SDNt does not provide any information on the number or characteristics of keys used, making it difficult to interpret their results.

B.2 Adaptive Inflow Implementation

The basic implementation of Inflow presented in Figure 4 is very rigid using static parameters. However, Waterfall allows for dynamic adjustment based on real-time traffic properties. By incorporating monitoring functions, embedding and detection capsules can adapt their parameters to optimize watermarking effectiveness. In this example, by using the function `bind_flow` in the embedding capsule, we request the capsule manager to bind the data flow in addition to the control flow. This allows us to measure the data throughput by setting up a monitor. Based on the throughput we can adjust the dropping interval in the embedding capsule and the detection window in the detection capsule. These changes can be seen in Figure 8.

B.3 Waterfall Services

Table 6 summarizes the Waterfall runtime services used for measurement, perturbation, key management, coordination, and matching.

Table 6: List of Waterfall services.

Service	Explanation
`WaterfallMonitor`	Provides flow- and packet-derived real-time traffic measurements, enabling adaptive embeddings and robust detectors.
`WaterfallAction`	Applies traffic perturbations required by schemes with explicit scope (packet/flow/group) and bounds (time/window/packet budget).
`WaterfallKey`	Manages scheme-specific watermark keys across runs for reproducible experiments.
`WaterfallCommunication`	Implements control plane between Waterfall nodes for coordination, to distribute keys, synchronization timings, watermarking parameters, and to collect detector outcomes at scale.
`WaterfallMatching`	Allows scheme-specific reconstruction and matching between embedded and reconstructed candidate keys (e.g., thresholding, ECC, scoring) and records results for later offline analysis.

B.4 Relationship to Shadow

Waterfall is complementary to network simulators such as Shadow [23] rather than a replacement for them. Waterfall operates at the packet-perturbation level, providing fine-grained control and observation of packet characteristics needed for watermark embedding and detection. By contrast, Shadow primarily provides application-level interactions and event scheduling within a simulated network. As a result, Waterfall-style watermarking cannot currently be integrated into Shadow out of the box using existing interfaces alone. Supporting such integration would require exposing additional packet-level hooks for perturbation and observation. We therefore view deeper Waterfall–Shadow integration as an interesting direction for future work, but intentionally design Waterfall as a standalone framework for watermark analysis rather than as a network-wide simulator.

C Additional Evaluation Results

C.1 Additional Defense Results

Tables 7 and 8 report RAINBOW detection for the randomized-amplitude baseline without defenses and for the pacing defense, respectively.

Table 7: RAINBOW detection on periodic VoIP traffic for the randomized-amplitude attacker variant without defense.

Attack variant	WM length	TPR (%)	FPR_1 (%)	FPR_2 (%)
Randomized amplitude, no defense	50	87.5	13.1	11.9
	100	97.5	2.2	2.8
	200	99.5	0.5	0.5
	500	99.5	0.3	0.8

Table 8: RAINBOW detection: VoIP traffic, with defense.

	1 ms, with defense			Randomized amplitude, with defense		
WM length	TPR (%)	FPR_1 (%)	FPR_2 (%)	TPR (%)	FPR_1 (%)	FPR_2 (%)
50	48.5	52.9	49.9	49.0	52.6	49.9
100	53.1	42.2	51.1	52.5	43.8	50.0
200	53.6	42.9	49.5	51.0	47.0	51.7
500	53.6	42.3	53.0	50.5	45.5	53.6

C.2 Conflux Multi-flow Embedding Results

Table 9 reports the multi-flow Inflow embedding results for Conflux that are referenced in §4.4. The table shows that synchronized drops across both Conflux legs keep FPR at 0% while reducing watermark duration with shorter dropping intervals.

Table 9: True Positive and False Positive Rates at various dropping intervals for Inflow and Conflux.

Dropping Interval (s)	TPR (%)	FPR_1 (%)	FPR_2 (%)	Duration (s)
3 (E30)	96	0	0	200
2 (E31)	90	0	0	133
1 (E32)	91	0	0	66
0.5 (E33)	87	0	0	33

C.3 Runtime Overhead

We performed a stress test of Waterfall's runtime on a 104-core, 256 GB RAM server using `iperf3` traffic [13]. The `iperf3` client and server communicated over a 10 GbE internal path, so the offered baseline throughput was intentionally much higher than in our Tor experiments. This should therefore be interpreted as a runtime stress test rather than a Tor-like or public proxy workload. We varied the number of concurrent flows and instantiated one capsule per flow. Detection capsules remained off the forwarding path and only counted packets once per second. As shown in Table 10, detection throughput stayed close to the no-Waterfall baseline across the tested concurrency range, while CPU and memory usage increased moderately. For embedding, we used no-op capsules that intercepted and bound flows to the active embedding path but applied no perturbation. In this case, throughput was limited, staying around 0.58–0.73 Gbps. Although memory usage increased with concurrency to levels similar to detection, CPU usage was consistently higher, indicating that the dominant embedding overhead arises from routing packets through the active packet-interception path (`iptables+NFQUEUE`).

Table 10: Runtime characterization of Waterfall. The throughput columns report measurements with and without Waterfall active under the same traffic setup.

Kind	Flows / Capsules	With Waterfall	Without Waterfall	CPU	Mem.
Detection	1 / 1	4.233 Gbps	4.316 Gbps	110.67%	26 MB
Detection	10 / 10	9.007 Gbps	9.358 Gbps	176.19%	52 MB
Detection	50 / 50	9.353 Gbps	9.368 Gbps	252.73%	129 MB
Detection	100 / 100	9.335 Gbps	9.350 Gbps	255.29%	180 MB
Embedding	1 / 1	0.575 Gbps	4.316 Gbps	451.57%	26 MB
Embedding	10 / 10	0.588 Gbps	9.358 Gbps	478.02%	26 MB
Embedding	50 / 50	0.717 Gbps	9.368 Gbps	541.11%	103 MB
Embedding	100 / 100	0.734 Gbps	9.350 Gbps	580.03%	180 MB

C.4 False Positives under Mixed Background Traffic

Our main FPR_1 experiments use closed-world negatives, i.e., repeated executions of the same application without watermarking. To evaluate whether detector thresholds remain conservative under more realistic traffic diversity, we generate a mixed HTTPS background workload and route it through Tor. Each run combines automated multi-tab browsing over Tranco-derived sites with randomized link clicks and tab switches, while concurrent background actors generate overlapping downloads and uploads. More details are provided in Appendix D.3.

We study three detector-side scenarios on the same mixed Tor workload: (i) a client-side scenario, (ii) a server-side proxy scenario, and (iii) a server-side onion-service scenario. Among the schemes in our study, SDNt and Inflow are the two that can be meaningfully evaluated on Tor. We therefore choose the scheme according to where its detector naturally operates. SDNt is evaluated in the client-side scenario using the best Tor parameters from §4.3 (1.6 s interval, ratio $\frac{1}{4}$); here the detector observes the client's aggregated Tor connections (E40). Inflow is evaluated in the two server-side scenarios using the four dropping intervals from §4.3 (E41). In the server-side proxy scenario, the Inflow detector observes downstream, post-Tor data flows between the SOCKS proxy and Tor exit nodes; these flows are non-Tor and largely separated (E42). In the server-side onion-service scenario, detection is performed at an onion service with a local SOCKS proxy; here the detector again observes downstream data flows, but multiple activities are re-aggregated into a smaller number of longer-lived Tor flows. This separation lets us study not only traffic diversity, but also how the same workload appears as either aggregated or separated depending on detector vantage point.

In the two aggregated scenarios, false positives remain close to our earlier single-application baselines. We collect 250 samples in each scenario. In the client-side SDNt scenario, the false-positive evaluation yields an FPR of 1.32%, comparable to our earlier SDNt Tor results (Table 5). This is consistent with SDNt's design: it spreads the watermark over repeated throughput windows rather than relying on isolated timing events. Consequently, its false positives are less sensitive to the precise application mix, although still sensitive to transport/path conditions. In the server-side onion-service Inflow scenario, we measure no false positives for 3 s and 2 s intervals, 0.08% FPR1 for 1 s intervals, and 0.64% for 0.5 s intervals. These results are comparable to the single file-download baseline (Table 3) and are lower than for MP4 and HLS traffic (Table 1). A likely reason is that aggregation suppresses many short application-level gaps and therefore makes spurious silent intervals less likely.

The server-side proxy scenario is particularly informative. Over a two-hour run, the detector observes 3,108 candidate flows, far more than in the aggregated client-side or onion-service scenarios. This is because multi-tab browsing, downloads, and uploads generate many overlapping application-level connections, each visible separately at the proxy. Waterfall starts and completes one detection capsule per flow with a peak of 125 concurrent capsules. This operational behavior illustrates why realistic proxy-side evaluation is inherently multi-flow and why concurrent capsule execution and scalability is necessary in Waterfall. The overall Inflow FPR1 values in this scenario are 0%, 0%, 0.12%, and 0.60% for 3 s, 2 s, 1 s, and 0.5 s intervals, respectively. However, most of these flows are too short to plausibly support longer watermark windows: only 47% remain active for at least 30 seconds samples and 36% for at least 60 seconds. Thus, the proxy vantage point increases the candidate set while sharply reducing per-flow watermark viability. When conditioning only on flows long enough to plausibly support detection, the proxy-side FPR1 rises to 0.34% for 1 s intervals and 1.28% for 0.5 s intervals, showing higher false positive rates than for the single file-download baseline (Table 3). This shows that the FPR depends on traffic characteristics and that short flows can dilute the aggregate FPR rather than trigger matches themselves.

This experiment shows that Waterfall can evaluate false positives beyond closed-world negatives by deploying detectors on mixed, concurrent background traffic at different detector-side scenarios. The results show that realistic false-positive analysis must account not only for traffic diversity, but also for the flow granularity and flow viability induced by the detector vantage point.

D Additional Experimental Setup

D.1 Custom Inflow Large-IPD Alignment Rule

We repeat the Inflow experiment 250 times with (i) progressive MP4 video and (ii) live HLS streaming. Buffering in both formats injects natural silent intervals, reducing the TPR to 54% and 50% and raising the FPR_2 to 0.8% and 0.9%. However, we notice that we can get better results when changing the matching algorithm on the already extracted data. We introduce a coarser *large-IPD alignment* rule that tests whether m key elements exist in the extracted IPDs with a threshold Δ as follows:

$$\sum_{i=1}^{m} \mathbf{1}\Big(\min_{\hat{d}_j \in \hat{D}} \big| d_i - \hat{d}_j \big| \le \Delta\Big) \;\ge\; q$$

where $D = \{d_1, \ldots, d_m\}$ is the set of key-defined timestamps, $\hat{D} = \{\hat{d}_1, \ldots, \hat{d}_{\hat{m}}\}$ is the set of extracted long-IPD timestamps, Δ is the per-mark tolerance window, and $\mathbf{1}(\cdot)$ is the indicator function. To get comparable timestamps, we synchronize the start times of the embedding and detection capsules. If at least q marks are detected within the key, the flow is considered as watermarked. We set $\Delta = 2$s and $q = 6$ for the video traffic and $\Delta = 2$s and $q = 8$ for the live video stream. The discrepancy in q lies in the fact that the live stream has much more silent intervals and this requires less tolerance to keep the FPR low.

D.2 Experiment Inventory

Table 11 lists all experiments referenced in the paper. Slugs encode scheme, setting, and variant as SCHEME-SETTING-VARIANT. INF=Inflow, SDN=SDNt, RNB=RAINBOW.

Table 11: Experiments and number of runs. Each experiment is referenced by an ID E01–E42.

ID	Sec.	Slug	Runs
Inflow			
E01	4.2	INF-TOR-STD	250
E02	4.2	INF-TOR-STREAM	250
E03	4.2	INF-TOR-VIDEO	250
E04	4.3	INF-TOR-INT2	250
E05	4.3	INF-TOR-INT1	250
E06	4.3	INF-TOR-INT0.5	250
E07	4.3	INF-TOR-INT200	600
E08	4.3	INF-TOR-INT300	600
E09	4.3	INF-TOR-INT500	600
E10	4.3	INF-TOR-INT338	600
SDN-based			
E11	4.2	SDN-VPN-AWND800	100
E12	4.3	SDN-VPN-BDP800	250
E13	4.3	SDN-VPN-BDP400	100
E14	4.3	SDN-VPN-BDP200	100
E15	4.3	SDN-VPN-BDP100	100
E16	4.3	SDN-TOR-AWND800	100
E17	4.3	SDN-TOR-AWND1600	100
E18	4.3	SDN-TOR-BDP800	100
E19	4.3	SDN-TOR-BDP1600	150
RAINBOW			
E20	4.2	RNB-SSH_TRACES-A10	200
E21	4.2	RNB-SSH_TRACES-A5	200
E22	4.2	RNB-SSH_TRACES-A2	200
E23	4.2	RNB-SSH_TRACES-A1	200
E24	4.2	RNB-VOIP-A1	200
E25	4.2	RNB-VOIP-A2	200
E26	4.2	RNB-VOIP-A5	200
E27	4.2	RNB-TOR-A10	100
E28	4.2	RNB-SSH_TRACES-PASSV	200
E29	4.2	RNB-VOIP-PASSV	200
Conflux / Multi-flow			
E30	4.4	INF-CONFLUX-INT3	250
E31	4.4	INF-CONFLUX-INT2	250
E32	4.4	INF-CONFLUX-INT1	250
E33	4.4	INF-CONFLUX-INT0.5	250
E34	4.4	SDN-CONFLUX	250
Defenses			
E35	4.5	RNB-KFA	200
E36	4.5	RNB-DEMARK	200
E37	4.5	RNB-RANDOM	200
E38	4.5	RNB-KFA-RANDOM	200
E39	4.5	RNB-DEMARK-RANDOM	200
Mixed False Positives			
E40	C.4	SDN-TOR-DETECT	250
E41	C.4	INF-TOR-DETECT	250
E42	C.4	INF-TOR-PROXY-DETECT	~3100

D.3 Background Traffic Generation

In our mixed-background experiments, traffic is generated by three concurrent actor classes: (i) interactive multitab web browsing, (ii) bulk HTTP(S) downloads, and (iii) paced HTTP(S) uploads. Each scenario is seeded so that the overall schedule is reproducible, while individual jobs still sample URLs, timings, tab counts, click counts, and transfer sizes pseudo-randomly. The runner allows one job per actor type, with up to three jobs active globally. A typical workload therefore consists of one browser session overlapping with one download and one upload stream.

The browsing actor generates long-lived browser-like sessions from a filtered Tranco-based corpus of sites that successfully open over Tor. For each job, it samples 2–5 sites, opens them in separate tabs, and performs 1–5 shuffled passes across the open tabs, inserting 3–10 randomized link-click attempts. This produces heterogeneous interactive HTTPS traffic with tab creation, tab switching, page loads, and secondary link traversals rather than a single linear fetch sequence.

The download actor generates bulk downstream traffic by selecting a URL from a small corpus of large test files and issuing a full HTTP GET. The client stops after a randomly sampled volume between 1 MB and 50 MB. The upload actor generates paced upstream traffic using HTTP POST requests to an application-neutral echo endpoint, again with payload sizes sampled uniformly between 1 MB and 50 MB. Together, these actors approximate bursty background file-transfer behavior while keeping per-job volume bounded.

The client-side and server-side scenarios use the same logical traffic mix but differ in routing. In the client-side scenario, all traffic is sent through Tor directly: browser sessions use the Tor-browser backend, while non-browser HTTP actors use Tor's SOCKS endpoint. In the server-side scenarios, the same workload first traverses Tor and then exits through an onion-reachable or public SOCKS proxy. Browser sessions use a separate Firefox-compatible backend through the local chaining proxy, while the HTTP actors use the same chained SOCKS path. Thus, the traffic generation logic remains constant, but the detector vantage changes.

E Waterfall API Reference

Tables 12–16 document the full Waterfall API grouped by category.

Table 12: Action-based API functions.

Function Signature	Return	Description
`add_to_action(flow_name: string, chain: string)`	`void`	Registers a given `flow_name` with the Action service, allowing traffic modifications to be applied.
`set_action_drop(flow_name: string, options?: { duration?: number, amount?: number })`	`void`	Drops all packets on the specified flow. Optional parameters restrict duration or packet count.
`set_action_modify(flow_name: string, header: string, value: any, options?: { duration?: number, amount?: number })`	`void`	Modifies the given header in packets for the specified flow, setting it to the provided `value`. Optional parameters restrict duration or packet count.
`set_action_delay(flow_name: string, delay: number, options?: { duration?: number, amount?: number })`	`void`	Delays all packets on the specified flow by the given `delay`. Optional parameters restrict duration or packet count.
`set_action_rate_limit_delay( flow_name: string, rate: number, unit: "bytes" | "packets", options?: { duration?: number, amount?: number })`	`void`	Applies a token-bucket limiter to `flow_name`, delaying and forwarding traffic at `rate` per second (in selected `unit`). Optional `duration`/`amount` constrain how long or how many packets the limiter controls.
`set_action_accept(flow_name: string)`	`void`	Reverts to accepting all packets on the specified flow.
`on_packet_action(event_name: string, flow_name: string, callback: function({ packet: mod_packet }))`	`void`	Runs `callback` on each packet in the specified flows. The packet can be queued, individually processed, modified or queued.
`send_packet(packet: mod_packet)`	`void`	Sends out a packet received with `send_packet`.

Table 13: Timing-based API functions.

Function Signature	Return	Description
`get_watermark_time()`	`number`	Returns the elapsed time since the watermark start.
`get_watermark_start_time()`	`number`	Returns the start time of the watermark, relative to the UNIX epoch.
`set_watermark_start(time: number)`	`void`	Sets the watermark start time, relative to capsule creation.
`set_watermark_start_absolute(time: number)`	`void`	Sets the watermark start time, relative to the UNIX epoch.
`on_time(event_name: string, time: number, callback: function)`	`function()`	Runs `callback` after `time`, relative to the watermark start. Returns a cancel function.
`on_timeout(event_name: string, delay: number, callback: function)`	`function()`	Runs `callback` after `delay` nanoseconds. Returns a cancel function.
`on_interval(event_name: string, duration: number, callback: function)`	`function()`	Repeatedly runs `callback` every `duration`. Returns a cancel function.
`sync_start_time(event_name: string, timeout: number, callback: function())`	`void`	Synchronizes start time of embedding and detection capsule. If no other capsule is found before `timeout`, this capsule is stopped. Once synchronized, the `callback` is run.

Table 14: Monitor-based API functions.

Function Signature	Return	Description
`detect_silent_interval( event_name: string, duration: number, packet_max: number, flows: string | string[], callback: func(start_time: number))`	`{set_duration: func(duration: number)}`	Runs `callback` when packet count falls below `packet_max` within `duration`. Returns an object supporting dynamic adjustments.
`detect_burst_interval( event_name: string, duration: number, packet_min: number, flows: string | string[], callback: func(start_time: number))`	`{set_duration: func(duration: number)}`	Runs `callback` when packet count exceeds `packet_min` within `duration`. Returns an object for dynamic adjustments.
`monitor_packet_count( event_name: string, duration: number, flows: string | string[], callback: function(packet_count: number))`	`void`	Observes how many packets traverse the specified flows within a `duration`. The `callback` receives the count per `duration`.
`monitor_data_throughput( event_name: string, duration: number, flows: string | string[], callback: function(data_amount: number))`	`void`	Observes data transmitted over specified flows. The `callback` receives the total data amount per `duration`.
`monitor_group( event_name: string, interval: number, flows: string | string[], metrics: string[], callback: function(per_flow: table<string, {bytes?: number, count?: number, . . . }>), start_time?: number)`	`void`	Periodically samples the requested `metrics` on `flows` every `interval`, delivering per-flow measurements to `callback`.
`on_packet(event_name: string, flows: string | string[], callback: function(packet: packet))`	`void`	Runs `callback` for each observed packet on given flows, passing packet data for analysis.

Table 15: Communication-based API functions.

Function Signature	Return	Description
`broadcast_data(name: string, data: table<string, string[]>)`	`void`	Broadcasts a data table to other Waterfall nodes.
`get_data(name: string, callback: function({ sender_id: number, data: table<string, string[]> }))`	`void`	Runs `callback` upon receiving data with the given `name` and its content.
`broadcast_message(name: string, message: string)`	`void`	Broadcasts a message to other Waterfall nodes.
`get_message(name: string, callback: function(string))`	`void`	Runs `callback` upon receiving a message with the given `name`.
`broadcast_key(scheme: string, key: number[] | string[], extra_data: table<string, []string>)`	`void`	Broadcasts the specified `key` and `extra_data` to other nodes for watermark detection.
`submit_key(scheme: string, key: number[] | string[])`	`void`	Submits an extracted key to the matching service for verification.
`submit_data(scheme: string, data: table<string, []string>)`	`void`	Submits the specified `data` to the matching service.
`send_data(scheme: string, recipient: string, field: string, values: string[] | number[], . . . )`	`void`	Sends structured data for `scheme` to `recipient`. Accepts any number of field/value pairs (each value as an array).

Table 16: Other API functions.

Function Signature	Return	Description
`get_capsule_id()`	`number`	Returns the numeric ID of the current capsule.
`get_flows()`	`{ flow_name: { src_ip: string, dst_ip: string, src_port: string, dst_port: string, protocol: string } }`	Lists all flows associated with the current capsule.
`bind_flow(event_name: string, name: string, flow: {src_ip: string | string[], dst_ip: string | string[], src_port: string | string[], dst_port: string | string[], protocol: string | string[]}, callback: function([]{ src_ip: string, dst_ip: string, src_port: string, dst_port: string, protocol: string }))`	`void`	Requests to bind a flow, then runs the `callback` with the captured flow.
`cancel_event(event_name: string)`	`void`	Stops listening for or triggering the event with `event_name`.
`get_key(scheme: string, key_list_name: string)`	`[]number`	Requests a key from the Key service for the specified `scheme`.
`get_parameter(name: string)`	`any`	Returns the parameter with given `name`.
`stop()`	`void`	Terminates the current capsule.