PhantomBox: Fully Homomorphic Web Application Firewall

Jiaao Ma
Duke University
Durham, NC, USA
jm790@duke.edu

Ning Liang
Duke University
Durham, NC, USA
ning.liang@duke.edu

Ceyu Xu
Duke University
Durham, NC, USA
ceyu.xu@duke.edu

Alexander van Lanschot
Duke University
Durham, NC, USA
alexander.van.lanschot@duke.edu

Yunming Xiao
Northwestern University
Evanston, IL, USA
yunming.xiao@u.northwestern.edu

Shihan Lin
Duke University
Durham, NC, USA
shihan.lin@duke.edu

Xiaowei Yang
Duke University
Durham, NC, USA
xwy@cs.duke.edu

Lisa Wu Wills
Duke University
Durham, NC, USA
lisa@cs.duke.edu

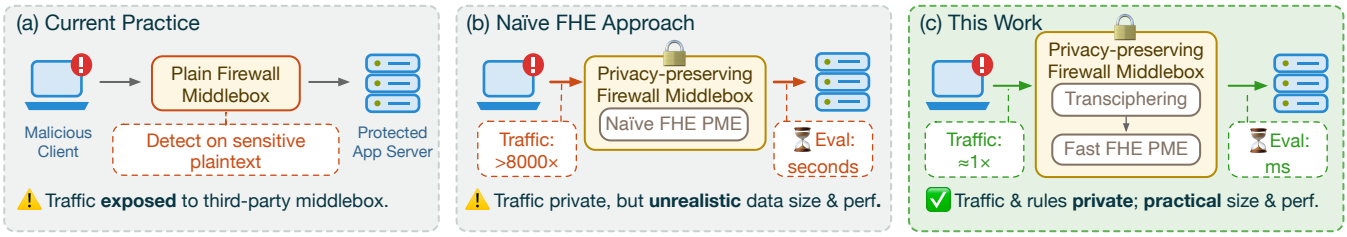


Figure 1. Three WAF deployment models. In current practice, a third-party firewall middlebox (a) performs intrusion detection on plaintext. Naive FHE implementation (b) preserves traffic privacy but suffers from prohibitive traffic size expansion. Naive FHE RegEx Pattern Matching Engine (PME) is prohibitively slow. Our system (c) achieves both privacy and practical performance by leveraging transciphering and fast PME.

Abstract

Web Application Firewalls (WAFs) defend against malicious traffic by inspecting application payloads against regular expression (RegEx) rulesets. Because ruleset maintenance requires specialized expertise, operators typically delegate WAF functionality to third-party middleboxes, which cannot inspect encrypted traffic without access to the secret keys. Prior privacy-preserving WAFs sacrifice either RegEx expressiveness, third-party offloading, or communication costs. Fully Homomorphic Encryption (FHE) can evaluate RegEx on encrypted traffic, but thousand-fold ciphertext expansion and seconds-per-character compute costs have kept FHE-based WAFs unrealistic.

We present PhantomBox, an FHE-based WAF that co-designs the datapath from client-side encryption through middlebox rule compilation and execution. To mitigate the compute bottleneck, PhantomBox introduces a novel ahead-of-time RegEx compiler with two FHE-specific optimizations: Fast Literal Matching and Boolean Fusion. The compiler also exploits equality saturation to further leverage the optimizations. To reduce network overhead, PhantomBox employs hardware-accelerated transciphering, keeping on-wire size comparable to plaintext while reducing client encryption latency by 84.6× over CPU. Evaluated on the Proofpoint ruleset with over 5,800 web-attack signatures, PhantomBox achieves a median evaluation latency of 7.7 ms on an AWS EC2 F2 instance. PhantomBox is the first system to make FHE-based WAF practical for real-world deployment, achieving full RegEx expressiveness, middlebox offloading, and cryptographic privacy simultaneously.

This work is licensed under a Creative Commons Attribution 4.0 International License.

ASPLOS '27, Heraklion, Greece

© 2027 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-3012-2/2027/04
https://doi.org/10.1145/3845814.3850054

CCS Concepts: • **Hardware → Hardware accelerators**; • **Networks → Middle boxes / network appliances**; • **Security and privacy → Firewalls**; **Privacy-preserving**

protocols; *Web application security*; • **Software and its engineering → Compilers**.

Keywords: Fully Homomorphic Encryption; Web Application Firewall; Regular Expression; Compiler; Equality Saturation; Transciphering; FPGA

ACM Reference Format:
Jiaao Ma, Ning Liang, Ceyu Xu, Alexander van Lanschot, Yunming Xiao, Shihan Lin, Xiaowei Yang, and Lisa Wu Wills. 2027. PhantomBox: Fully Homomorphic Web Application Firewall. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '27), April 11–15, 2027, Heraklion, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3845814.3850054

1 Introduction

Modern Internet applications heavily rely on Web Application Firewalls (WAFs) to defend against malicious traffic [5]. Unlike basic network firewalls, WAFs perform Deep Packet Inspection (DPI) on the application payload to detect exploits, cross-site scripting, and remote code execution [50, 55]. Because attack vectors are diverse and continuously evolving, WAF inspection relies on complex rule sets built on regular expressions (RegEx), where each pattern captures the structural characteristics of a specific threat.

Maintaining these complex rulesets requires dedicated security expertise, so operators typically delegate WAF functionality to third-party middlebox providers such as Akamai [1] and Cloudflare [2], which inspect incoming requests against professionally curated rulesets before forwarding benign traffic to the protected application server.

However, this ubiquitous delegation model (Figure 1a) breaks end-to-end encryption. To inspect payloads, the middlebox must see them in plaintext. Therefore, with HTTPS now securing over 95% of Internet traffic [4], the middlebox must terminate Transport Layer Security (TLS) on the application's behalf, which requires applications to hand over their TLS private keys to the middlebox [15, 67]. As a result, a third party observes the plaintext of every request, including the passwords, session cookies, and personal data of overwhelmingly benign users. This drives a critical need for privacy-preserving WAFs that protect the application server from malicious requests while keeping all user payloads confidential from the middlebox.

Existing attempts to build privacy-preserving WAFs cannot satisfy three properties simultaneously: *generic RegEx expressiveness*, *third-party offloading* (inspection on the middlebox, not the application server endpoint), and *practical communication costs*. Searchable-encryption schemes [14, 39, 48, 58, 71] sacrifice **expressiveness**, supporting only keyword matching and requiring decryption fallbacks for full RegEx evaluation. Server-side inspection designs [24, 41] sacrifice **offloading**, moving rule evaluation to application servers. Multi-Party Computation (MPC) approaches [36, 62–64] sacrifice **communication efficiency**, incurring multiple interactive rounds per inspection. Trusted Execution Environment (TEE) designs [31] bypass these trade-offs but rely on hardware vendor trust instead of cryptographic guarantees, leaving them vulnerable to side-channel attacks [11–13, 17, 59, 68]. Achieving all three properties under purely cryptographic protection has remained an open problem.

In principle, Fully Homomorphic Encryption (FHE), a family of encryption schemes that allow computing directly on encrypted data, is an ideal fit: the middlebox could apply its rules to encrypted requests without seeing the plaintext. In practice, however, prior work has considered FHE-based WAFs prohibitive (Figure 1b) [24, 32]. Even with FHE over the Torus (TFHE) [20], the most efficient scheme for byte-level computation, two roadblocks remain.

First, RegEx evaluation under TFHE is prohibitively slow. State-of-the-art plaintext engines compile RegEx into Deterministic Finite Automata (DFAs) [61], processing one state per input character via a simple table lookup. Under FHE, however, the middlebox cannot see the character, so it must perform every lookup on ciphertext as a Programmable Bootstrapping (PBS) operation. PBS is an encrypted table lookup whose cost grows exponentially with the number of input bits. Since the table is indexed by both the current state and the input character, even a minimal 4-state DFA requires a 10-bit PBS, costing ~2.4 s per character (Section 3.2.2).

Second, TFHE ciphertexts are enormous, making communication prohibitively expensive. For example, encrypting a 100 KB request inflates it to 814 MB, taking 6.5 s to upload even over a 1 Gbps link before inspection begins.

In this work, we present *PhantomBox* (Figure 1c), the first FHE-based WAF practical for real-world deployment, inspecting encrypted requests in milliseconds rather than minutes. Our key insight is that FHE cannot serve as a drop-in replacement within existing designs. *Practicality* requires rebuilding them around TFHE's own cost model: DFA-based rule execution fits naturally on CPU but is prohibitive under TFHE. Likewise, replacing the cipher inside TLS with TFHE introduces thousands-fold data size inflation. PhantomBox therefore co-designs the entire datapath around TFHE to tackle both prohibitive computation and communication costs.

To tackle *computation*, we introduce the PhantomBox Compiler, an Ahead-of-Time (AOT) RegEx compiler that optimizes rules once, before any encrypted traffic arrives at the middlebox. PhantomBox Compiler abandons DFAs in favor of an Abstract Syntax Tree (AST) that decomposes each RegEx into small sub-operations, avoiding costly wide lookup tables. Profiling the industrial Proofpoint Emerging Threats ruleset [52] reveals that just two operations, Boolean combination (40.2%) and literal matching (35.0%), account for over 75% of all AST nodes. We target each one with a TFHE-specific optimization. *Boolean Fusion* evaluates an entire N-input AND/OR chain with a single PBS, instead of the

traditional implementation of a pairwise reduction tree ($N-1$ PBS). *Fast Literal Matching* replaces expensive per-character comparison with a randomized equality check whose PBS count stays nearly constant (*e.g.*, 3 instead of 17 PBS for a 16-character literal).

Besides reducing per-operation runtime, these optimizations also change the conventional heuristic for global rule rewriting. While conventional compilers factor out shared fragments to avoid recomputation, our cost model often favors the opposite: duplicating a shared fragment into each branch forms longer literals and fewer nodes that are actually cheaper to evaluate. However, the optimization path to such a globally optimized rewrite often passes through intermediate rewrites that temporarily increase the cost, which a greedy optimizer would reject and stop earlier. We therefore use *equality saturation* to explore the space of equivalent rewrites and extract the globally cheapest one. At runtime, the optimized program is further specialized against each request's plaintext metadata (*e.g.*, payload length), pruning dead branches and infeasible rules.

To tackle *communication*, PhantomBox adopts *transciphering*, which keeps on-wire size comparable to plaintext and avoids ciphertext expansion. The client encrypts traffic with a compact, non-TFHE symmetric cipher. The middlebox then homomorphically converts these ciphertexts into TFHE ciphertexts for inspection. We adopt a Learning-With-Rounding (LWR) [22] cipher, where the conversion overhead is negligible. However, this shifts the workload to the client: unlike AES with dedicated instructions on modern CPUs, LWR has no special hardware support. Commodity laptops can absorb this cost in software, but power-constrained IoT devices show high encryption latency. For these devices, we design a client-side FPGA accelerator built on a key observation: LWR's core Kreyvium PRNG, which naively emits one bit per cycle, is limited by FPGA routing rather than logic. By redesigning its shallow logic to emit 12 bits per cycle, a single Kreyvium core outperforms the conventional choice, SHAKE-256, at only a fraction of its area. The resulting accelerator sustains 2.16 M ciphertexts per second on a sub-5 W FPGA and reduces encryption latency by 84.6× over an optimized CPU baseline.

In summary, we make the following contributions:

- We architect and implement PhantomBox, the first FHE-based WAF that offloads full RegEx inspection to third-party middleboxes, without exposing user traffic or proprietary rulesets.
- We build an ahead-of-time FHE RegEx compiler with two FHE-specific optimizations, *Boolean Fusion* and *Fast Literal Matching*, and use equality saturation to exploit the cost model they create, achieving a median 6.5× speedup over an unoptimized AST baseline.
- We present a hardware-accelerated transciphering architecture that keeps on-wire size at ∼1× plaintext, sustains 2.16 M ciphertexts per second under 5 W, and reduces encryption latency by 84.6× over an embedded CPU.
- We evaluate PhantomBox on an AWS EC2 F2 instance with the Proofpoint Emerging Threats ruleset [52] (over 5,800 signatures), achieving a median latency of 7.7 ms. We further identify legacy rule design as the main source of remaining tail latency, and propose privacy-aware rule engineering and API design that eliminate it.

2 Background and Related Work

2.1 Web Application Firewall and Middlebox

A Web Application Firewall (WAF) inspects application traffic to distinguish benign requests from malicious ones before they reach the protected services. In modern deployments, WAF inspection spans multiple layers of the request, including network and transport metadata, application-level structure such as URI paths and headers, and, most importantly, content-bearing fields such as request bodies and cookies. Among these layers, content inspection is the most expressive part of the pipeline: it captures the semantics of common web attacks[1].

In practice, WAF content inspection is driven by pattern-matching rules that range from simple literals to full regular expressions (RegEx). Literal patterns capture fixed indicators such as keywords or command fragments, while full regular expressions capture structural variation across families of syntactically related payloads. As a result, RegEx-oriented pattern matching forms a central component of modern WAF engines [50, 55].

Maintaining an effective WAF ruleset requires continuous tracking of emerging vulnerabilities, a specialized effort that application operators prefer to offload. In practice, applications delegate WAF functionality to third-party middlebox providers such as Akamai [1] and Cloudflare [2], who maintain professionally curated rulesets as a managed service [26]. Architecturally, the provider's middlebox sits on the request path between clients and the application server, inspecting every request before forwarding it. This delegation model, however, requires the middlebox to observe request content in order to apply RegEx-based rules, raising the question of how to preserve traffic confidentiality under middlebox delegation.

2.2 Related Work of Privacy-Preserving WAF

With over 95% of Internet traffic now carried over HTTPS [4], delegated middleboxes can no longer inspect payloads without obtaining the endpoints' TLS keys [15, 67], which negates the confidentiality HTTPS was designed to provide. This

[1]Examples include SQL injection [29], XSS [65], and command injection [33].

conflict has motivated a substantial body of work on privacy-preserving middlebox inspection, exploring different trade-offs among RegEx expressiveness, delegation model, communication overhead, and trust assumptions.

Searchable Encryption and Garbled Circuits. One line of work, pioneered by BlindBox [58] and its follow-ups [14, 39, 48, 71], evaluates searchable encryption and garbled circuits for *keyword and token matching* over encrypted traffic at the middlebox. While these designs prove that an encrypted middlebox is feasible, they are restricted to exact keyword patterns or coarse heuristics for limited RegEx rules. Furthermore, they incur high payload expansion ($\sim 5\times$ [39] to $\sim 14\times$ [58]) and heavy per-connection setup times (e.g., 97 seconds in BlindBox [58] and up to 3,000 hours in Embark [39]). Recent schemes such as PrivRE [32] extend expressiveness by masking sensitive fields, but remain restricted to simple patterns and leak partial rule information to endpoints.

Application Server-Side Inspection. Alternative architectures such as SPABox [24] and InviCloak [41] avoid middlebox payload exposure by moving RegEx evaluation to the application servers. In particular, SPABox delegates garbled circuits-encoded proprietary rules [43] to application servers for RegEx evaluation. However, performing inspection at application server endpoints defeats the primary operational motivation of outsourced WAF: the middlebox cannot proactively protect servers from malicious payloads.

Zero-Knowledge Proof (ZKP) Middleboxes. A recent approach, notably zkMB [27] and Zombie [75], proposes using zero-knowledge proofs (ZKP) to let clients prove that encrypted traffic complies with firewall policies without revealing payload content. However, ZKP middleboxes present three fundamental limitations for enterprise WAFs. First, they place the computationally intensive proof generation on the client: generating a proof for a 1 MB payload requires several minutes and gigabytes of RAM even on high-end server processors, while inflating traffic by orders of magnitude (e.g., up to 240× [75]). Second, to generate the proof, the client must evaluate the policy, which inherently requires *exposing the proprietary ruleset* to untrusted clients (unless heavy cryptographic obfuscation is employed). Third, existing ZKP middleboxes support only simple, fixed patterns (*e.g.*, basic keyword/digit checks and DNS lookups), falling far short of full PCRE expressiveness.

Multi-Party Computation (MPC) Protocols. Protocols based on Multi-Party Computation [36, 62–64] enable private DFA evaluation over encrypted inputs. However, because state transitions depend sequentially on prior states, evaluating an L-character string requires L interactive rounds between client and middlebox. Across typical latencies ($\sim$ 36 ms client-to-CDN round-trip time (RTT) [8, 41]), evaluating a 100-character URL may require several seconds of network transit.

Protocol-Level Slicing. Multi-Context TLS (mcTLS) [47] and its follow-up [10] modify TLS to support fine-grained read/write permissions, allowing the client and application server to jointly grant a middlebox access to selected portions of the data stream. However, in these schemes, a middlebox performing DPI must still be granted read access, exposing the inspected plaintext to it.

Trusted Execution Environments (TEEs). Finally, systems such as mbTLS [46] and Phoenix [31] propose leveraging TEEs to execute DPI over plaintexts inside hardware-isolated enclaves. While TEEs achieve near-native performance, they place *confidentiality trust* in the hardware vendor and processor microarchitecture: the enclave holds decryption keys and operates on plaintexts, leaving it vulnerable to architectural side-channel and transient-execution attacks [11–13, 17, 59, 68]. In contrast, PhantomBox employs hardware acceleration (*e.g.*, FPGAs) solely for improving throughput: because the middlebox never holds the decryption key, confidentiality is based on the hardness of LWE instead of hardware vendor trust.

In short, a practical WAF that simultaneously achieves versatile PCRE expressiveness, middlebox offloading, lightweight client overhead, and cryptographic privacy remains an open challenge.

2.3 Why Fully Homomorphic Encryption?

Fully homomorphic encryption (FHE) is uniquely positioned to close this gap: it supports arbitrary computation directly on ciphertexts, requires no interaction during evaluation, and derives its security from mathematical hardness assumptions alone. An FHE-powered middlebox can therefore evaluate RegEx matching on encrypted request content and return an encrypted detection result, without learning the plaintext or leaking rulesets.

2.3.1 TFHE as the Target Scheme.

An FHE scheme specifies an encryption function, supported data types, and operations. Different schemes make different trade-offs, and the choice of scheme largely determines whether a given application is feasible [6, 23].

Among contemporary FHE schemes, CKKS [18, 19] and TFHE [20] represent the two most widely adopted families. While CKKS targets batch linear-algebra workloads [30, 56], TFHE encrypts individual multi-bit integers and exposes a *Programmable Bootstrapping* (PBS) primitive that evaluates an arbitrary lookup table (LUT) on a single ciphertext, naturally supporting nonlinear and bitwise logic at per-value granularity [35, 38, 42].

WAF content inspection operates on strings, demanding exact equality and comparison logic. These characteristics align directly with TFHE's strengths: per-byte granularity, exact integer arithmetic, and arbitrary LUT evaluation via PBS. We therefore build PhantomBox on the TFHE scheme.

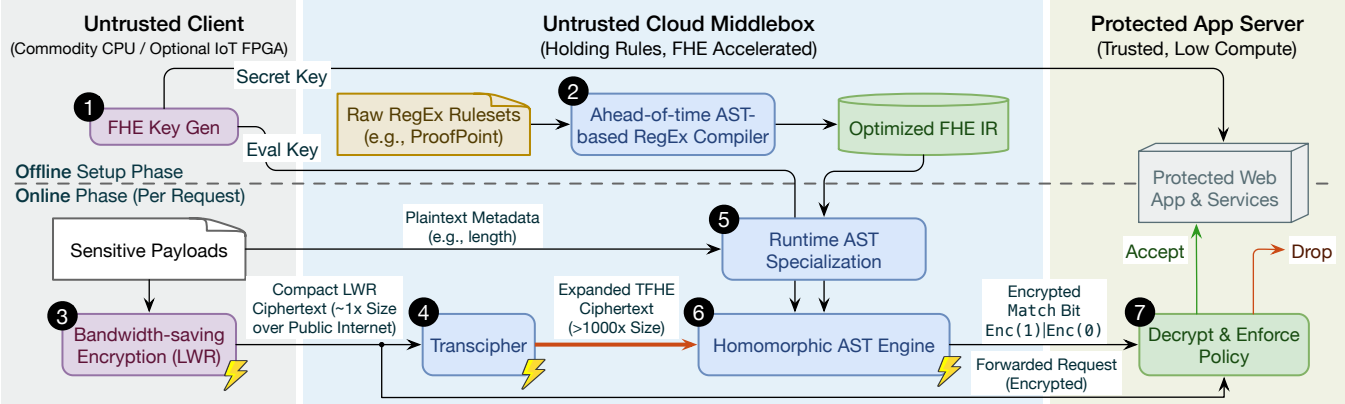


Figure 2. PhantomBox end-to-end architecture and traffic flow. Circled numbers ①–⑦ denote offline setup and per-request evaluation stages (Section 3.3). Lightning icons mark components supporting hardware acceleration.

2.3.2 TFHE Computation Model.

TFHE follows a client–server key model. The client generates a *secret key* for encryption and decryption, together with a public *evaluation key* that enables homomorphic computation but not decryption. The server, in our setting, the middlebox, operates solely with the evaluation key: it can transform ciphertexts according to prescribed programs but never recover the underlying plaintexts.

During encryption, TFHE encodes 2–10 bits of plaintext message, depending on the concrete parameter set [72], as a Learning With Errors (LWE) [54] sample[2]: a vector of hundreds to thousands of unsigned integers, totaling kilobytes per encrypted value. LWE security rests on the deliberate injection of a small random *error* (noise) during encryption. This noise, however, gets accumulated with subsequent operations. Once the accumulated noise exceeds a scheme-dependent threshold, the encoded message becomes irrecoverable even with the secret key, requiring a corrective step to suppress noise for subsequent homomorphic evaluation.

TFHE addresses the noise problem with *Programmable Bootstrapping* (PBS). PBS applies a *server-specified* lookup table (LUT), defined as part of the FHE program, to a ciphertext, and resets its noise to a fresh level to enable further computation. This dual role makes PBS the most powerful yet expensive TFHE primitive [35, 38]. By contrast, *LWE-native* operations (modular addition, subtraction, and scalar multiplication) work directly on LWE ciphertext vectors without noise suppression and are much cheaper. [3]. Because PBS dominates execution time, the total cost of a TFHE program is effectively determined by the *PBS count*. Minimizing this metric is the central optimization target for TFHE program design, and the primary focus of our RegEx compiler (Section 4).

[2]Throughout this paper we target the 128-bit security level. [7]

[3]A PBS is four orders of magnitude slower than a LWE-native operation, measured on AMD EPYC 7R13, single-threaded.

3 Privacy-preserving WAF Architecture

3.1 Deployment Model and Trust Boundaries

Modern WAF architectures follow a delegation model where web application operators contract third-party cloud/CDN providers (such as Cloudflare or Akamai) to inspect incoming traffic against professionally curated, proprietary attack signatures [1, 2]. PhantomBox preserves this delegation model while establishing cryptographic confidentiality for both user traffic and provider rulesets. It spans three primary entities:

Clients (adversary or victim). Clients initiate requests and range from standard desktop and mobile browsers to embedded IoT devices. In particular, malicious clients may craft payloads to evade detection and observe accept/drop outcomes, but cannot obtain the WAF ruleset. Clients encrypt sensitive payload fields locally under their own FHE key, ensuring no plaintext leaves the client. The encryption does not need specialized hardware modifications: lightweight symmetric LWR encryption runs entirely in software (e.g., via WebAssembly in browsers or a TLS client extension). On an Apple M4 MacBook Air, client-side encryption sustains 39 Mbps within a 10 W power budget. For power-constrained IoT devices (e.g., Cortex-A53), we propose an optional FPGA accelerator to minimize latency and energy (Section 7.1).

Middlebox (honest-but-curious). Following the standard honest-but-curious model of prior work [14, 32, 39, 40, 48, 58, 71], the middlebox executes the inspection protocol faithfully to ensure service quality, but may attempt to infer payloads from ciphertexts. However, the cryptographic hardness of FHE ensures that the middlebox learns nothing about request payloads or evaluation outcomes. Because homomorphic evaluation requires arithmetically intensive high-degree polynomial computations, the middlebox is the only entity in PhantomBox that requires dedicated hardware acceleration (*e.g.*, FPGA/ASIC clusters).

Protected Application Server (trusted). The fully trusted application server hosts the backend service. It deploys a lightweight software gateway shim (e.g., an Nginx reverse-proxy filter) in front of internal application services. The gateway holds the client's secret key, decrypts the single-bit match result, and enforces an accept/drop policy. Because single-bit LWE decryption requires only a basic vector inner product, the application server requires no hardware acceleration.

3.2 Two Fundamental Obstacles for FHE WAFs

3.2.1 Network Bottleneck: Ciphertext Expansion.

Each TFHE ciphertext encrypting a single byte takes over 8 KB, yielding an expansion factor of 8,144× (Table 1). A 100 KB HTTP request body would expand to 814 MB (6.5 s on a 1 Gbps link), making direct TFHE encryption over public Internet infeasible.

3.2.2 Compute Bottleneck: DFA Evaluation in TFHE.

Standard RegEx engines such as Hyperscan [61] compile patterns into DFAs that index a transition table by current state and input character. Under FHE, encrypted values cannot serve as branching conditions, so the DFA must be realized by a PBS lookup table per state transition whose compute cost scales *exponentially* with input bit-width. A transition consumes $\lceil \log_2 N \rceil + 8$ input bits (N states, one ASCII byte)[4]: even a 4-state DFA requires a 10-bit LUT at ~418 ms for a single state transition on a single character, while real-world patterns routinely produce hundreds of states (Figure 3). DFAs fundamentally rely on these wide LUTs and cannot leverage fast, LWE-native linear operations. A completely new execution model is necessary.

3.3 End-to-End Workflow and Traffic Flow

To overcome these roadblocks, we propose an architecture that coordinates offline preparation and per-request online evaluation through the following staged pipeline (Figure 2):

Offline setup. ① Key Generation and Distribution: The client generates an FHE key pair, sending the evaluation key to the middlebox (reused across sessions) and the secret key to the trusted application server.

② AOT Rule Compilation: The middlebox compiles its PCRE ruleset into an optimized High-level Intermediate Representation (HIR) using cost-driven equality saturation (Section 4) to avoid DFA state count explosion. This compilation runs ahead-of-time only once per ruleset release.

Online evaluation (per request). ③ Client Encryption: The client partitions a request into unencrypted metadata (*e.g.*, IP address, URL path, insensitive headers) and sensitive payload fields (*e.g.*, POST body, query strings). It encrypts the payload with compact Learning-With-Rounding (LWR) [22] stream encryption and keeps on-wire data size near 1× plaintext (Section 6).

④ Transciphering: For each request, the middlebox homomorphically transciphers the compact LWR ciphertexts into TFHE ciphertexts at ~ 0.5 PBS per byte, enabling further homomorphic evaluation.

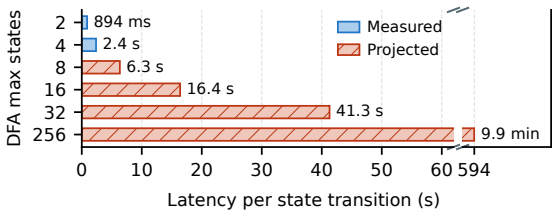
⑤ Specialization: The middlebox then specializes the precompiled HIR against request metadata (*e.g.*, payload length) to prune infeasible AST branches (Section 5).

⑥ Homomorphic AST Evaluation: The middlebox evaluates the specialized AST over the encrypted payload, emitting a single encrypted match bit Enc(b) (malicious/benign).

⑦ Policy Enforcement: The middlebox forwards the request payload alongside Enc(b) to the application server. The gateway decrypts Enc(b). If b = 1, it drops the request; if b = 0, the gateway decrypts the LWR payload and forwards the plaintext request to internal web application services.

All heavy homomorphic computation runs on the middlebox, whereas clients perform only lightweight symmetric encryption and application servers perform single-bit decryption. This asymmetry ensures that PhantomBox remains highly practical for real-world web applications.

Conventional CDNs provide Denial-of-Service (DoS) attack mitigation by absorbing volumetric attacks (*e.g.*, SYN floods and high-frequency HTTP request flooding) at the CDN edges. In production, mitigation operates exclusively on insensitive network- and transport-layer metadata (*e.g.*, source IP addresses, SYN cookies, request frequencies, and connection state) [2]. Because PhantomBox leaves request



(a) Per-transition PBS latency vs. DFA state count.

Example RegEx pattern	DFA states
`^GET$`	4
`^/bin/sh$`	8
`^Host:\slocalhost$`	16
`^GET /etc/passwd HTTP/\d.0$`	32
`^SELECT .{1,19} FROM$`	245

(b) Example Suricata-flavored patterns at each state-count tier.

Figure 3. Latency blows up as DFA states grow. (a) Single-thread PBS latency per transition; bars ≤ 32 states are measured (blue) via `tfhe-rs` with Concrete Optimizer [72] parameters, bars ≥ 64 states are projected (orange). (b) Example DFAs for Suricata-style RegExes.

[4]We assume ASCII encoding, though UTF-32 is also supported. Particularly, variable-width encodings such as UTF-8 are not supported due to FHE's data-obliviousness requirement.

metadata in plaintext, the cloud middlebox discards volumetric traffic before invoking FHE evaluation so it never reaches the protected application servers.

4 Ahead-of-Time Compiler

Plaintext WAF engines [61] compile Perl-Compatible RegEx (PCRE) patterns into DFAs, but under FHE a single DFA transition is prohibitively expensive (Section 3.2.2). TFHE-rs Pattern Matching Engine (PME) [73] avoids this by parsing each pattern into an AST of RegEx operations (`Lit`, `Concat`, `Alt`) and so on, and evaluating the tree directly. We adopt the same tree-structured evaluation model, but lower each AST into our own *High-level Intermediate Representation* (HIR) that our optimizations rewrite over. The engine then walks the HIR bottom-up, producing one encrypted boolean per node and combining them into a single match result.

We optimize the two most frequent operation types, boolean combination and literal matching, by exploiting LWE-native arithmetic in multi-bit TFHE (Section 4.1). These optimizations expose bidirectional rewrite opportunities that we harness with cost-driven equality saturation (Section 4.2).

4.1 Operation Optimization

An analysis of 7,634 patterns from the Proofpoint Emerging Threats ruleset [52] shows that two operations dominate the HIR: *literal matching* (35.0% of nodes) and *Boolean combination* (40.2%). Together they account for over 75% of all HIR nodes. We find that both can be optimized to reduce PBS counts by leveraging nearly free LWE-native operations in TFHE.

4.1.1 Boolean Fusion. A Boolean combinator (AND/OR) can be evaluated by using one PBS. N consecutive Boolean combinators with the same type can be evaluated by N PBS operations naively [73]. We propose an optimization for such Boolean combination chains by fusing multiple inputs with nearly free LWE additions, and requiring only one PBS for multiple combinators. We call this optimization **Boolean Fusion**.

TFHE-rs PME evaluates boolean trees by issuing one PBS per pairwise combinator; an N-input AND or OR chain therefore costs $N-1$ PBS. We identify that these can be fused under multi-bit TFHE with message space $\mathbb{Z}_{2^b}$. Given N encrypted booleans $b_1, \ldots, b_N \in \{0,1\}$, the sum $S = \sum_{i=1}^{N} b_i$ is computed via PBS-free LWE vector addition, and a single PBS evaluates an arbitrary lookup table with S as input. Setting the LUT to $f(S) = [S \neq 0]$ realizes N-input OR; setting $f(S) = [S = N]$ realizes N-input AND, the two combinators RegEx syntax needs.

The sum S is exact as long as $N \leq 2^b - 1$. With a common $b = 4$, this accommodates up to 15 inputs (and 255 for $b = 8$) in one PBS. For $N > 2^b - 1$, we build a balanced N-input reduction tree: each level sums groups of partial results, applies one PBS per group, and feeds the resulting booleans upward. In practice, the input counts (fan-in) of Boolean chains are modest: across the Proofpoint Emerging Threats ruleset, 95% of alternation nodes have fan-in ≤ 6 and 95% of concatenation nodes have fan-in ≤ 9, so the vast majority collapse to a single PBS (Figure 4a). Even when the reduction tree is needed, LWE additions are effectively free: wall-clock measurements show PBS dominates by >4,000× (Figure 4b).

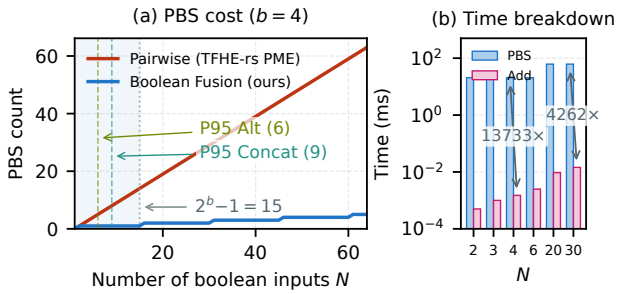


Figure 4. (a) PBS count vs. number of boolean inputs N for an N-input OR. (b) Log-scale time breakdown (single-threaded): PBS dominates LWE additions at every tested N.

4.1.2 Fast Literal Matching via Fixed-Weight Sampling. We propose an optimization that checks if two literals match with a negligible false positive rate by requiring only *a few* PBS operations. Matching an n-character literal against encrypted input requires $n + 1$ PBS in the naive approach. An appealing shortcut is to sum the difference vector of the known literal against inputs via nearly free LWE subtractions and test zero with one PBS, which reduces the cost to only 1 PBS. However, non-zero differences can cancel modulo q, producing false positives.

We introduce a Freivalds-style equality check [25] lifted to LWE ciphertexts, whose cost depends not on n but on a target false-positive bound ε. Just as Freivalds verifies $AB = C$ by checking $ABr = Cr$ for a random vector r, we verify $\mathbf{x} = \mathbf{p}$ by checking random projections of the difference $\mathbf{x} - \mathbf{p}$ against zero. The check is **sound on matches** (no false negatives) and **one-sided on mismatches** (false positives bounded by ε). By controlling ε to be lower than the PBS failure rate, the false positive rate can be negligible in practice.

Algorithm. Let $\mathbf{x} = (x_1, \ldots, x_n)$ be the encrypted input literal and $\mathbf{p} = (p_1, \ldots, p_n)$ the known literal. Compute the difference vector $d_i = x_i - p_i \pmod q$ using free LWE subtractions. For each of t independent trials, sample a random binary vector $\mathbf{w} \in \{0,1\}^n$ with exactly $\lfloor n/2 \rfloor$ ones (a *fixed-weight* mask). The weighted sum $z = \sum_i w_i \cdot d_i \pmod q$ is again a free LWE operation: since $\mathbf{w}$ is a plaintext binary mask, each $w_i \cdot d_i$ is a trivial select (include or exclude the ciphertext). A single PBS then tests whether $z = 0$. Fixing the Hamming weight to $\lfloor n/2 \rfloor$ guarantees every trial probes

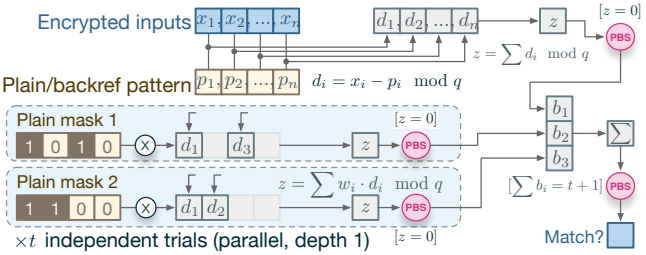


Figure 5. Pipeline of fast literal matching.

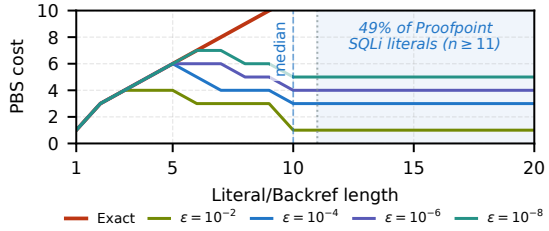


Figure 6. PBS cost vs. literal length n. Fast Literal Matching flattens cost to a constant beyond $n \approx 11$, where 49% of literals reside (shaded).

roughly half the positions, reducing false-positive probability to a combinatorial quantity that shrinks rapidly with n[5].

False-positive and cost analysis. If $\mathbf{x} = \mathbf{p}$, every $d_i = 0$ and $z = 0$ deterministically, yielding no false negatives. If $\mathbf{x} \neq \mathbf{p}$, the false-positive probability per trial is:

$$p_{\text{fp}} = \frac{1}{q} + \frac{q-1}{q \cdot \binom{n}{\lfloor n/2 \rfloor}} \tag{1}$$

where $q = 2^b$ is the ciphertext modulus. The $1/q$ term is the probability that a uniformly random element of $\mathbb{Z}_q$ equals zero; the second term accounts for the fraction of fixed-weight masks whose weighted sum over a nonzero difference vector collides with zero. After t independent trials the overall false-positive rate is $P_t = (p_{\text{fp}})^t$, so the minimum trials satisfying $P_t \leq \varepsilon$ is $t = \lceil \ln \varepsilon / \ln p_{\text{fp}} \rceil$. Since $\binom{n}{\lfloor n/2 \rfloor}$ grows exponentially with n, $p_{\text{fp}} \to 1/q$ rapidly: for $q = 256$ and $n \geq 11$, $t = 2$ already achieves $\varepsilon < 6.2 \times 10^{-5}$. The total cost is t PBS (which execute in parallel at depth 1) plus $\lceil \log_{255} t \rceil$ combining PBS via Boolean Fusion: 3 PBS total at depth 2 for any literal of 11 chars or longer. Compare this with the naive $n + 1$ PBS: for a 16-byte literal, fast literal matching reduces the cost from 17 to 3 PBS, a 5.7× improvement.

Across the Proofpoint ruleset, 49% of content literals have $n \geq 11$ (Figure 6), placing them in the flat-cost region where our scheme reduces cost from n+1 to as few as 3 PBS.

The optimization naturally extends to non-contiguous literals where the width of the gaps can be determined at compile time (*e.g.*, `foo.bar`) by using non-contiguous masks, avoiding multiple literal matching and extra concatenations.

PCRE Syntax Coverage. The compiler handles standard regular PCRE syntax, including quantifiers, alternations, character classes, and anchors. Note that our evaluation model naturally supports *look-around assertions* (both positive and negative lookahead and bounded lookbehind). Because subexpressions are evaluated without stateful cursor tracking, a look-around assertion lowers to a positional sub-tree evaluation at a fixed relative offset, directly reusing the Boolean Fusion combinators and Fast Literal Matching primitives described above.

In contrast, *back-references* are intentionally excluded. Matching a dynamically captured encrypted substring against another region requires pairwise cross-comparisons and causes prohibitive PBS overhead. However, back-references are practically negligible in real-world WAF rulesets, appearing in only 3 out of 3,425 Proofpoint ET rules (< 0.09%).

4.2 Global AST Rewriting

The regex AST produced by a standard parser offers limited opportunities for the optimizations described above. The benefit is capped by the number of alternations and literal matches that already exist in the AST: a single pass over the tree can apply Boolean Fusion wherever a Boolean combination chain exists, but beyond that, little else can be done.

We observe, however, that the AST can be rewritten bidirectionally and equivalently through *factoring* (extracting common sub-operations) and *distributing* (inlining shared sub-operations back into branches).

Our compiler deploys a catalog of 16 bidirectional rewrite rules spanning three categories: (1)*Factoring and distribution* (10 rules), which perform prefix, suffix, and mid-concatenation factoring over alternations and literals; (2)*Literal and character-class normalization* (3 rules), which merge adjacent literals and unroll or collapse character classes; and (3)*Structural and algebraic cleanup* (3 rules), covering double-negation elimination, repetition merging ($X\{n\}X\{m\} \leftrightarrow X\{n+m\}$), and dead-branch pruning. The three rules below represent the dominant factoring and normalization categories:

Prefix/suffix factoring ↔ Distribution: $ax \mid ay \leftrightarrow a(x \mid y)$

Class unrolling ↔ Construction: `[45]` ↔ `Alt("4","5")`

Mid-position distribution: `Cat`$(A,$ `Alt`$(X, Y), B) \leftrightarrow$ `Alt`(`Cat`(A, X, B), `Cat`(A, Y, B))

Traditionally, factoring is the only direction reducing the cost. For example, extracting a common digit-matching node that matches a digit from both branches to save PBS count for re-evaluating the node. However, rewriting in the distribution direction can also be beneficial because the two optimizations introduced in Section 4.1 yield the following unconventional cost implications under TFHE:

[5]Uniform sampling risks drawing an all-zero mask, which trivially yields $z = 0$ regardless of the input.

Longer literals are cheaper per byte. Under Fast Literal Matching, a 24-char literal and a 64-char one both cost 3 PBS (Figure 6). This inverts the traditional heuristic that code expansion is undesirable: *distributing* shared context into branches to form longer literals actively reduces cost.

Wide alternations are nearly free to combine. An N-way OR costs $\lceil N/(2^b-1) \rceil$ combining PBS via Boolean Fusion (Figure 4). Splitting a factored form $a(x|y)$ into $ax|ay$ increases the alternation width by one but creates two longer literals: if those literals cross the fixed-weight threshold, the net effect is a cost *reduction*.

Class unrolling enables literal absorption. A character class `[ab]` costs 1 PBS as a range check. Unrolling it to `Alt("a","b")` and then distributing the surrounding context creates two longer literals that may each cost fewer PBS than the original factored form.

Note that these rewrites do not always reduce cost monotonically: reaching a profitable rewrite may require first passing through an intermediate form with *higher* cost. As shown in Figure 7, a greedy optimizer may early terminate rewriting after State II for its even higher cost and cannot effectively exploit this bidirectional rewrite opportunity. This motivates the bidirectional rewrite rules that our compiler explores via equality saturation [60, 66, 69], which optimizes the rewriting strategy globally using an e-graph [60, 69]. Figure 7 traces such a chain: intermediate forms (II–III) cost *more* than the original, yet the e-graph preserves every form in shared equivalence classes rather than replacing one with another, so a cost-increasing rewrite never traps the search. We compare costs only at extraction, after saturation, and select the minimum-cost form (IV) regardless of the path used to reach it.

Termination criteria and convergence. To prevent combinatorial state explosion on complex patterns, we enforce a ceiling of at most 80 iterations or 100,000 e-graph nodes per rule. In practice, only tens of Proofpoint rules trigger the iteration or node cap. Further sensitivity analysis confirms that doubling both limits (160 iterations, 200,000 nodes) produces identical extracted costs, indicating that the search space converges rapidly.

Cost extraction and optimality guarantees. The cost function aligns with TFHE's execution costs: minimizing total PBS count as the primary objective, with PBS depth acting as a tie-breaker. If the e-graph saturates within the resource budget, the extracted AST is the global minimum-cost expression reachable under all rewrite rules. Alternatively, if rewriting stops at the iteration or node threshold, extraction returns the minimum-cost AST discovered among all equivalent forms, guaranteeing that the compiled output is never worse than the unoptimized input.

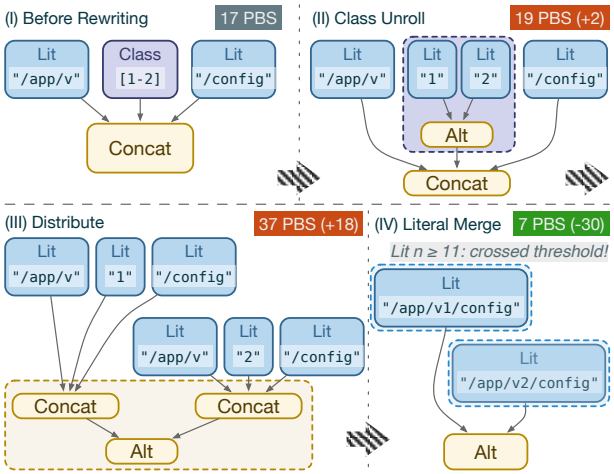


Figure 7. E-graph rewrite chain for `/app/v[12]/config`. (I) Original HIR before rewriting. (II) Class unrolling increases cost. (III) Distribution doubles it. (IV) Merging into a literal and collapsing the cost. Equality saturation finds this because it defers cost evaluation until the full rewrite space is visible.

5 Runtime Specialization

The AOT compiler of Section 4.2 optimizes each pattern in isolation. At runtime, a second opportunity arises: the encrypted payload length L is available as free metadata before any PBS executes. The *runtime specializer* partially evaluates the optimized HIR against L, pruning infeasible rules, dead branches, and impossible length splits to reduce PBS counts.

Inter-rule filtering. Every regex pattern can only match inputs of certain length ranges. For example, `a{2,4}b` matches exactly 3, 4, or 5 characters, never 2 or 6. The compiler precomputes this set of feasible lengths $\mathcal{L}$ for each rule by a bottom-up walk over the HIR. At runtime, if the packet length $L \notin \mathcal{L}$ (anchored) or $L < \min(\mathcal{L})$ (unanchored), the rule is discarded before any PBS executes.

Intra-rule budget propagation. For surviving rules, the specializer resolves how many characters each sub-expression can consume, given that the entire match must consume exactly L characters. It propagates a *budget* B (initially L) top-down through the HIR. At each node, it caps unbounded repetitions (e.g., `a{1,∞}` with B=20 becomes `a{1,20}`), prunes alternation branches that cannot fit within B, and unrolls resolved repetitions so that adjacent literals can be merged. A worklist iterates to a fixpoint: narrowing one child often tightens sibling budgets, triggering further pruning.

The specializer also distributes B across a concatenation. Consider `Cat`(C_1, C_2) where each child can match 2 or 5 bytes, and B=7. Interval arithmetic gives $[2,5] + [2,5] = [4,10] \ni 7$ and keeps both children at full width. It cannot see that lengths 3 and 4 are impossible for either child, since only

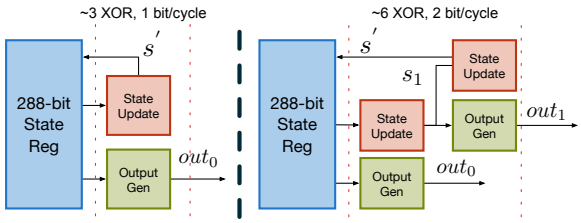


Figure 8. Comparison between naive Kreyvium pipeline (left) and 2-bit unrolled pipeline (right). Note the wires are much narrower compared to 288 (< 20) per output/state, and XOR has a very short critical path.

the splits $(2, 5)$ and $(5, 2)$ sum to 7. We solve this exactly using Boolean convolution. Each child's length set is represented as a polynomial $P_i(x) = \sum_{\ell \in \mathcal{L}(C_i)} x^\ell$ over Boolean. The feasible lengths for C_i are recovered via prefix and suffix products of siblings' polynomials. This yields the exact feasible set per child, eliminating ghost branches that interval arithmetic retains[6].

Caching. Speculative execution may re-evaluate the same sub-expression at the same cursor many times. We eliminate this redundancy with a content-addressed result cache keyed by (h, c), where h is a position-independent structural hash of the HIR sub-tree and c is the input cursor. A cache hit returns the stored end positions at *zero* PBS cost, whereas a miss evaluates normally and inserts the result.

6 Hardware-Accelerated Transciphering

6.1 The Two-Sided Transciphering Bottleneck

TFHE's ciphertext expansion prevents direct transmission over the public Internet. Even with *compressed* ciphertexts [73], a 100 KB payload expands to 0.8–1.6 MB (Table 1).

Transciphering decouples network cost from FHE expansion: the client encrypts with a compact symmetric cipher (1× overhead), and the middlebox homomorphically evaluates the decryption circuit to produce TFHE ciphertexts [22]. Cipher selection, however, introduces a two-sided bottleneck (Table 2). AES [44] and ChaCha20 [49] encrypt at >1,500 and >2,500 Mbps on the client thanks to dedicated CPU instructions, but their excessive PBS counts make middlebox-side homomorphic evaluation intractable [9, 28]. We adopt a Learning-With-Rounding (LWR)-based transciphering scheme [22], designed to align with TFHE's PBS structure while maintaining 128-bit security, requiring only ~0.5 PBS operations per ciphertext on the middlebox. Despite making the transmission cost feasible, an optimized Rust implementation[7] requires ~ 93 ms for encrypting only 10 Kb of data on an edge-class Cortex-A53.

Making transciphering viable at line rate requires solving this client-side bottleneck. Our accelerator reduces encryption latency by 84.6× over the software baseline, enabling real-time transciphering in an encrypted web architecture (Table 2).

Table 1. On-wire cost of transmitting a 100 KB payload under different encryption (1 Gbps link, 128-bit security). Native TFHE sizes are computed from concrete-optimizer parameters; seeded ciphertext sizes are amortized over a shared PRNG seed [73].

Encryption	100 KB payload	1 Gbps tx
Plaintext	100 KB (1×)	0.8 ms
Native TFHE, 4-bit	1.19 GB (11,888×)	9.5 s
Native TFHE, 8-bit	814 MB (8,144×)	6.5 s
Seeded TFHE, 4-bit	1.6 MB (~16×)	12.8 ms
Seeded TFHE, 8-bit	800 KB (~8×)	6.4 ms
Symmetric Cipher	**~100 KB (~1×)**	**0.8 ms**

Table 2. Comparison of transciphering schemes (latency measures the encryption time of a 10 Kb payload). Traditional ciphers are fast on the client but intractable to evaluate homomorphically. LWR shifts this trade-off.

Symmetric Cipher	Client Latency	Middlebox Latency	Viability
AES-128-GCM	$< 10\,\mu$s	~ 32 s	✗
ChaCha20-Poly1305	$< 10\,\mu$s	~ 93 s	✗
LWR (Baseline)	93.1 ms	~4 ms	✗
LWR (HW-accel)	**1.1 ms (84.6×)**	~4 ms	✓

6.2 Hardware-Accelerated LWR Encryption

Pseudo-random number generation (PRNG) is the dominant computational bottleneck in the LWR encryption pipeline, accounting for approximately 99% of CPU execution time. This overhead stems from the PRNG's reliance on bit-level operations such as shifts and XORs, which are inexpensive on FPGAs but poorly suited for CPUs. We implement a full end-to-end LWR encryption accelerator and focus our optimization effort on the PRNG datapath.

We evaluate the LWR encryption pipeline with two structurally distinct, hardware-friendly PRNGs, **Kreyvium** [16] and **SHAKE256** [45], which stress the FPGA hardware resource in opposite ways. SHAKE256 produces 1,088 bits of output every 24 rounds (48 clock cycles in our design); the width and depth of its datapath push each core above

[6] A similar constraint applies to capture-backref pairs: when a capture group and its backreference are siblings, their equal-length requirement further narrows the budget to lengths ℓ where $(B - r) \bmod m = 0$ (r is the combined sibling length, m the number of copies).

[7] Uses SHAKE256 as the underlying PRNG, compiled with `rustc` 1.94.0 using `opt-level=3` and fat LTO.

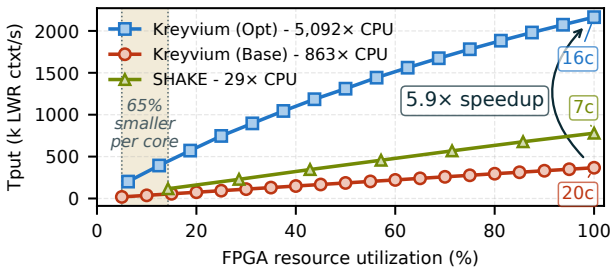


Figure 9. Client-side encryption throughput (kilo-ciphertexts/second) vs. FPGA resource utilization on a Zynq ZU3 (< 5 W). Endpoints indicate maximum core counts per configuration. The shaded region highlights the low-resource regime (< 15% utilization) where only compact Kreyvium cores fit. Optimized Kreyvium achieves up to 2.16×10^6 ctxt/s (16 cores), yielding a 5092× speedup over the Cortex-A53 CPU baseline and a 5.9× speedup over baseline Kreyvium.

90% LUT utilization, making it *LUT-bottlenecked*. A naive Kreyvium core, by contrast, emits only one output bit per cycle with a tiny per-core footprint; replicating it plateaus at ~30% LUT utilization because the design is bottlenecked by routingresourcese rather than compute-bound, leaving most of the device's logic idle.

We exploit this routing slack with a simple observation: Kreyvium's critical path for each output bit is only one to two XOR/AND gates deep. Combinationally unrolling the next-state logic lets a single core emit 12 bits per cycle within one clock period, absorbing the unused routing headroom without adding clock pressure. The unrolled core climbs to > 90% LUT utilization, which matches SHAKE256's density. The optimized Kreyvium delivers a 5.9× throughput speedup over the unoptimized baseline (Figure 8) under similar hardware resource utilizations.

7 Evaluation

7.1 Client-side Encryption Performance

Setup. For hardware acceleration on power-constrained IoT endpoints, we synthesize the complete LWR encryption datapath (Section 6) integrating three CSPRNG variants: SHAKE-256, baseline Kreyvium, and our optimized Kreyvium (Section 6.2). Each variant is realized as a Beethoven accelerator core on an inexpensive, passively cooled Xilinx Zynq UltraScale+ ZU3 SoC (< 5 W total power budget) clocked at 100 MHz. The edge software baseline is our optimized Rust implementation of Section 6.1 on the same SoC's ARM Cortex-A53 processor. All configurations enforce 128-bit security parameters [22].

FPGA Throughput and Area Efficiency. Figure 9 plots encryption throughput against FPGA resource utilization on the ZU3. Note that because LWR operates as a stream cipher over independent chunks, encryption runtime scales linearly with payload size. Kreyvium's compact datapath allows a single core to occupy 65% less area than SHAKE-256, making it the only cipher capable of fitting within the severely constrained 15% utilization budget window (the shaded region).

While baseline Kreyvium exhibits low per-core throughput (~18.6 k ctxt/s), our pipelined optimizations flip this trade-off: a single optimized Kreyvium core delivers ~201.7 k ctxt/s, surpassing a multi-cycle SHAKE-256 core with far fewer logic cells. At full chip utilization, the ZU3 accommodates 16 optimized Kreyvium cores, achieving an aggregate throughput of 2.16×10^6 ciphertexts per second. This translates to a 5,092× speedup over the Cortex-A53 CPU software baseline and a 5.9× speedup over the 20-core baseline Kreyvium implementation, enabling line-rate LWR encryption even on low-power embedded devices for the first time.

Client Experience on Commodity Endpoints. For everyday web browsing, clients do not require specialized hardware accelerators. On an Apple MacBook Air (M4 chip), the software implementation sustains an encryption throughput of 39 Mbps within a 10 W power budget. The client-side overhead remains almost imperceptible for human-interactive web traffic.

7.2 Ruleset Coverage

In this section, we evaluate the coverage of RegEx and payload inspection in a DPI ruleset. Specifically, we examine the publicly available portion of Proofpoint's ruleset [52]. This ruleset is adopted by Suricata [50], a modern production DPI engine that supersedes the legacy Snort [55] used in prior work [32, 48, 58].

We focus on the web application category of the ruleset, which is the largest category and contains 5,893 active rules across 222 applications, covering standard web-attack categories such as SQL injection, XSS, and remote code execution. Nevertheless, PhantomBox's compiler optimizations are generalizable to any PCRE rulesets because they exploit common structural features in intrusion-detection signatures rather than ruleset-specific heuristics. After removing 66 rules that either inspect the response direction or carry no constraint, we consider the 5,827 rules that the middlebox would evaluate on an incoming request.

Rule structure. To accelerate rule matching, each Suricata rule includes a pre-filter that checks request metadata such as IP address, port, and URL. Only requests that pass the pre-filter proceed to RegEx evaluation. The RegEx could be applied to both metadata (*e.g.*, URI) and the payload. PhantomBox inherits this staged structure. However, since metadata is typically considered public information [41], PhantomBox performs homomorphic RegEx matching only on the payload and still runs plaintext pre-filtering on the metadata.

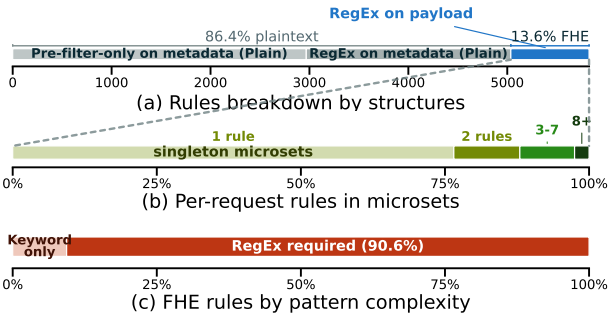


Figure 10. Rule breakdown. *(a)* Of the 5,827 rules, 13.6% require FHE evaluation. *(b)* Zoom-in on the distribution of FHE-required rule microsets: 76.6% are singleton and 11.6% have two rules. *(c)* Of the FHE-required rules, 90.6% go beyond keyword matching.

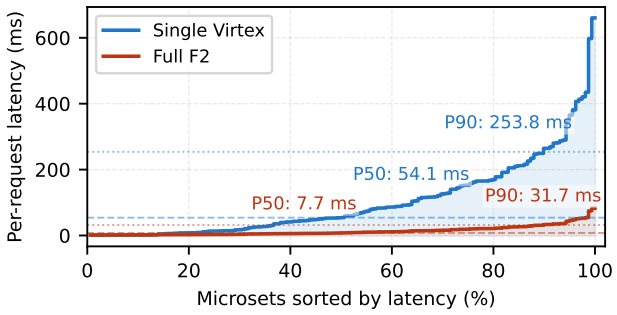


Figure 11. Inverse CDF of per-request latency introduced by middlebox computation over rule microsets.

FHE-based RegEx coverage. As Figure 10(a) shows, 86.4% of rules examine only the metadata, either via a pre-filter (50.9%) or via plaintext RegEx (35.5%), and are thus evaluated without FHE. The remaining 13.6% apply RegEx to the payload, which may contain sensitive data, and are therefore evaluated using FHE-based RegEx in PhantomBox. This considerable fraction confirms that privacy-preserving payload inspection is a practical necessity rather than a corner case.

Per-request FHE microset. Given the ruleset, a request does not necessarily trigger every FHE-required rule in the 13.6% portion. Typically, the FHE-required rules that share the same plaintext pre-filter form a ***microset*** against which a request's payload is checked. As Figure 10(b) shows, 76.6% of microsets are singletons and 11.6% have two rules, meaning a request typically triggers FHE evaluation fewer than twice in PhantomBox. Actually, this observation grants a relaxed latency budget for per-request FHE processing in Internet applications. We will evaluate PhantomBox's middlebox performance based on these microsets (Section 7.3).

Necessity of a RegEx engine. Only 9.4% of rules in Figure 10(c) consist of pure keyword patterns matchable by prior keyword-only privacy-preserving WAFs [14, 39, 48, 58, 71], motivating the need for a RegEx engine.

7.3 Middlebox Performance

Given the per-request rule microsets characterized in Section 7.2, we now evaluate the per-request latency of the FHE-based RegEx engine on the middlebox.

Setup. We deploy the middlebox on an AWS EC2 F2 instance equipped with 8 AMD Virtex UltraScale+ VU47P FPGAs, each with 16 GB of HBM. The FHE backend uses the Homomorphic Processing Unit (HPU) architecture from Zama [74].[8] We report results for both a single Virtex FPGA and the full F2 instance (8 FPGAs); the workload scales linearly across devices[9].

Measurement scope. The reported latency covers the full middlebox computation path: middlebox-side transciphering, runtime specialization (Section 5), and FHE-based RegEx evaluation, excluding client-side encryption (Section 7.1) and network transmission. Runtime specialization is negligible relative to FHE execution (50–150 μs depending on pattern complexity). Rules for plaintext (Figure 10(a)) are excluded from all figures, as they require no FHE and take microseconds via the conventional Hyperscan [61] path.

Latency distribution. Figure 11 plots the inverse CDF of per-request latency across the rule microsets. On a single FPGA, the median microset latency is 54 ms, and 90% complete within 254 ms. The full F2 instance achieves a median latency of 7.7 ms and P90 of 32 ms, a ~7× improvement reflecting near-linear throughput scaling. For security-sensitive Internet applications, these numbers indicate that a typical incoming request triggers FHE RegEx evaluation that completes within a reasonable latency, given FPGA acceleration on the middlebox. Dedicated FHE ASICs [21, 34, 37, 51, 53, 57, 70, 76] are projected to deliver another order of magnitude, placing PhantomBox well into the sub-millisecond regime.

Ablation study of the compiler. Figure 12 shows the cumulative effect of each optimization: The full pipeline achieves a median 6.5× speedup over the unoptimized baseline, with the e-graph optimization contributing the largest single improvement. Figure 13 isolates the effect of inter-rule structural caching, which eliminates redundant PBS evaluations for sub-trees shared across rules and starting positions. Caching is orthogonal to the compile-time optimizations and compounds with them: the full pipeline benefits most because post-optimization patterns exhibit tighter structural sharing.

[8] HPU originally targets the AMD Alveo V80; we backported it to F2. Compiler: Rust 1.94, TFHE-rs 1.5 [73], egg 0.9 [66]; FPGA backend: HPU v1.2.
[9] PBS operations are almost perfectly parallel across FPGAs, with per-app latency bounded below by the program's sequential depth.

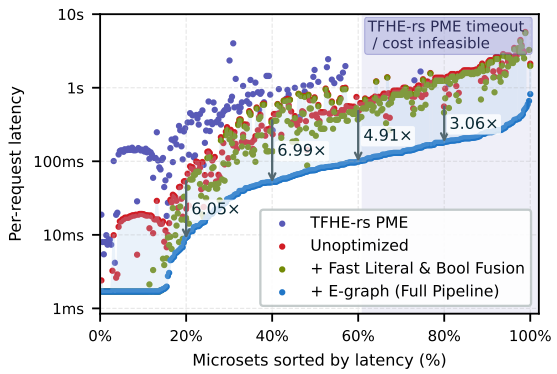


Figure 12. Ablation study of compiler optimizations. Each point represents a microset; all curves share the same x-axis (microset sorted by full-pipeline latency). Annotated ratios show the speedup of the full pipeline over the unoptimized one at the 20th, 40th, 60th, and 80th percentiles.

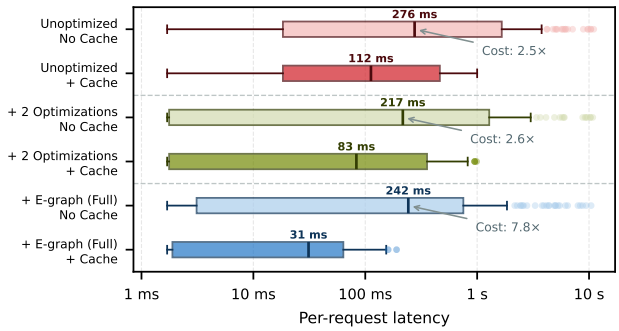


Figure 13. Effect of inter-rule structural caching across three compiler configurations. Each box summarizes the per-request latency distribution.

AOT compilation time. The AOT compilation of the full set of FHE-required rules takes ~2 minutes on an AMD EPYC 7R13 CPU. Since rulesets are updated infrequently (daily [3]), this one-time cost introduces negligible overhead to the middlebox.

7.4 Trade-Off Comparison with Prior Work

As summarized in Table 3, prior encrypted middleboxes trade off rule expressiveness, client feasibility, and network overhead.

Head-to-head with ZKP (Zombie). Evaluating the full Proofpoint ruleset under ZKP is not feasible because existing systems support only simple, fixed patterns. Zombie [75] supports only five keyword-and-digit-only rules while requiring 409 s of client proving time on a 16-core server CPU, ~10 GB RAM, and 120–240× payload inflation for a 1 MB payload.

Table 3. Cross-paradigm comparison of privacy-preserving middleboxes.

System (Type)	Rule Support	Client Cost (100KB)	Payload Size	MB Latency	Ruleset Private?
PhantomBox (FHE)	**Full PCRE**	**25 ms** (Laptop)	**~ 1 ×**	**7.7 ms**	**Yes** (LWE)
mbTLS [46] (TEE)	Full PCRE	~ μs (TLS)	~ 1×	~100 μs	Yes (TEE)
Zombie [75] (ZKP)	PCRE Subset	409 s (16-c EPYC)	120–240×	4.9 s	No
zkMB [27] (ZKP)	Keywords	25 min (32-c EPYC)	1.2–1.5×	~ms	No
BlindBox [58] (GC)	Keywords	97 s for Setup Time	1.1–14×	33 μs	Yes (Garbled)
Embark [39] (GC)	Keywords	3000 h for Setup Time	~5×	~ms	Yes (Garbled)

Example: Kubix SQL Injection (original)

```
alert http $EXTERNAL_NET -> $HTTP_SERVER                ✓ protocol
content:"/index.php?"; http_uri;                        ✓ path
content:"member_id="; http_uri;                         ✓ param name
content:"SELECT"; http_uri;                             ✗ sensitive payload
```

FHE rewriting options:

```
pcre:"/ member_id=.* SELECT.+FROM/Ui";                  ✗ bridge (bad perf)
pcre:"/SELECT.+FROM/Ui"; on member_id value             ✓ isolated
```

Figure 14. A SQL injection rule illustrating the metadata-to-payload bridge problem. The third keyword targets the user payload. Isolating the regex to the payload parameter eliminates the expensive FHE bridge without leaking plaintext.

For a quantitative comparison, we ran PhantomBox on Zombie's exact five benchmark rules: PhantomBox executes in 30 ms vs. Zombie's 4.9 s (**163× faster**).

Latency vs. Hardware Trust (mbTLS). TEE-based middleboxes such as mbTLS [46] achieve native ~100 μs latency by executing plaintext RegEx inside enclaves. PhantomBox eliminates hardware vendor trust and side-channel vulnerabilities entirely, relying solely on LWE hardness.

Overhead vs. MPC and Garbled Circuits. Interactive MPC [36] requires one round-trip per character. Across a typical 36 ms client-CDN RTT [41], inspecting even a 100-character URL incurs > 3.6 s of network transit alone. Garbled-circuit systems (BlindBox [58], Embark [39]) avoid interactive per-character rounds but are restricted to exact keywords, inflate payload sizes by up to 14×, and require heavy setup phases (97 s in BlindBox and up to 3,000 hours in Embark). PhantomBox evaluates full PCRE with ~1× expansion and zero extra RTT.

8 Discussion

Toward privacy-aware rule engineering. Figure 11 shows a heavy tail where a handful of rules dominate per-request

cost. Profiling attributes this to any-length wildcards (`.*`) that *bridge* insensitive metadata to sensitive payload inside a single RegEx, as illustrated by the first `pcre` entry in Figure 14. Bridges arise because legacy rules assume plaintext engines access metadata and payload freely. Across the Proofpoint ruleset, bridge patterns account for every top-ten rule with the highest latency, while non-bridged patterns targeting the same attack classes cost only 3–100 PBS.

Treating payload keywords as plaintext prefilters leaks user queries, while folding metadata into encrypted regexes introduces expensive wildcard bridges. Figure 14 illustrates the fix. The solution is fine-grained buffer isolation: keeping metadata as plaintext prefilters while restricting the regex to the isolated payload value (typically 10× shorter). Bridges account for 92% of PBS above the 90th percentile, whereas privacy-aware isolation eliminates this tail. This optimization exceeds any compiler-level optimization in Section 4.2, but requires rule authors to partition boundaries explicitly.

Compatibility with HTTPS. A naive construction requires encrypting HTTP payloads with LWR, diverging from standard TLS, which encrypts payloads using traditional symmetric ciphers (AES-GCM, ChaCha20). Following InviCloak [41], PhantomBox uses a two-layer encryption model: LWR acts as an *inner* application-level layer inside standard TLS. The client encrypts only sensitive fields (*e.g.*, query strings) with LWR, leaving routing metadata visible inside the TLS session. The middlebox unwraps TLS, inspects plaintext metadata for prefiltering, and evaluates FHE rules over the inner LWR ciphertext without modifying TLS or breaking HTTPS compliance.

Client Orchestration and Opt-In Deployment. PhantomBox is deployed on an *opt-in, per-application basis*, much like CDN adoption. Participating application providers distribute an inspection manifest alongside a client-side script (*e.g.*, a browser Service Worker or WebAssembly module [73]). The manifest specifies URL patterns and field selectors requiring private inspection (*e.g.*, `/api/v1/checkout/*`). For matching requests, the client runtime encrypts designated payload fields with LWR before passing them to the browser's standard TLS stack; non-matching requests or visits to non-participating servers fall back to standard HTTPS. This design requires no TLS or browser modifications and enables incremental adoption.

9 Conclusion

We presented PhantomBox, the first FHE-based WAF that simultaneously achieves full RegEx expressiveness, middlebox offloading, and cryptographic privacy. By co-designing the datapath from client-side transciphering through ahead-of-time rule compilation, PhantomBox evaluates industrial signatures at a median latency of 7.7 ms. This result moves FHE-based traffic inspection from theoretical curiosity to deployable infrastructure.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their valuable feedback. Shihan Lin is the corresponding author. This work was supported in part by a National Science Foundation CAREER Award CCF-2045974.

References

[1] [n. d.]. Akamai App & API Protector. https://www.akamai.com/resources/product-brief/app-and-api-protector
[2] [n. d.]. Cloudflare Web Application Firewall. https://developers.cloudflare.com/waf
[3] [n. d.]. ET OPEN Ruleset Download Instructions. https://rules.emergingthreats.net/OPEN_download_instructions.html
[4] [n. d.]. Let's Encrypt Stats. https://letsencrypt.org/stats/
[5] [n. d.]. What is a WAF? | Web Application Firewall Explained. https://www.cloudflare.com/learning/ddos/glossary/web-application-firewall-waf/
[6] Ahmad Al Badawi, Yuriy Polyakov, Khin Mi Mi Aung, Bharadwaj Veeravalli, and Kurt Rohloff. 2021. Implementation and Performance Evaluation of RNS Variants of the BFV Homomorphic Encryption Scheme. *IEEE Transactions on Emerging Topics in Computing* 9, 2 (April 2021), 941–956. doi:10.1109/TETC.2019.2902799
[7] Martin R. Albrecht, Rachel Player, and Sam Scott. 2015. On the concrete hardness of Learning with Errors. *Journal of Mathematical Cryptology* 9, 3 (Oct. 2015), 169–203. doi:10.1515/jmc-2015-0016
[8] Emir Beganović. 2019. *Analysing Global CDN Performance.* https://labs.ripe.net/Members/emirb/analysing-global-cdn-performance
[9] Sonia Belaïd, Nicolas Bon, Aymen Boudguiga, Renaud Sirdey, Daphné Trama, and Nicolas Ye. 2025. Further Improvements in AES Execution over TFHE. 2, 1 (2025), cc2–1–65. doi:10.62056/ahmp-4tw9
[10] Karthikeyan Bhargavan, Ioana Boureanu, Antoine Delignat-Lavaud, Pierre-Alain Fouque, and Cristina Onete. 2018. A Formal Treatment of Accountable Proxying Over TLS. In *2018 IEEE Symposium on Security and Privacy (SP)* (San Francisco, CA, 2018-05). IEEE, 799–816. doi:10.1109/SP.2018.00021
[11] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *31st USENIX Security Symposium (USENIX Security 22)* (Boston, MA, 2022-08). USENIX Association, 3917–3934. https://www.usenix.org/conference/usenixsecurity22/presentation/borrello
[12] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. *Software Grand Exposure: SGX Cache Attacks Are Practical.* arXiv:1702.07521 [cs] doi:10.48550/arXiv.1702.07521
[13] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)* (Baltimore, MD, 2018-08). USENIX Association, 991–1008. https://www.usenix.org/conference/usenixsecurity18/presentation/bulck
[14] Sébastien Canard, Aïda Diop, Nizar Kheir, Marie Paindavoine, and Mohamed Sabt. 2017. BlindIDS: Market-Compliant and Privacy-Friendly Intrusion Detection System over Encrypted Traffic. In *Proceedings of ACM AsiaCCS'17*. 561–574.
[15] Frank Cangialosi, Taejoong Chung, David Choffnes, Dave Levin, Bruce M Maggs, Alan Mislove, and Christo Wilson. 2016. Measurement and Analysis of Private Key Sharing in the HTTPS Ecosystem. In *Proceedings of CCS'16*. ACM, 628–640.
[16] Anne Canteaut, Sergiu Carpov, Caroline Fontaine, Tancrède Lepoint, María Naya-Plasencia, Pascal Paillier, and Renaud Sirdey. 2018. Stream

Ciphers: A Practical Solution for Efficient Homomorphic-Ciphertext Compression. 31, 3 (2018), 885–916. doi:10.1007/s00145-017-9273-9

[17] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2019. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)* (2019-06). 142–157. arXiv:1802.09085 [cs] doi:10.1109/EuroSP.2019.00020

[18] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2019. A Full RNS Variant of Approximate Homomorphic Encryption. In *Selected Areas in Cryptography – SAC 2018*, Carlos Cid and Michael J. Jacobson (Eds.). Vol. 11349. Springer International Publishing, 347–368. doi:10.1007/978-3-030-10970-7_16

[19] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology – ASIACRYPT 2017*, Tsuyoshi Takagi and Thomas Peyrin (Eds.). Vol. 10624. Springer International Publishing, 409–437. doi:10.1007/978-3-319-70694-8_15

[20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: Fast Fully Homomorphic Encryption Over the Torus. 33, 1 (2020), 34–91. doi:10.1007/s00145-019-09319-x

[21] Xianglong Deng, Shengyu Fan, Zhicheng Hu, Zhuoyu Tian, Zihao Yang, Jiangrui Yu, Dingyuan Cao, Dan Meng, Rui Hou, Meng Li, Qian Lou, and Mingzhe Zhang. 2024. *Trinity: A General Purpose FHE Accelerator.* arXiv:2410.13405 [cs] doi:10.48550/arXiv.2410.13405

[22] Amit Deo, Marc Joye, Benoît Libert, Benjamin R. Curtis, and Mayeul De Bellabre. 2025. Fast Homomorphic Evaluation of LWR-based PRFs. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security* (Taipei Taiwan, 2025-11-19). ACM, 2951–2965. doi:10.1145/3719027.3765177

[23] Léo Ducas and Daniele Micciancio. 2015. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In *Advances in Cryptology – EUROCRYPT 2015*, Elisabeth Oswald and Marc Fischlin (Eds.). Vol. 9056. Springer Berlin Heidelberg, Berlin, Heidelberg, 617–640. doi:10.1007/978-3-662-46800-5_24

[24] Jingyuan Fan, Chaowen Guan, Kui Ren, Yong Cui, and Chunming Qiao. 2017. SPABox: Safeguarding Privacy during Deep Packet Inspection at a Middlebox. *IEEE/ACM Transactions on Networking* 25, 6 (2017), 3753–3766.

[25] Rusins Freivalds. 1977. Probabilistic Machines Can Use Less Running Time. In *IFIP Congress*. https://api.semanticscholar.org/CorpusID:45405368

[26] David Gillman, Yin Lin, Bruce Maggs, and Ramesh K Sitaraman. 2015. Protecting Websites from Attack with Secure Delivery Networks. *Computer* 48, 4 (2015), 26–34.

[27] Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish. 2022. Zero-Knowledge Middleboxes. In *31st USENIX Security Symposium (USENIX Security 22)* (Boston, MA, 2022-08). USENIX Association, 4255–4272. https://www.usenix.org/conference/usenixsecurity22/presentation/grubbs

[28] Antonio Guimarães, Gabriela M. Jacob, and Hilder V. L. Pereira. 2026. PETCHA: Post-quantum Efficient Transciphering with ChaCha. In *Post-Quantum Cryptography* (Cham, 2026), Magali Bardet and Ruben Niederhagen (Eds.). Springer Nature Switzerland, 178–203.

[29] William G Halfond, Jeremy Viegas, Alessandro Orso, et al. 2006. A Classification of SQL-Injection Attacks and Countermeasures. In *Proceedings of International Symposium on Secure Software Engineering*, Vol. 1. IEEE, 13–15.

[30] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. 2019. Logistic Regression on Homomorphic Encrypted Data at Scale. *Proceedings of the AAAI Conference on Artificial Intelligence* 33, 01 (July 2019), 9466–9471. doi:10.1609/aaai.v33i01.33019466

[31] Stephen Herwig, Christina Garman, and Dave Levin. 2020. Achieving Keyless CDNs with Conclaves. In *Proceedings of USENIX Security Symposium 2020*. USENIX, 735–751.

[32] Xiaoyang Hou, Jian Liu, Tianyu Tu, Rui Zhang, and Kui Ren. 2024. PrivRE: Regular Expression Matching for Encrypted Packet Inspection. In *Proceedings of IEEE ICDCS'24*. IEEE, 1306–1317.

[33] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. 2004. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the 13th International Conference on World Wide Web* (New York NY USA, 2004-05-17). ACM, 40–52. doi:10.1145/988672.988679

[34] Lei Jiang, Qian Lou, and Nrushad Joshi. 2022. MATCHA: A Fast and Energy-Efficient Accelerator for Fully Homomorphic Encryption over the Torus. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco California, 2022-07-10). ACM, 235–240. doi:10.1145/3489517.3530435

[35] Marc Joye. 2021. Guide to Fully Homomorphic Encryption over the [Discretized] Torus. https://eprint.iacr.org/2021/1402. cryptology eprint archive:2021/1402

[36] Florian Kerschbaum. 2006. Practical Private Regular Expression Matching. In *IFIP International Information Security Conference*. Springer, 461–470.

[37] Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. 2023. SHARP: A Short-Word Hierarchical Accelerator for Robust and Practical Fully Homomorphic Encryption. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando FL USA, 2023-06-17). ACM, 1–15. doi:10.1145/3579371.3589053

[38] Jakub Klemsa. 2022. Hitchhiker's Guide to the TFHE Scheme. https://eprint.iacr.org/2022/1315. cryptology eprint archive:2022/1315

[39] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. 2016. Embark: Securely Outsourcing Middleboxes to the Cloud. In *Proceedings of USENIX NSDI'16*. 255–273.

[40] Shihan Lin, Suting Chen, Yunming Xiao, Yanqi Gu, Aleksandar Kuzmanovic, and Xiaowei Yang. 2025. PreAcher: Secure and Practical Password Pre-Authentication by Content Delivery Networks. In *Proceedings of NSDI'25*. USENIX.

[41] Shihan Lin, Rui Xin, Aayush Goel, and Xiaowei Yang. 2022. InviCloak: An End-to-End Approach to Privacy and Performance in Web Content Distribution. In *Proceedings of CCS'22*. ACM, 1947–1961.

[42] Jiaao Ma, Ceyu Xu, and Lisa Wu Wills. 2023. PyTFHE: An End-to-End Compilation and Execution Framework for Fully Homomorphic Encryption Applications. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, Raleigh, NC, USA, 24–34. doi:10.1109/ISPASS57527.2023.00012

[43] Payman Mohassel, Salman Niksefat, Saeed Sadeghian, and Babak Sadeghiyan. 2012. An Efficient Protocol for Oblivious DFA Evaluation and Applications. In *Cryptographers' Track at the RSA Conference*. Springer, 398–415.

[44] National Institute of Standards and Technology. 2001. *Announcing the Advanced Encryption Standard (AES)*. Federal Information Processing Standards Publication FIPS PUB 197. U.S. Department of Commerce. https://csrc.nist.gov/pubs/fips/197/final

[45] National Institute of Standards and Technology (US). 2015. SHA-3 Standard : Permutation-Based Hash and Extendable-Output Functions. error: 202 pages. doi:10.6028/NIST.FIPS.202

[46] David Naylor, Richard Li, Christos Gkantsidis, Thomas Karagiannis, and Peter Steenkiste. 2017. And Then There Were More: Secure Communication for More Than Two Parties. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies* (Incheon Republic of Korea, 2017-11-28). ACM, 88–100. doi:10.1145/3143361.3143383

[47] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R. López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. 2015. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London United Kingdom, 2015-08-17). ACM, 199–212.

doi:10.1145/2785956.2787482

[48] Jianting Ning, Geong Sen Poh, Jia-Ch'ng Loh, Jason Chia, and Ee-Chien Chang. 2019. PrivDPI: Privacy-Preserving Encrypted Traffic Inspection with Reusable Obfuscated Rules. In *Proceedings of ACM CCS'19*. 1657–1670.

[49] Yoav Nir and Adam Langley. 2015. ChaCha20 and Poly1305 for IETF Protocols. RFC 7539. doi:10.17487/RFC7539

[50] Open Information Security Foundation. 2026. Suricata: Open Source Intrusion Detection and Prevention System. https://suricata.io. Accessed: 2026-04-14.

[51] Prasetiyo, Adiwena Putra, and Joo-Young Kim. 2024. Morphling: A Throughput-Maximized TFHE-based Accelerator Using Transform-domain Reuse. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (Edinburgh, United Kingdom, 2024-03-02). IEEE, 249–262. doi:10.1109/HPCA57654.2024.00028

[52] Proofpoint, Inc. 2020. *Proofpoint ET Pro Ruleset*. https://www.proofpoint.com/sites/default/files/pfpt-us-ds-et-pro-ruleset.pdf

[53] Adiwena Putra, Prasetiyo, Yi Chen, John Kim, and Joo-Young Kim. 2023. Strix: An End-to-End Streaming Architecture with Two-Level Ciphertext Batching for Fully Homomorphic Encryption with Programmable Bootstrapping. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2023) *(Micro '23)*. Association for Computing Machinery, 1319–1331. doi:10.1145/3613424.3614264

[54] Oded Regev. 2009. On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. 56, 6 (2009), 1–40. doi:10.1145/1568318.1568324

[55] Martin Roesch. 1999. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Conference on System Administration (LISA)*.

[56] Lorenzo Rovida and Alberto Leporati. 2024. Encrypted Image Classification with Low Memory Footprint Using Fully Homomorphic Encryption. 34, 05 (2024), 2450025. doi:10.1142/S0129065724500254

[57] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2021) *(Micro '21)*. Association for Computing Machinery, 238–252. doi:10.1145/3466752.3480070

[58] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. 2015. Blindbox: Deep Packet Inspection over Encrypted Traffic. In *Proceedings of ACM SIGCOMM'15*. 213–226.

[59] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *2020 IEEE Symposium on Security and Privacy (SP)* (San Francisco, CA, USA, 2020-05). IEEE, 54–72. doi:10.1109/SP40000.2020.00089

[60] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. Vectorization for Digital Signal Processors via Equality Saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual USA, 2021-04-19). ACM, 874–886. doi:10.1145/3445814.3446707

[61] Yang Wang, Junqing Kong, Tian Hua, Cong Wang, Qian Wang, and Kui Ren. 2019. Hyperscan: A Fast Multi-Pattern Regex Matcher for Modern CPUs. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 631–648.

[62] Lei Wei and Michael K Reiter. 2012. Third-party Private DFA Evaluation on Encrypted Files in the Cloud. In *Proceedings of European Symposium on Research in Computer Security (ESORICS'12)*. Springer, 523–540.

[63] Lei Wei and Michael K Reiter. 2013. Ensuring File Authenticity in Private DFA Evaluation on Encrypted Files in the Cloud. In *Proceedings of European Symposium on Research in Computer Security (ESORICS'13)*. Springer, 147–163.

[64] Lei Wei and Michael K Reiter. 2015. Toward Practical Encrypted Email that Supports Private Regular-Expression Searches. *International Journal of Information Security* 14, 5 (2015), 397–416.

[65] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song. 2011. A Systematic Analysis of XSS Sanitization in Web Application Frameworks. In *Proceedings of ESORICS*. Springer, 150–171.

[66] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29. doi:10.1145/3434304

[67] Rui Xin, Shihan Lin, and Xiaowei Yang. 2023. Quantifying User Password Exposure to Third-Party CDNs. In *Proceedings of PAM'23*. Springer, 652–668.

[68] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy* (San Jose, CA, 2015-05). IEEE, 640–656. doi:10.1109/SP.2015.45

[69] Yichen Yang, Phitchaya Mangpo Phothilimthana, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. (2021).

[70] Yinghao Yang, Huaizhi Zhang, Shengyu Fan, Hang Lu, Mingzhe Zhang, and Xiaowei Li. 2023. Poseidon: Practical Homomorphic Encryption Accelerator. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (Montreal, QC, Canada, 2023-02). IEEE, 870–881. doi:10.1109/HPCA56546.2023.10070984

[71] Xingliang Yuan, Xinyu Wang, Jianxiong Lin, and Cong Wang. 2016. Privacy-preserving deep packet inspection in outsourced middleboxes. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 1–9.

[72] Zama. 2022. Concrete: TFHE Compiler That Converts Python Programs into FHE Equivalent. https://github.com/zama-ai/concrete.

[73] Zama. 2022. TFHE-rs: a pure rust implementation of the TFHE scheme for boolean and integer arithmetics over encrypted data. https://github.com/zama-ai/tfhe-rs

[74] Zama. 2025. A SystemVerilog implementation of the Homomorphic Processing Unit (HPU) targeting AMD Alveo V80 FPGA board. https://github.com/zama-ai/hpu_fpga.

[75] Collin Zhang, Zachary DeStefano, Arasu Arun, Joseph Bonneau, Paul Grubbs, and Michael Walfish. 2024. Zombie: Middleboxes That Don't Snoop. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)* (Santa Clara, CA, 2024-04). USENIX Association, 1917–1936. https://www.usenix.org/conference/nsdi24/presentation/zhang-collin

[76] Minxuan Zhou, Yujin Nam, Xuan Wang, Youhak Lee, Chris Wilkerson, Raghavan Kumar, Sachin Taneja, Sanu Mathew, Rosario Cammarota, and Tajana Rosing. 2024. UFC: A Unified Accelerator for Fully Homomorphic Encryption. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Austin, TX, USA, 2024-11-02). IEEE, 352–365. doi:10.1109/MICRO61859.2024.00034