Zombie: Middleboxes that Don't Snoop

Collin Zhang,[†] Zachary DeStefano,[*] Arasu Arun,[*] Joseph Bonneau,[*] Paul Grubbs,[‡] Michael Walfish[*]

[*]NYU [†]Cornell (work done mostly at NYU) [‡]University of Michigan

Abstract. Zero-knowledge middleboxes (ZKMBs) are a recent paradigm in which clients get privacy while middleboxes enforce policy: clients prove in zero knowledge that the plaintext underlying their encrypted traffic complies with network policies, such as DNS filtering. However, prior work had impractically poor performance and was limited in functionality.

This work presents Zombie, the first system built using the ZKMB paradigm. Zombie introduces techniques that push ZKMBs to the verge of practicality: preprocessing (to move the bulk of proof generation to idle times between requests), asynchrony (to remove proving and verifying costs from the critical path), and batching (to amortize some of the verification work). Zombie's choices, together with these techniques, reduce client and middlebox overhead by $\approx 3.5\times$, lowering the critical path overhead for a DNS filtering application on commodity hardware to less than 300ms or, in the asynchronous configuration, to 0.

As an additional contribution that is likely of independent interest, Zombie introduces a portfolio of techniques to encode regular expressions in probabilistic (and zero-knowledge) proofs. These techniques significantly improve performance over a standard baseline, asymptotically and concretely. Zombie builds on this portfolio to support policies based on regular expressions, such as data loss prevention.

1 Introduction

A fundamental conflict frequently arises in network security: administrators' policy enforcement vs. users' privacy. Organizations want, or in some cases need (by legal obligation), to enforce network usage policies. Users want end-to-end encrypted protocols like TLS to provide privacy against network observers, including administrators. Traditionally, policy enforcement requires a middlebox to scan traffic and block policy-violating use. End-to-end encryption is in direct conflict with middleboxes, which can't see plaintext and therefore can't assess policy compliance. This conflict has led some administrators to take draconian steps, like inserting themselves as an all-seeing middleperson (MITM) proxying TLS connections ("split TLS"), or even blocking the use of TLS.

Resolving this conflict has been a goal of the network security research community for some time. Existing approaches have fallen into two categories. First are protocols that use novel cryptography to enable policy checks on encrypted data, but require server support and/or changes to standard protocols like TLS [63, 79, 98] (§7). Changing TLS is a huge task though: it took ten years of extensive design effort to go from TLS 1.2 [30] to TLS 1.3 [90]. Deploying server-side changes is also slow: five years after the standardization of TLS 1.3, only 60% of HTTPS servers on the web support it [60]. Furthermore, implementing TLS securely is notoriously complex and subtle [24], meaning that protocol changes are risky. Second, by contrast, are middleboxes designed to work with standard TLS-encrypted traffic but rely on trusted hardware enclaves (TEEs) to enforce policy [31, 44]. Our perspective is that, although it promises good performance, trusted hardware has a wide attack surface (§7), as demonstrated by many attacks [47, 77, 92, 102, 109–112].

Our goal is to support policy enforcement on standard TLS 1.3 traffic, inheriting its existing security guarantees and avoiding any changes to existing TLS code bases. We eschew any trusted hardware assumptions. We do, however, accept modifications to clients, observing that modern browser vendors can update the vast majority of users within months [114].

Zero-knowledge middleboxes. We build on top of the recently-proposed ZKMB paradigm [49]. With ZKMBs, clients prove in zero knowledge [43] to the middlebox that the plaintext underlying their encrypted traffic is policy-compliant. The middlebox verifies these proofs, allowing only policy-compliant traffic to pass. Because the proof is zero knowledge, the middlebox learns nothing about the underlying plaintext, except that it is policy-compliant. ZKMBs require no changes to existing encryption protocols and no trusted hardware; they promise an elegant solution to the policy vs. privacy conflict. However, the initial ZKMB prototype offered implausible performance for most network applications, adding several seconds of latency to traffic even under optimistic assumptions and with a relatively simple policy.

The key question remains: *Can ZKMBs perform well enough, and express a wide-enough range of policies, for real-world use?* This paper gives a cautious affirmative answer, with the design, implementation, and experimental evaluation of a system called *Zombie*.

Contributions and results. Zombie applies three techniques to reduce end-to-end delay (both client proving costs and middlebox verifying costs on the critical path). First, Zombie precomputes and pre-proves part of the encryption step in TLS, moving it off the critical path to periods when the client is idle (§3.1). This part includes legacy cryptographic primitives like ChaCha20 encryption, which, for reasons we explain later (§2, §3.1), are expensive to represent in proof frameworks. Such a split is perhaps surprising: how could a client

precompute an encryption before the plaintext is known?

Zombie's next two performance enhancements are conditional on assumptions about client and middlebox behavior. For these, our contribution is primarily analyzing and evaluating the techniques in this context, rather than the mechanics of the techniques. One of these is *optimistic approval* (§3.2), via asynchronous verification. We make the simple but consequential observation that, in many applications, administrators may be willing to allow client traffic to proceed as normal, on the condition that clients supply valid proofs in short order. A similar approach, *near real-time verification*, is already taken by some real-world middleboxes [18, 23, 38]. The other is *batch proof verification* by the middlebox, reducing the overall verification burden by amortizing it across proofs for multiple packets (§3.3).

Another set of contributions enables Zombie to handle policies based on regular expression matching, a crucial building block in various middleboxes, including intrusion detection systems (IDS), network traffic classification, and data loss prevention (DLP). The core challenge derives from probabilistic proofs themselves: using these frameworks requires representing the target computation in arithmetic circuits or constraints. Meanwhile, as with ChaCha20 mentioned earlier, circuits and constraints are inefficient and inhospitable for many computations (§2), including (at first glance) regular expressions. Zombie tackles this challenge with a collection of techniques (§4), including a new encoding of substring matching in arithmetic constraints, a new encoding of Boolean algebra in arithmetic constraints, and a new finite automaton formalism. Some of these techniques are likely to be of independent interest for other applications of probabilistic proofs, even beyond regular-expression matching.

We implement Zombie for TLS 1.3 with the ChaCha20 cipher (§5). The result of all this work is near-practicality for some ZKMB uses (§6). In the precomputation regime, Zombie adds less than 300 ms of delay to DNS queries, which may be tolerable (§8). In the asynchronous regime, this number drops to 0. Proofs are large (30KB) but never leave the local network. Memory requirements for prover and verifier can be substantial, but small packet sizes mitigate this issue. The sticking point is middlebox resources: although throughput improves almost 5× from batching, even this improvement (380 255-byte packets/second in our experiments) is too low to imagine proof verification on every packet. Similarly, Zombie's regular expression techniques reduce the overhead of encoding real-world DLP policies in zero knowledge by over an order of magnitude; however, the resulting overhead, 1–2 ms processing delay per byte, is uncomfortably large for networking applications.

Thus, although Zombie is designed to be extensible to any read-based public policy, it is most practical for multi-packet flows, with small packets, that represent a fraction of traffic, for example enforcing a domain blocklist on a long-running connection with a DNS server or enforcing a keyword or regexp blocklist on search engine queries. Our work has other limitations (§8). Most notably, our implementation requires that policies be public, per-packet, read-only, and stateless, though these restrictions are not fundamental.

2 Background

Zero-knowledge proofs. At a high level, a ZKP is a cryptographic protocol between two parties: a *prover* and a *verifier*. The protocol pertains to a computation $\mathbf{S}$ (we also call this the "statement"), which we formulate as having two inputs $\mathbf{X}$ and $\mathbf{W}$, each a vector of variables, and producing an output $\mathbf{Y}$. We call $\mathbf{X}$ the *public input* and $\mathbf{Y}$ the *output*, respectively.

In this paper, we consider non-interactive ZKPs, which work as follows. Both the verifier and the prover agree on a computation $\mathbf{S}$. To convince the verifier that a particular $(\mathbf{X}, \mathbf{Y})$ pair known to both parties is *valid*, the prover sends the verifier a *proof* π. Validity here is defined as the existence of a *witness* $\mathbf{W}$ such that $\mathbf{S}(\mathbf{X}; \mathbf{W}) = \mathbf{Y}$ for a particular $(\mathbf{X}, \mathbf{Y})$. Following convention, we use a semicolon to separate public and private variables in statements. The proof also convinces the verifier that the prover *knows* this witness—this guarantee is called *knowledge soundness*. Moreover, it hides the witness from the verifier—this is the *zero-knowledge* guarantee. The notions of soundness and zero-knowledge have precise cryptographic definitions that we elide here; Zombie inherits these properties directly from the underlying cryptographic tools.

In general, there is a deep cryptographic literature on ZKPs; for a survey, we refer the reader to Thaler [103].

A concrete example. Consider using ZKPs to prove that an encrypted packet does not contain a DNS query for a blocked domain [49, §7]. The output $\mathbf{Y}$ is true/false, the input $\mathbf{X}$ is the encrypted packet, and the witness $\mathbf{W}$ includes the decryption key. The computation $\mathbf{S}$ asserts that the packet, after decrypting to plaintext using the decryption key and extracting the domain name, does not contain a domain in the blocklist. Under standard cryptographic assumptions, knowing $\mathbf{X}$ and $\mathbf{Y}$, without knowing $\mathbf{W}$, is insufficient to verify the correctness of $(\mathbf{X}, \mathbf{Y})$ with respect to $\mathbf{S}$. A ZKP, by contrast, convinces a verifier, who has no access to $\mathbf{W}$, of the correctness of $(\mathbf{X}, \mathbf{Y})$.

Zero-knowledge proof pipelines. Most generic ZKP schemes decompose into a *front-end* and a *back-end*. The front-end takes $\mathbf{S}$, a high-level specification of a program, for example in C code or a domain-specific language (DSL). The front-end compiles this program into an *intermediate representation*, often called a *circuit* (see below). This circuit acts as a blueprint for provers to show that a program produces specific outputs, given specific inputs.

The back-end then enables the prover to take the circuit representation of the program, along with $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{W}$, and output a proof π. The verifier also has access to a circuit representation of the program and uses the back-end, $\mathbf{X}$, and $\mathbf{Y}$ to verify a proof π, outputting a true/false value.

R1CS instances. Most modern ZKP front-ends compile programs to a generalization of arithmetic circuits called *rank-one constraint systems* (R1CS). An R1CS instance is a collection of algebraic constraints. The instance is parameterized by a finite field $\mathbb{F}$, a number of constraints m, a number of variables n, and three $m \times n$ matrices A, B, C. An input-output pair $(\mathbf{X}, \mathbf{Y})$ satisfies the R1CS instance if there exists a $\mathbf{W}$ such that for the vector $z = (\mathbf{X}, \mathbf{Y}, 1, \mathbf{W})$, $Az \circ Bz = Cz$, where the operation $\circ$ is entry-wise multiplication. Notice that an R1CS instance consists of m constraints in n variables, where each constraint $i \in \{1, \ldots, m\}$ restricts any satisfying $z = (z_1, \ldots, z_n)$ as follows:

$$(A_{i,1}z_1 + \ldots + A_{i,n}z_n) \cdot (B_{i,1}z_1 + \cdots + B_{i,n}z_n) = (C_{i,1}z_1 + \cdots + C_{i,n}z_n).$$

Following convention, we sometimes refer to an R1CS representation as a set of *constraints* or loosely as a *circuit*.

Efficiently expressing a computation as a circuit is challenging. First, the primary efficiency metric of a circuit representation is the *number of constraints*, as the back-end's costs—specifically, the prover's costs—scale linearly or superlinearly in this quantity. Second, circuits are frequently verbose, as they are algebraic constructs, not hardware circuits or a general-purpose processor.

Among other limitations, circuits do not support looping, conditionality, order comparisons, bitwise operations, or random-access memory. Compiling a high-level computation to a circuit requires the front-end to unroll all loops to their maximum iteration count, inline all function calls, represent all branches of conditionals explicitly, and then *arithmetize* each statement (translating it into constraints), often introducing additional variables [16, 17, 85, 94, 96, 115, 124].

As a simple example, consider this line of `C` code:

```
y = (x == 0);
```

where the mathematical variable x (representing the program variable `x`) is in $\mathbf{X}$ and y (likewise representing `y`) is in $\mathbf{Y}$. To compile this to constraints, one introduces a variable W in $\mathbf{W}$ and writes the following, called `EQUALS-ZERO` [96, Appx D]:

$$\left\{ \begin{array}{c} y \cdot x = 0 \\ W \cdot x = 1 - y \end{array} \right\}$$

The constraints can be satisfied if and only if y is 1 when x is 0 and y is 0 otherwise, thus enforcing the desired computation. These constraints can be expressed in the form of an R1CS instance as the following A, B, and C matrices:

$$\begin{array}{cccc} x & y & 1 & W \\ \end{array} \quad \begin{array}{cccc} x & y & 1 & W \\ \end{array} \quad \begin{array}{cccc} x & y & 1 & W \\ \end{array}$$

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 \end{bmatrix}$$

$$A \qquad\qquad B \qquad\qquad C$$

Spartan ZKP. As its back-end, Zombie uses Spartan [93], specifically the SpartanNIZK variant (which we refer to as just Spartan for simplicity). By contrast, prior work [49] used Groth16 [48]. Spartan is a non-interactive ZKP protocol that strikes an attractive balance among prover time (lower than in Groth16), verifier time (higher than Groth16 but sufficient for our purposes, §6), and proof size (again, higher than Groth16 but sufficient). Like many ZKP protocols, Spartan has a *setup phase* to generate parameters that are used in the proof protocol; in Spartan, (unlike Groth16), this phase does not require trusting any party, only a source of public randomness. Consequently, provers (clients) can use the generated parameters across different verifiers (networks).

3 Zombie's protocol

We start with the existing ZKMB paradigm [49], though our notation differs from the original. A middlebox begins with encryption protocol E (such as TLS 1.3), content type F, and policy P, with the goal of enforcing P on traffic of type F that is sent via E. The middlebox—or a third-party—defines the following subcomputations, which are composed into statements (§2) that the client proves and the middlebox verifies:

(1) A *channel-opening* subcomputation $\mathbf{S}_E$ takes as input a packet and the information required to re-derive a session key. This subcomputation outputs the decrypted packet, in the sense of delivering that decrypted packet to the next composed subcomputation; for clarity, we note that the decrypted packet itself is never available to the middlebox, which has no access to the values of the circuit wires used in the client's proof. We follow the amortized ZKMB model, which, for TLS 1.3, reuses the expensive work of this phase over multiple per-packet proofs. Specifically, $\mathbf{S}_E$ is split into $\mathbf{S}_{E.1}$ (*derive-and-commit*) and $\mathbf{S}_{E.2}$ (*decrypt*).

(2) A *parse-and-extract* subcomputation $\mathbf{S}_F$ takes as input the decrypted packet and outputs (in the sense above) a snippet of policy-relevant data from the packet.

(3) A *policy-check* subcomputation $\mathbf{S}_P$ takes as input the snippet of policy-relevant data and outputs whether or not the policy is satisfied (for example if a domain being queried is part of a blocklist or not).

Figure 1 depicts the high-level protocol. The middlebox sends $\mathbf{S}_P$ to each client when it joins the network [49]. (We discuss other deployment possibilities in Section 8.) When a client wants to communicate with a particular server, it first negotiates the shared key K using a *handshake* protocol, the transcript of which is public but the generation of which involves secrets shared between the client and the server. $\mathbf{S}_{E.1}$ re-derives this session key K by taking the handshake transcript as public input and the client's secrets as witness, and then hashes the session key to produce h_K. The client sends to the middlebox h_K and the proof π_K of $\mathbf{S}_{E.1}$. This proof

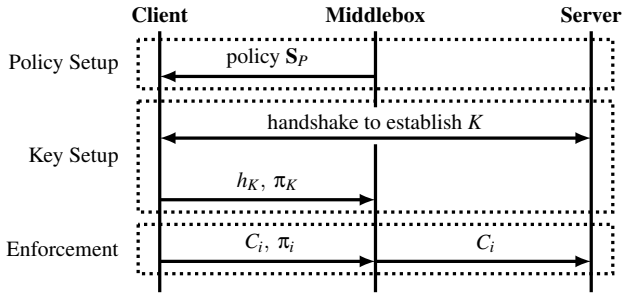


Figure 1: The ZKMB paradigm with amortized key setup [49] divided into 3 phases. The *policy setup* step occurs once when the client connects to the middlebox; the middlebox sends the policy $\mathbf{S}_P$ to the client. The *key setup* step occurs once per session; it involves a handshake between the client and the server, a commitment h_K to a session key K, and a proof of this commitment π_K. Finally, in per-packet *enforcement*, the client sends the middlebox ciphertext C_i and a proof π_i of the policy-compliance of the plaintext corresponding to C_i and to the key commitment; π_i is with reference to the composition of the $\mathbf{S}_{E.2}$, $\mathbf{S}_F$, and $\mathbf{S}_P$ subcomputations.

convinces the middlebox that h_K is indeed the commitment to some key that is consistent with the handshake.

Then, for each packet, $\mathbf{S}_{E.2}$ takes the ciphertext C and the key commitment h_K as public inputs, and the session key K as witness. After ensuring that K hashes to h_K, $\mathbf{S}_{E.2}$ outputs (again, in the sense above) the decrypted packet. Finally, the client needs to convince the middlebox that C is valid with respect to h_K and the composition of $\mathbf{S}_{E.2}$, $\mathbf{S}_F$, and $\mathbf{S}_P$. It does so with a proof π_i. When the middlebox receives (C, h_K, π_i), it verifies π_i, and only then forwards C.

Zombie's enhancements. Zombie introduces three changes to the ZKMB paradigm. *Precomputation* (§3.1) allows Zombie to generate and verify the most expensive part of the proof during idle times, before the ciphertext is known to the client, reducing proving times in the critical latency path. *Asynchronous verification* (§3.2) relaxes the requirement that proofs about traffic are verified before each packet leaves the network. This moves the main ZKP-related costs out of the critical path entirely, greatly reducing delay but changing Zombie's security model. *Batching* (§3.3) lets Zombie middleboxes reuse the results of expensive computations when verifying a batch of proofs created by the client. Figure 2 comprehensively classifies when these techniques should be used and combined.

3.1 Precomputation

Precomputation in Zombie changes both the statement being proved and the protocol flow (adding an extra message). At a high level, precomputation splits the per-packet computation $\mathbf{S}_{E.2}$ (*decrypt*) into two subcomputations $\mathbf{S}_{E.2a}$ (*pad-commit*) and $\mathbf{S}_{E.2b}$ (*decrypt-from-pad*), the first of which can be computed before the plaintext is known. As noted in the introduction, it may be surprising that it is possible to prove $\mathbf{S}_{E.2a}$

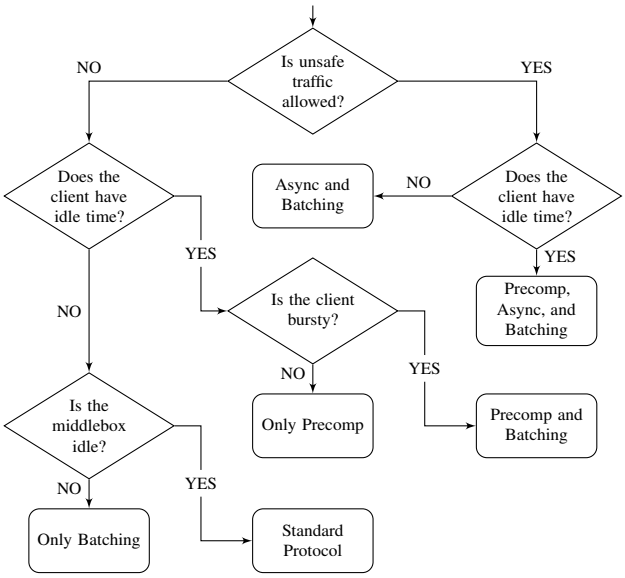


Figure 2: Appropriate combinations of protocol enhancements. The goal is to minimize client latency while respecting security. Unsafe traffic means that some non-compliant traffic is allowed to exit the network. To summarize the logic: (1) If unsafe traffic is allowed, then asynchronous mode is appropriate; (2) If clients have idle time, precomputation is appropriate; (3) If {(a) unsafe traffic is allowed, or (b) neither the client nor the middlebox has idle time, or (c) the client workload is bursty and the client has idle time} then batching is appropriate.

before any plaintext is known, but when using stream ciphers the keystream can be generated (and proved) independently of the plaintext. We explain how this works for TLS 1.3 below. The technique may be more broadly relevant; it is orthogonal (and complementary) to the split between $\mathbf{S}_{E.1}$ and $\mathbf{S}_{E.2}$.

We will simplify by omitting some operations. Let K be the session key output by the handshake. TLS 1.3 encrypts session data using a stream cipher, which can be thought of as a pseudorandom one-time pad. This pad is derived via a function PadGen that takes K, a packet number SN, and a length L_{pad}, and outputs an L_{pad}-byte pseudorandom $\mathsf{pad}_{\mathsf{SN}}$. For an L_{pad}-byte message M, its ciphertext is $\mathsf{pad}_{\mathsf{SN}} \oplus M$.

We make two key observations. First, the inputs to PadGen are independent of the message, so $\mathsf{pad}_{\mathsf{SN}}$ can be computed by the client before M is known. Second, computing PadGen is the most expensive part of the channel-opening subcomputation $\mathbf{S}_E$. This is because PadGen involves the legacy stream cipher ChaCha20, which is difficult to represent in the constraint formalism used for ZKPs (§2). In our evaluation of a DNS filtering application [49] (§6), we find that PadGen accounts for nearly two-thirds of the total constraint size (§6.3).

Zombie uses these observations to move PadGen into the subcomputation $\mathbf{S}_{E.2a}$ (*pad-commit* in Figure 3). This involves running PadGen and then hashing its output to produce $h_{\mathsf{pad}_{\mathsf{SN}}}$. Also, K is hashed to ensure that it corresponds to the hash h_K provided as a public input. The client computes the

pad-commit$(h_K, \mathsf{SN}, L_{pad}; K)$:

$\mathsf{pad}_{\mathsf{SN}} \leftarrow \mathsf{PadGen}(K, \mathsf{SN}, L_{pad})$

$h_{\mathsf{pad}_{\mathsf{SN}}} \leftarrow H(\mathsf{pad}_{\mathsf{SN}})$

Return $(h_K = H(K))\ ?\ h_{\mathsf{pad}_{\mathsf{SN}}} : \perp$

decrypt-from-pad$(C, h_{\mathsf{pad}_{\mathsf{SN}}}, \mathsf{SN}; \mathsf{pad})$:

$M \leftarrow \mathsf{pad} \oplus C$

Return $(h_{\mathsf{pad}_{\mathsf{SN}}} = H(\mathsf{pad}))\ ?\ M : \perp$

Figure 3: Pseudocode for the statements $\mathbf{S}_{E.2a}$ (*pad-commit*) and $\mathbf{S}_{E.2b}$ (*decrypt-from-pad*) used in Zombie's precomputation.

proof $\pi_{E.2a}$ for this subcomputation and sends it, along with the pad hash $h_{\mathsf{pad}_{\mathsf{SN}}}$, to the middlebox. The middlebox verifies this proof and stores the hash with the corresponding sequence number.

In the second part of the protocol, run once the Zombie client receives the plaintext M, the client encrypts M with $\mathsf{pad}_{\mathsf{SN}}$ for sequence number SN to get the ciphertext C. Then, it generates the proof for the statement $\mathbf{S}_{E.2b}$ (see *decrypt-from-pad* in Figure 3). This statement takes a pad pad as the witness, which is purportedly $\mathsf{pad}_{\mathsf{SN}}$; ensures that this witness hashes to the stored hash $h_{\mathsf{pad}_{\mathsf{SN}}}$; and passes the decrypted $M = \mathsf{pad} \oplus C$ to the $\mathbf{S}_F$ subcomputation.

Roughly, the security of Zombie's precomputation follows from the zero-knowledge and soundness properties of the proof protocol, and the hiding and binding properties of the hash function H. Specifically, neither the hash $h_{\mathsf{pad}_{\mathsf{SN}}}$ nor $\pi_{E.2a}$ reveal $\mathsf{pad}_{\mathsf{SN}}$—thus, the middlebox cannot learn anything about the client's traffic. The collision-resistance of H and the soundness of the proof system prevent the client from lying about $\mathsf{pad}_{\mathsf{SN}}$, or the key used to derive it—thus, the client cannot equivocate about the sent message M.

3.2 Optimistic approval via asynchronous verification

While precomputation can greatly reduce the per-packet delay incurred by proof generation (down to under 300 ms, §6.1), some applications (for example, web browsing) require even less delay. This section describes how Zombie can perform the ZKP-related parts of its protocol asynchronously, that is, without blocking the flow of normal traffic. Client traffic passes optimistically while the middlebox detects policy violations retroactively.

In more detail, after the client encrypts its packet, it immediately sends the ciphertext C. The middlebox forwards this packet to the server and sets a timer. Then the client generates π and sends it to the middlebox. If the timer has not expired and π is valid, the middlebox does nothing. If the proof is invalid, or the timer fires, the middlebox takes some action, for example, blocking the client (§8).

Asynchronous verification requires the middlebox to keep track of unverified ciphertexts until the client sends the corresponding proofs. Done naively, this could require high memory usage at the middlebox. This memory usage can be reduced by storing a hash of the ciphertexts instead of the ciphertexts themselves. The client would re-send the ciphertext along with the proof, and the middlebox would use the hash to validate the ciphertext. This increases bandwidth usage, since a single ciphertext must traverse the client-middlebox link twice. Since bandwidth usage was not the bottleneck in any of our experiments, we did not implement this optimization.

Security. The middlebox cannot prevent non-compliant packets from leaving the network, nor can it prevent clients from receiving responses to such packets. Zombie will only eventually learn if this has occurred. We claim, though, that this relaxed security is sufficient for many applications. For DNS filtering, the policy goal is to prevent users from browsing blocked sites. Even if the user learns the IP address of a blocked site by sending a non-compliant DNS query, as long as the middlebox can detect this reasonably quickly, further browsing can be blocked. As another example, if Zombie is used to stop users from uploading sensitive data to external sites, it may be sufficient to detect and shut down uploads in time to prevent too much sensitive data from being uploaded, even if (say) the beginning of a file is successfully uploaded. Other context-specific policies may be appropriate, for example a middlebox might optimistically send packets to servers but hold the response packets pending proof verification.

3.3 Batching in Zombie

The final protocol improvement Zombie makes is batch proof generation and verification. Concretely, given ciphertexts $C_1, \ldots, C_b$, Zombie can generate one proof π that verifies only if all b underlying plaintexts are policy-compliant. This single proof is much more efficient for the middlebox to verify than b separate ones. The batch size does not need to be fixed by the middlebox; it can be dynamically chosen by clients based on their current workloads. Figure 2 depicts the conditions under which batching is useful. The combination of batching and asynchronous verification is potent: clients can gather larger batches because they can wait longer to send proofs. Batching is also complementary to precomputation: the client can batch together multiple $\pi_{E.2a}$ proofs.

At a high level, batching works by modifying Zombie's underlying ZKP protocol, Spartan [93], to allow for parallel runs of proof generation to share randomness. Sharing randomness to batch proofs is a known technique, and has been used in other modern proof systems [95, 113]; however, its application to Spartan in this work is novel. We use "SpartanBatch" to refer to our variant of Spartan that supports batching. Appendix A gives details and security analysis.

4 Regular expressions in Zombie

This section describes how Zombie supports middlebox functionality based on regular expressions, which we sometimes call *regexps*. Regexps feature in real-world policies for data

loss prevention (DLP) [71], intrusion detection (IDS) [22, 39], and traffic classification [119, 120]. For example, a DLP system might use a regexp to specify that all outgoing packets containing a social security number should be blocked.

The high-level picture is as follows. Zombie begins with a policy P that uses regexps. This policy (§3) is a restriction on the plaintext payloads (which this section calls simply *payloads*) allowed to pass through the middlebox, and is expressed as a computation $\mathbf{S}_P$ that takes the payload as input and returns 1 or 0 depending on whether the payload adheres to the policy. The policy can be as simple as whether any substring of the payload matches the given regular expression. Or it could include more sophisticated combinations, for example, whether two regexps match within close proximity, or whether there are more than four matches to a given regexp.

Zombie produces both a constraint representation of the computation $\mathbf{S}_P$ and a prover recipe for executing this computation and satisfying those constraints. The constraints $\mathcal{C}_P$ are constructed to be satisfiable if and only if the prover correctly reports whether the payload adheres to P.

As we have described (§1–§2), constraints are an inefficient way to represent general-purpose computations. The same holds for regular expressions: one cannot simply take $\mathbf{S}_P$ to be a regexp library parameterized by a specific regexp, because that would involve compiling, say, C code that uses program constructs that are prohibitive when expressed in constraints.

For this reason, we depart from prior work on regular expressions [119, 120], which aims to make *matching* fast. In our context, matching is the step where the prover executes its recipe to identify a satisfying assignment, and this step is swamped by the costs of proving. Instead, we focus on the driver of those costs: number of constraints (§2). Specifically, our metric of interest is constraints per character in the payload, which we want to be small.

The rest of this section describes how Zombie lowers this metric versus a naive approach. Zombie introduces a series of techniques that achieve substantial improvements in both constants and asymptotics (§6).

4.1 Setup and framework

A given policy P comprises one or more regexps, Boolean combinations of them, and proximity checks. So $\mathbf{S}_P$ has one or more subcomputations, which we denote $\mathbf{S}_R$, referring to a specified regular expression R.

The input to one such $\mathbf{S}_R$ is the payload T (of length L_T); typically, L_T is in the thousands (the number of bytes in a plaintext network packet). The output of a given $\mathbf{S}_R$ is an *array* of L_T Boolean variables; slot ℓ is True if there is a match to R ending at position ℓ and False otherwise; notice that $\mathbf{S}_R$ thus captures not only whether the given R matches any substring(s) of T but also the (ending) position of the match(es).

$\mathbf{S}_P$ processes the output array produced by $\mathbf{S}_R$, or multiple such arrays if there are multiple regexps. Section 4.7 describes that process in detail; until then, we focus on a given $\mathbf{S}_R$.

Zombie encodes $\mathbf{S}_R$ in constraints via several translation phases: $R \to FA \to IR \to \mathcal{C}_R$, where $\mathcal{C}_R$ is the constraint representation of $\mathbf{S}_R$, *FA* is a finite automaton, and *IR* is an intermediate representation that has Boolean logic (`AND`, `OR`, `NOT`), augmented with equality and inequality tests (`==`, `!=`, `<=`, etc.).

Sections 4.2–4.6 describe the main ideas in this translation: a new string matching primitive (§4.2), Zombie's translation from NFAs to constraints (§4.3), a new arithmetization of Boolean logic to substantially lower the cost of encoding Boolean `OR` (at the expense of Boolean `NOT`) (§4.4), techniques for rewriting the regular expression to admit a more efficient translation (§4.5), a new FA formalism that memoizes the results of character class matching (§4.6), and finally exploiting structure in character classes (§4.6).

4.2 Efficient string matching in constraints

Suppose R represents a fixed string, say `a{k}` (`a` repeated k times), so $\mathbf{S}_R$ must determine for each $\ell \in \{0, \ldots, L_T - 1\}$ whether the pattern appears in the payload, ending at position ℓ. If so, a Boolean $b^{(\ell)}$ is 1 and otherwise 0. For illustration, we skip *FA*, so the translations are $R \to IR \to \mathcal{C}_R$. The *IR* is:

$$b^{(\ell)} := (T[\ell]\texttt{==a}) \wedge (T[\ell-1]\texttt{==a}) \wedge \cdots \wedge (T[\ell-k+1]\texttt{==a}).$$

To encode this in R1CS constraints (§2), one expresses $\wedge$ using field multiplication and `==` using `EQUALS-ZERO` (§2, see also [96, Appx D]):

$$\begin{aligned} b^{(\ell)}_{k-1} &:= \texttt{EQUALS-ZERO}(T[\ell-k+1]-\texttt{a}) \\ b^{(\ell)}_{k-2} &:= b^{(\ell)}_{k-1} \cdot \texttt{EQUALS-ZERO}(T[\ell-k+2]-\texttt{a}) \\ &\cdots \\ b^{(\ell)}_{1} &:= b^{(\ell)}_{2} \cdot \texttt{EQUALS-ZERO}(T[\ell-1]-\texttt{a}) \\ b^{(\ell)} &:= b^{(\ell)}_{1} \cdot \texttt{EQUALS-ZERO}(T[\ell]-\texttt{a}) \end{aligned} \tag{1}$$

Notice that $b^{(\ell)}$ equals 1 iff there is a match, and 0 otherwise.

Of course, expression (1) is not literal constraints. To produce those, one expands lines of the form $b^{(\ell)}_i = b^{(\ell)}_{i+1} \cdot \texttt{EQUALS-ZERO}(T[\ell-i]-\texttt{a})$, as follows:

$$\left\{ \begin{array}{l} b^{(\ell)}_i = b^{(\ell)}_{i+1} \cdot M_i, \\ M_i \cdot (T[\ell-i]-\texttt{a}) = 0, \\ Z_i \cdot (T[\ell-i]-\texttt{a}) = 1 - M_i \end{array} \right\}$$

The variable M_i represents the outcome of $\texttt{EQUALS-ZERO}(T[\ell-i]-\texttt{a})$, and Z_i is non-deterministically supplied. Altogether, $\mathbf{S}_R$ for this pattern requires roughly $3 \cdot k$ constraints per character position, so $3 \cdot k \cdot L_T$ in all.

As a more efficient alternative, Zombie introduces a primitive: `STRING-MATCH`. `STRING-MATCH` exploits the observation that, in constraints, the indivisible unit (akin to a bit on a

CPU) is a finite field element, which holds many bits, and thus conceptually "has room" for packing the information about whether many characters matched. Letting Λ be the alphabet, $|\Lambda|$ be its size (256 for ASCII), and S_1, S_2 be two strings:

$$\texttt{STRING-MATCH}(S_1[0]\ldots S_1[k-1], S_2[0]\ldots S_2[k-1]) \triangleq \texttt{EQUALS-ZERO}\left(\sum_{i=0}^{k-1} |\Lambda|^i \cdot (S_1[i] - S_2[i])\right).$$

Zombie replaces expression (1) with $\texttt{STRING-MATCH}(T[\ell-k+1]\ldots T[\ell], \texttt{a....a})$, which (assuming loose limits on k; see below) is 2 constraints per input character, down from $3 \cdot k$. To see why, note that the argument to EQUALS-ZERO is a weighted sum of the variables $T[i]$ plus a constant term, with the weights and constant term known at compile time. Plugging that argument into EQUALS-ZERO keeps the constraints in R1CS format (§2).

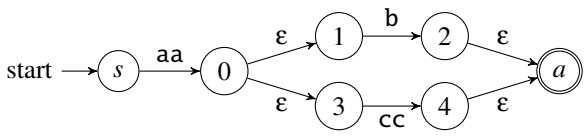
The loose limits are determined by the size of the alphabet and the size of the field that the constraints are expressed over. Assuming a field of size q, the maximum length of a pattern that can be compiled into a single STRING-MATCH is $\lfloor \log_{|\Lambda|}(q) \rfloor$. For our application, we consider the alphabet of ASCII characters ($|\Lambda| = 2^8$) and a 255-bit prime field (the base field of curve25519 [11]); thus, patterns of at most 31 characters can be compiled into a single STRING-MATCH.

If these loose limits do not hold, the pattern compiles into several STRING-MATCHs, connected by AND ($\wedge$).

4.3 From regular expressions to constraints

Real-world systems [25, 58, 67] translate regular expressions to executable code in two steps. First, they produce a nondeterministic finite automaton (NFA), via Thompson's algorithm [104]. Second, they determinize the NFA to get a DFA [101, Ch. 1]. This step represents the DFA's state transition function as a table: an entry for every state and every character. This representation makes execution very fast. However, in our context, the entire exponentially-sized table would turn into constraints, exploding proving costs.

Thus, Zombie stops after the Thompson step. Because of its packing technique, Zombie produces FAs that have *string* transitions instead of the usual character transitions. As an example, consider the regular expression: aa(b|cc). Here is the NFA (ε refers to the empty string; s and a are the start and accepting states):



Zombie's *IR* representation of this FA uses functions, one for the final state and each intermediate state that has non-epsilon incoming transitions. Each function encodes, for each character position ℓ, whether the FA could be in the given state at character position ℓ.

$$\begin{aligned} f_0(\ell) &:= \texttt{STRING-MATCH}(T[\ell-1]T[\ell], \texttt{aa}) \\ f_2(\ell) &:= f_0(\ell-1) \wedge (T[\ell]\texttt{==b}) \\ f_4(\ell) &:= f_0(\ell-2) \wedge \texttt{STRING-MATCH}(T[\ell-1]T[\ell], \texttt{cc}) \\ f_a(\ell) &:= f_2(\ell) \vee f_4(\ell) \end{aligned} \quad (2)$$

Translating a function $f(\cdot)$ to constraints means that each evaluation $f(0), \ldots, f(L_T - 1)$ is separately translated and possibly assigned to a constraint variable. For example, $f_2(\ell) \vee f_4(\ell)$ translates to $f_2[\ell] + f_4[\ell] - f_2[\ell] \cdot f_4[\ell]$, where $f_2[\ell]$ is a constraint variable that represents $f_2(\ell)$. Notice that the translation of $\vee$ requires a constraint, because of the multiplication. Also, each AND ($\wedge$) translates to a constraint that multiplies ($\cdot$) its terms. So, expression (2) is 9 constraints for each position ℓ (2 for ==, 2 for each of two STRING-MATCH, and 1 for each of the three multiplications). Notice from the definition of STRING-MATCH earlier that the cost is relatively insensitive to the length of the substrings. For example, if the pattern were a{k}(b{k}|c{k}) (a k-length run of a followed by a k-length run of b or c), then the number of constraints is unchanged (assuming the aforementioned loose limits on k).

4.4 A new arithmetization of Boolean logic

Traditionally, when *arithmetized*—that is, translated to constraints—Boolean logic maps True to 1 and False to 0. Letting p, q, r be Boolean variables [8, 85, 94–97]:

$r := p \wedge q$	customarily translates to:	$r = p \cdot q$
$r := p \vee q$	customarily translates to:	$r = p + q - p \cdot q$
$q := \neg p$	customarily translates to:	$q = 1 - p$

Above, multiplication ($\cdot$) and addition ($+, -$) are over the underlying finite field $\mathbb{F}$ (§2).

Zombie introduces an alternate arithmetization: False still maps to 0 but any non-zero value in the underlying finite field functions as True:

$r := p \wedge q$	now translates to:	$r = p \cdot q$, as above
$r := p \vee q$	now translates to:	$r = p + q$ (assuming no overflow; see below)
$q := \neg p$	now translates to:	$q = \texttt{EQUALS-ZERO}(p)$

For example, in (2), $f_a(\ell)$ translates to $f_2[\ell] + f_4[\ell]$, shedding the term $f_2[\ell] \cdot f_4[\ell]$. This concretely goes from 9 to 8 constraints.The source of the savings is that $f_a(\ell)$ no longer needs a constraint itself: any other constraint that uses $f_a(\ell)$ can substitute in the sum $f_2[\ell] + f_4[\ell]$. Notice that any such substitution retains R1CS format (§2), whether the substitution happens in the "A"-part of the constraint, the "B"-part, the "C"-part, or combinations thereof. That is, $f_2[\ell]$ and $f_4[\ell]$

are components of the z vector from Section 2, and their inclusion in a constraint simply adds 1 to the corresponding coefficients. More generally, arithmetizations that are linear combinations (that is, no degree-2 terms, meaning no multiplications of two or more variables) cost no constraints. We will use this fact over and over again.

Consequently, `OR` has become mostly free: addition of degree-1 terms, being a linear combination, doesn't require constraints. We say *mostly* because, for this to work, $p+q$ must not overflow, that is, wrap around the finite field modulus and become 0 when at least one of the summands is non-zero. Our implementation of Zombie (§5) handles this issue at compile time. The compiler tracks the maximum possible value of variables and, if overflow is possible, inserts constraints to reduce a summand to a 0-1 term before it has the chance to overflow. The specific constraints are `NOT-EQUALS-ZERO` [96, Appx D], which maps 0 to 0 and non-zero values to 1.

By contrast, `NOT` ($\neg$) has gone from free (because it was a linear combination) to requiring two constraints, for `EQUALS-ZERO` (see §2). Finally, `AND` ($\wedge$) costs one constraint in both arithmetizations. The overall trade, then, is to make `NOT`s more expensive in exchange for free `OR`s.

This trade not only is a dramatic improvement but also carries broader significance. In our context, the 9-to-8 savings in the earlier example is a restricted case; in fact, this arithmetization has a quadratic-to-linear improvement. To see why, consider a state that has $s-1$ inbound paths, one for each of the other states in an s-state FA. For example: $f_a(\ell) = f_1(\ell) \vee f_2(\ell) \vee \cdots \vee f_{s-1}(\ell)$. In the traditional arithmetization, each disjunct requires a constraint with a field multiplication, each of which costs one constraint; the total for $f_a(\ell)$ in this example is s constraints. In the worst case, then, $O(s)$ states can each require $O(s)$ constraints, for a total of $O(s^2)$ constraints for each $\ell \in \{1, \ldots L_T\}$. In Zombie, by contrast, f_a would be translated into $f_1[\ell] + f_2[\ell] + \cdots + f_{s-1}[\ell]$. This costs 0 constraints because it is a linear combination.

Qualitatively, this arithmetization means that Zombie gains enormously from devising *IR* representations that use mainly `OR`, with `AND` entering only when necessary. Beyond Zombie, this point applies to the constraint translation of any problem naturally expressed with many conjunctions and disjunctions, such as 3-SAT.

4.5 Preprocessing regular expressions

Another technique in Zombie is rewriting regular expressions at compile time to favor longer substring matches. Doing so exploits packing (§4.2) to reduce the number of `AND`s and the number of states in the IR. For example, Zombie rewrites `aa(b|cc)` as `(aab|aacc)`, yielding the following *IR*, which should be compared to (2):

$$f_0(\ell) := \texttt{STRING-MATCH}(T[\ell-2]T[\ell-1]T[\ell], \texttt{aab})$$
$$f_1(\ell) := \texttt{STRING-MATCH}(T[\ell-3]T[\ell-2]T[\ell-1]T[\ell], \texttt{aacc})$$
$$f_a(\ell) := f_0(\ell) \vee f_1(\ell) \tag{3}$$

Whereas we saw in the previous section that the formulation in (2) costs 8 constraints for each character position ℓ, the one in (3) costs 4 constraints (two for each `STRING-MATCH`).

4.6 Character classes and a new FA formalism

A common and convenient feature of regular expressions is *character classes*, for example, `[0-9]` or `[A-Za-z]`, which respectively match any digit and any ASCII alphabet character. Naively treating a character class as a union (using the `|` operator) would be expensive. Although real-world regexp frameworks have special optimizations for character classes, these would not contribute to efficient constraint representations, for the reasons discussed at the beginning of this section. Instead, Zombie applies several of its own optimizations.

First, Zombie deduplicates so that the costs associated with matching to a class are paid once, even if there are multiple instances of the class in the regular expression. To do so, Zombie constructs a new kind of FA, one that uses "sub-FAs" to *write* to separate tapes (FAs are not typically modeled as writing to a tape) and then reads the tapes in the "main" FA. The sub-FAs are each supposed to produce an array of Booleans. As an example, consider the regexp `[0-9]a[0-9]`. Zombie produces the following *IR*:

$$t_0[\ell] := \texttt{MATCH-CLASS}(T[\ell], \texttt{[0-9]})$$
$$f_a(\ell) := \texttt{STRING-MATCH}(t_0[\ell-2]T[\ell-1]t_0[\ell], 1_{\mathbb{F}}\texttt{a}1_{\mathbb{F}})$$

$1_{\mathbb{F}}$ is 1 in the finite field and is used to encode the Boolean result of `MATCH-CLASS`. Think of t_0 as memoizing the sites of matches found by a sub-FA; notice how the values in t_0 are reused in $f_a(\cdot)$.

Outside of the present context, the requirement for an additional tape would seemingly require more memory for the prover. In our context (constraints), each extra tape *saves* memory, by reducing the number of variables necessary to represent a match to the character class.

Besides deduplication, another benefit of Zombie's FA formalism is that it enables longer substring matches. The idea is similar to the example in Section 4.5. Here, the packing technique (§4.2), this time applied to the results of other tapes, lets f_a consist of a single `STRING-MATCH`. Conversely, can we use deduplication on that earlier example? No, because the union components were different lengths.

As another optimization, Zombie exploits structure in the character class. For example, Zombie encodes `[A-Za-z]` with only 25 constraints (fewer than the 52 characters in the class!):

$$\texttt{MATCH-CLASS}(T[\ell], \texttt{[A-Za-z]}) = (T[\ell] \texttt{ >= A}) \wedge (T[\ell] \texttt{ <= z}) \wedge \cdots$$

The elided terms check that $T[\ell]$ is not one of the few ASCII characters between `Z` and `a`. This approach relies on the IR primitives <= and >=, which translate to $\log_2|\Lambda|$ + 1 constraints [96], which is 9 if Λ is the 8-bit ASCII characters.

Zombie's compiler tries to optimize the encoding of a class; for example, treating the class `[0-9]` as a range with <= and >= operators is not worthwhile. Larger classes see greater benefit from being treated as a range.

4.7 Applying regexp-based policies in ZK

In this section, we move from considering a single $\mathbf{S}_R$ to a higher-level policy P, expressed as $\mathbf{S}_P$.

One-shot expressions. Consider a basic case: P is simply whether there is a match to some R somewhere in the payload. Recall that $\mathbf{S}_R$ is already encoded as constraints for $\{f_a(\ell)\}_{\ell=0,\ldots,L_T-1}$. $\mathbf{S}_P$, then, is `NOT-EQUALS-ZERO`$(f_a(0) \vee f_a(1) \vee \cdots \vee f_a(L_T-1))$. Assuming no overflow (so the `OR`s are free; §4.4), the overhead of $\mathbf{S}_P$ beyond $\mathbf{S}_R$ is two constraints, stemming from `NOT-EQUALS-ZERO` (§4.4). Zombie's compiler handles possible overflow as described earlier (§4.4).

Proximity. In network security, simple regexp searches can have too many false positives. Thus, the policy P is sometimes concerned with context: individually two patterns are not sensitive, but close together they are. For example, a DLP policy might disallow a pattern matching a driver's license number within 100 characters of strings like "driving license," "driver's license," "DL," etc. (§6.3).

Perhaps surprisingly, Zombie can handle such policies with very little overhead beyond the cost of matching the individual regexps. Consider a computation $\mathbf{S}_P$ that returns 1 if there are respective matches to two regular expressions R_1 and R_2 within d characters of each other. Notice that, for correctness, all possible combinations of occurrences of the two patterns have to result in $\mathbf{S}_P$ returning 1. To capture these possibilities, $\mathbf{S}_P$ performs two steps. First, it takes the f_a array of R_1, call it f_{r_1}, and produces a new array $f^d_{r_1}$, which for each position ℓ holds a Boolean indicating whether there is a match within d characters of ℓ. Concretely,

$$f^d_{r_1}[\ell] = \sum_{k=1-d}^{d-1} f_{r_1}[\ell+k].$$

Because each $f^d_{r_1}[\ell]$ is a linear combination of existing variables, there is no cost in constraints to produce it (§4.4). Second, $\mathbf{S}_P$ checks whether the entrywise product of $f^d_{r_1}$ and the f_a array of R_2, call it f_{r_2}, has any non-zero entries. This check requires L_T+2 constraints: one for each product, and two for a `NOT-EQUALS-ZERO` applied to the sum of these products.

Thus, in total, the requirement for proximity costs an amortized 1 constraint per character in the payload. In contrast, naively encoding proximity as a single regexp, $(R_1(.\{0,\mathtt{d}\})R_2)\,|\,(R_2(.\{0,\mathtt{d}\})R_1)$, would introduce an extra $O(L_T \cdot \log d)$ constraints.

5 Implementation

Our implementation of Zombie has two main components: a client and a middlebox. We currently support two classes of applications. The first is DNS filtering [49] (see also §2), as applied to the DNS-over-TLS and DNS-over-HTTPS protocols. The second is arbitrary policies involving regular expressions, for example DLP policies for files sent by clients via HTTPS, applied to text files (as opposed to formats such as PDF).

5.1 ZKP implementation

Circuits. The circuits used for Zombie's ZKPs are specified in the ZoKrates domain-specific language (DSL) [34] and compiled to R1CS using CirC [84], a ZKP compiler framework. The circuits comprise 1832 handwritten lines of ZoKrates code and 630 lines automatically generated by our own regexp compiler (which is a standalone component that could be integrated with other projects). The handwritten code was optimized and features a large improvement in the encoding of $\mathbf{S}_F$ (§3).

The regexp compiler takes as input (a) a list of regexps and (b) a list of proximity restrictions between pairs of regexps. Using built-in knowledge of the constraint-level costs of ZoKrates' semantics, the compiler applies the techniques in Section 4. The compiler is 5425 lines of C++, 463 lines of `yacc`, and 50 lines of `lex` code on top of the BNFC library [1].

ZKP improvements. In implementing Zombie, we made several improvements to CirC and the existing Spartan implementation. First, we created an adapter that integrates CirC with Spartan. We have also configured Spartan to use curve25519 [11] as its underlying cryptographic group, a standard choice believed to offer ≈128 bits of security.

Our CirC improvements make witness generation more efficient; in early experiments, witness generation was slower than proof generation. We modified internal CirC data structures to prevent unnecessary memory copying, which greatly improves performance.

We improved the Spartan prover and verifier to take full advantage of parallelism, resulting in better performance for generating multiple proofs even in the non-batch setting.

5.2 Client implementation

The client implementation comprises 1976 lines of Python. When performing DNS filtering, the Zombie client acts as a local DNS proxy. It accepts UDP DNS requests then sends them to a recursive DNS resolver (we use Google's 8.8.8.8 resolver [46]) over TLS (DoT [53]) or HTTPS (DoH [52]). The web browser is configured to point to the local proxy for DNS resolution. The client performs the channel-opening (§3) with the Zombie middlebox on startup to set up a session. It uses this session for as long as the recursive resolver will allow (up to five minutes in our testing). It generates and sends proofs, and forwards traffic to the middlebox; we ensure this via routing tables.

Precomputation. In our implementation, the client has a child process for precomputation (§3.1) that has lower priority than the main proxy process, constantly generating *pad-*

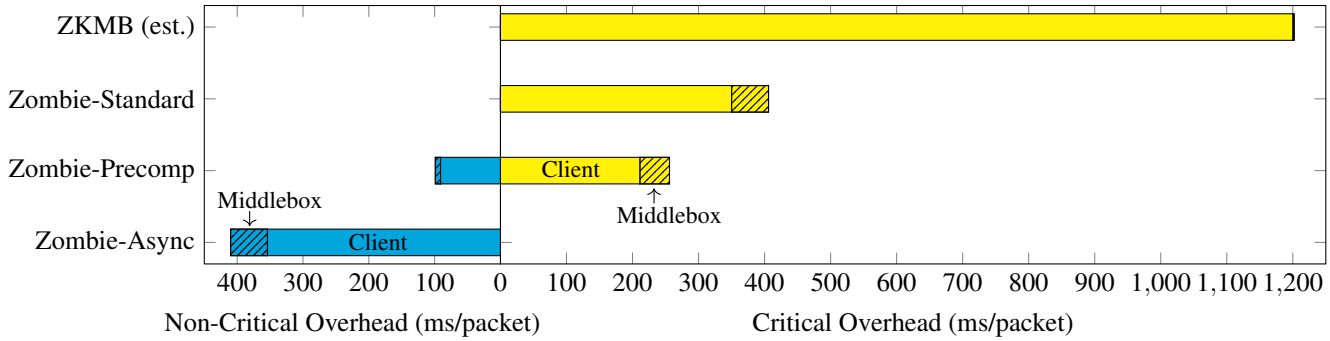


Figure 4: Per-Packet critical and non-critical overheads (relative to a no-policy baseline) for enforcing a blocklist on DNS requests over TLS in zero-knowledge with ZKMB [49] and with Zombie-Standard, Zombie-Precomputation, and Zombie-Asynchronous configurations.

commit proofs when the proxy is idle. When generating these proofs ($\pi_{E.2a}$) (§3.1), the client has two parameters to balance. First is the length of the pad, L_{pad}, which, for each proof, must match or exceed the size of each packet to be encrypted. Choosing a larger L_{pad} will result in more proving work, but will allow the client to send larger packets. The second parameter, m, is the number of *pad-commit* proofs to batch to amortize prover and verifier work. A higher m is less likely to be exhausted by a burst of traffic, but it risks performing excess precomputation that might not be used. In our implementation, for DNS-over-TLS we set $L_{pad} = 255$ (the size of DNS request payloads) and $m = 16$.

5.3 Middlebox implementation

The middlebox is implemented in 1595 lines of Rust. The middlebox configures IP packet filter rules using `iptables`. When packets arrive at the middlebox, they are put on a queue implemented via `libnetfilter_queue` in Linux. The middlebox dequeues packets and performs the following steps. First, it determines whether they are policy-relevant. If so, the middlebox increments the TLS sequence number; it needs to have an accurate count of the sequence number to verify proofs. Then, it buffers the received packet for verification. When the middlebox receives the proof from the client, it links the proof to the packet by sequence number, verifies the proof, and forwards it.

6 Evaluation

We evaluate Zombie with these questions:

(1) What are the overheads added by different configurations of Zombie?

(2) How does batching improve middlebox throughput?

(3) What are the costs of different components of Zombie?

(4) How effective are Zombie's regexp techniques?

Method, applications, and baselines. Our experiments measure client, server, and overall end-to-end delay introduced by Zombie, and we compare these overheads against those introduced by the original ZKMB work [49]. We evaluate Zombie for DNS filtering and DLP policies applied to traffic over TLS 1.3. The DNS filtering benchmarks use a representative adult-content domain blocklist from prior work [49, §8.2] [69] (with 2 million domains). The DLP benchmarks use policies from Microsoft DLP Purview [71].

Our experiments that require networking run on CloudLab [33] while those that do not are run on Amazon Web Services (AWS). On CloudLab, we use `c6525-25g` instances. Each has a 16-core 3GHz AMD 7302P CPU, with 128GB RAM, SSDs, and two Mellanox 25Gb/s NICs. On AWS, we use an instance with a 16-core 2.90GHz Intel 8375C CPU with 128GB RAM and SSDs. This instance has similar computation resources to that of CloudLab `c6525-25g`.

6.1 Computational overhead and delay, no batching

We measure the overhead introduced by Zombie with respect to a no-policy setup, in which the middlebox forwards the traffic without enforcing any policy. We also compare this overhead to ZKMB [49], via conservative estimates of ZKMB's performance extrapolated from microbenchmarks on the same hardware; this is labeled "ZKMB (est.)".

We run three configurations of Zombie, with the following settings of precomputation and synchrony:

(1) *Zombie-Standard*: No precomputation, no asynchrony.

(2) *Zombie-Precomputation*: Precomputation, no asynchrony.

(3) *Zombie-Asynchronous*: No precomputation, asynchrony.

We do not evaluate precomputation combined with asynchrony here; asynchrony moves all overheads to the non-critical path, so precomputation does not affect latency in this case. Batching can be applied to all of the above; we evaluate it separately (§6.2). We run this experiment on CloudLab and report the average of 20 255-byte DNS requests.

Figure 4 depicts the results. Zombie-Standard incurs approximately 3× lower latency than ZKMB. A major difference in client and middlebox work comes from the transition

from the Groth16 [48] proof system to Spartan [93] (§2). We see additional gains from the enhancements detailed in Section 5. The average additional latency is about 400 ms for Zombie-Standard: approximately 350 ms for proof generation and 50 ms for verification.

With precomputation, the average latency is lower, about 250 ms. While much worse than the average latency for a DNS request, which is about 20 ms [86], it may still be tolerable (§8). With optimistic approval via asynchronous verification, Zombie introduces no additional latency. Of course, asynchronous mode requires (in addition to assumptions about policy enforcement; §3.2) additional storage requirements for the middlebox, for buffering packets. We return to these storage requirements when evaluating batching, as the effects on storage are more pronounced there.

Observe that Zombie-Standard and Zombie-Asynchronous introduce the same total per-packet overhead. This is expected: they involve proving and verifying the exact same statements. Surprisingly, Zombie-Precomputation requires slightly less overall per-packet overhead despite the need to produce and verify more proofs than Zombie-Standard. This decreased overhead is because the work to generate multiple *pad-commit* proofs is parallelized in our implementation.

Turning to the communication overhead of sending proofs: in the synchronous setting with precomputation, each online proof is approximately 30 KB. As we will argue (§8), the overall increase in required bandwidth is expected to be small.

6.2 Effect of batching

We investigate the effect of batching (§3.3) on the throughput (number of (proof, packet) pairs processed per second) and the storage requirements of the middlebox. We do not separately evaluate batch proof size; a batch of proofs is the same size as the sum of the non-batched proofs.

We run the DNS benchmark in a new Zombie-Async-Batch configuration. We model each client as a Poisson process, whose parameter is scaled by the batch size. For example, if the batch size is 8, then we set the average interarrival time to be 8× longer than when the batch size is 1, and when an arrival event happens, the client sends 8 packets. For each batch size, as offered load increases, throughput does not collapse, but instead it approaches a maximum. We interpret this maximum throughput as the middlebox's empirical capacity for that particular batch size.

We measure this quantity under four batch sizes; for each size, we average three experiments. We also create a model to predict maximum throughput: we measure the time to verify a single Zombie DNS proof on one thread on CloudLab, and individually measure the execution time of the fixed cost code block and the marginal cost code block, obtaining 121 ms per batch and 38 ms per proof, respectively.

Figure 5 compares the empirical measurements and the model; the divergence is around 5%. This discrepancy owes to lower-order middlebox costs related to packet forwarding,

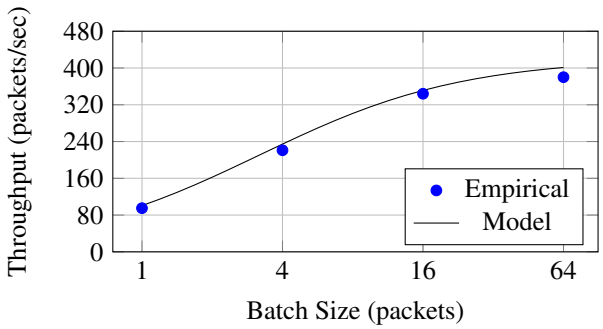


Figure 5: Middlebox throughput vs batch size.

listening, and proof parsing. The maximum throughput we observe is 380 packets per second, at a batch size of 64. Although larger batches would further increase throughput, the possible improvement levels off (even in theory); besides, larger batches are impractical.

Batching brings additional storage requirements (to accumulate ciphertexts). However, storage capacity is not a limiting factor here, owing to the small size of DNS requests and the throughput of 380 packets per second. Even if we assume that the middlebox is exactly at capacity, never falls behind, always has a proof to check (even if it might be waiting on other proofs), and has a generous window of 60 seconds for proofs to arrive after the first corresponding ciphertext in the batch, then the middlebox would still need to store less than 6 MB of ciphertexts at any given time. This is well within the capacity of even the smallest of middleboxes [83].

6.3 Circuit benchmarks

To pinpoint the costs of individual components, we run the prover and verifier in isolation on AWS, taking the average of 5 executions, for various circuits. Figure 6 shows the results.

DNS-related circuits. The first five rows are related to the DNS blocklist policy. The first and second rows serve to compare Zombie and ZKMB, using the same circuit as the one for the Zombie-Standard and Zombie-Asynchronous benchmarks in Figure 4. We see a significant reduction in the number of constraints in Zombie, which owes to the optimizations mentioned in Section 5; thus, the performance gain in Figure 4 for Zombie results not only from the Spartan back-end but also our optimizations. The total number of constraints for the third row is less than one-third of the number of constraints in the second row, indicating that the dominant cost in our implementation of Zombie-Standard is $\mathbf{S}_{E.2}$. The sum of the constraints in the fourth and fifth rows (measuring offline and online cost with precomputation) is slightly larger than the number of constraints in the second row; this is expected because there is extra work to commit to $\mathsf{pad}_{\mathsf{SN}}$ (Figure 3).

Regexp circuits. The last four rows of Figure 6 include costs for a DLP benchmark. This benchmark combines five Microsoft Purview policies for detecting sensitive information in the US locale: bank account number [72], driver's license

Benchmark: Circuit description and size								
$\mathbf{S}_E$	$\mathbf{S}_F$	$\mathbf{S}_P$	Payload Size	Constraints	Prover Time	Verifier Time	Memory	Proof Size
E.2	DNS	Blocklist	255 B	*176000*	*1200 ms*	*2 ms*	-	*128 B*
E.2	DNS	Blocklist	255 B	128702	365.4 ms	39.8 ms	531.8 MB	30.2 KB
	DNS	Blocklist	255 B	40295	191.2 ms	29.8 ms	212.3 MB	21.3 KB
E.2a			255 B	86568	278.0 ms	36.4 ms	377.2 MB	30.2 KB
E.2b	DNS	Blocklist	255 B	48562	221.4 ms	31.2 ms	258.5 MB	21.3 KB
E.2	HTTP	Microsoft DLP	100 B	64438	207.0 ms	29.0 ms	225.7 MB	21.3 KB
E.2	HTTP	Microsoft DLP	2000 B	1186241	6363.8 ms	319.6 ms	10114.6 MB	49.1 KB
	HTTP	Microsoft DLP	100 B	20080	112.2 ms	23.2 ms	89.2 MB	20.5 KB
	HTTP	Microsoft DLP	2000 B	490966	4909.0 ms	245.4 ms	8402.9 MB	48.0 KB

Figure 6: Costs of various circuits in Zombie. The first row shows estimated overheads for the prior work, ZKMB [49], using the Groth16 [48] back-end (§2). Memory is a single column because the prover and verifier have the same memory requirements with the Spartan back-end. Blank cells indicate that a subcomputation of that type is not included in that row's benchmarked circuit.

number [73], taxpayer number (ITIN) [74], social security number [75], and passport number [76]. These policies use substring matches, regular expressions, and proximity checks. A message must pass all five to pass; for brevity, we refer to the overall policy simply as "Microsoft DLP". We encode this policy using Zombie's regexp pipeline (§4–§5) and benchmark it on HTTP POST messages of varying sizes.

Per byte, the cost of this policy is 25–50% more expensive than the DNS blocklist benchmark. This can be seen by comparing the DNS blocklist (no $\mathbf{S}_{E.2}$) and HTTP Microsoft DLP (no $\mathbf{S}_{E.2}$) rows, and dividing the metrics by the payload size. The result is approximately 200–245 constraints per byte for DLP and 160 constraints per byte for the blocklist.

Packet size brings a complication. However, this is not because of latency, which scales linearly with packet size, so overall latency is driven by total bytes (in a policy-relevant flow), rather than the distribution of bytes over packets. The main issue with packet size is in memory consumption, which scales linearly with circuit size, and reaches into the GB in our experiments (Figure 6). The mitigant is that the memory cost is paid once per circuit, and circuits are reusable, creating benefit from small packets. Concretely, using a circuit for a 100-byte packet 20 times is much better for middlebox memory than using a circuit for a 2000-byte packet once.

Regexp circuit techniques. Figure 7 depicts the results of each of our regexp techniques in more detail. We show the decrease in per-byte overheads from incrementally applying the optimizations discussed in Section 4. We use the Microsoft DLP policy for these benchmarks and exclusively consider the $\mathbf{S}_P$ subcomputation.

For this policy, our optimizations reduce the per-byte overhead by almost an order of magnitude, from over 20 ms per byte to under 2 ms per byte. The most substantive improvements come from `STRING-MATCH` (§4.2) and from creating multiple tapes (§4.6).

To contextualize these costs, we compare them to the per-byte overhead of decryption, approximated by fixing a payload size, subtracting the costs for a given row without $\mathbf{S}_{E.2}$ from the corresponding cost with $\mathbf{S}_{E.2}$ (or by looking at the $\mathbf{S}_{E.2a}$ row), and dividing by the payload size. The result is 348 constraints per byte, which should be compared to the 242 constraints per byte at the bottom of Figure 7 (down from 1566, at the top of the figure). That is, after the techniques in Section 4, the dominant cost is no longer regular expression handling but rather representing decryption in the circuit.

Techniques	# Constraints	Prover time	Verifier time
Baseline	1566 / B	*19000 ms*	*1400 ms*
+ `STRING-MATCH`	996 / B	*11000 ms*	*860 ms*
+ Alt Arithmetization	901 / B	*10000 ms*	*760 ms*
+ Regexp Preprocessing	873 / B	*9800 ms*	*740 ms*
+ Additional Tapes	288 / B	*2300 ms*	*160 ms*
+ Optimized Classes	242 / B	1705 ms	38 ms

Italicized entries indicate estimates.

Figure 7: Effect of techniques, in the order they are introduced in Section 4, on the Microsoft DLP policy (1 KB payload). The final line is the result of running CirC on the zok file produced from all optimizations. The times in the other lines are estimates from the compiler with all prior optimizations enabled and all subsequent optimizations still disabled.

7 Related work

Systems built using probabilistic proofs. Probabilistic proofs are a foundational concept in complexity theory with a deep and rich literature [6,7,9,42,43]; for a survey, we recommend Goldreich [41]. The last decade has seen rapidly growing interest from the applied cryptography community, with a particular emphasis on zero-knowledge proofs. For a survey, we recommend Walfish and Blumberg [117] or Thaler [103].

Zombie is part of a growing line of work applying proba-

bilistic proofs to solve practical problems, such as privacy-preserving payments [28, 91] private smart contracts [15, 19, 57], proofs of solvency [4, 27], verifiable delay functions [14, 55], proofs of software vulnerability [26] and cryptographic transparency logs [21, 107, 108]. Of particular relevance to our work are DECO [123] and Reclaim [100], which employ probabilistic proofs about TLS plaintext. However, both systems aim to prove statements about a TLS session (e.g. "My bank account balance is greater than $\$X$") to an out-of-band third party, rather than an in-band middlebox. This makes the proof more challenging, as the verifier needs to be convinced that the claimed ciphertext really came from a session with the claimed server. To solve this, DECO relies on multiparty computation between the client and a third-party notary. However, these applications do not face tight latency constraints, as ZKMBs do, enabling much different performance tradeoffs.

Regular expressions in zero knowledge. Zombie introduces a portfolio of techniques for encoding regular expressions in probabilistic proofs. Previous to Zombie, the approach taken [3] was direct translation of a DFA, with exponential costs. We are aware of only two other works, both concurrent with Zombie, that improve on the exponential baseline; like Zombie, both target network security applications.

Exciting work by Luo et al. [68] transforms a regular expression to a Thompson NFA [104], like Zombie does (§4.3). Unlike Zombie, Luo et al. transform the NFA to a Boolean circuit and then use MPC-in-the-head [29, 54]. In addition to the setting where the client knows the policy, Luo et al. also consider the setting where the middlebox wants to keep the policy private but still apply it to the client's traffic. This part of their application thus has a significantly different performance profile than ours (an extra logarithmic term is introduced in the size of the regexp, and extra overheads are incurred to preserve the privacy of the policy itself.) Additionally, because most of our optimizations rely on constraints over large finite fields while theirs are tailored Boolean circuits, the respective techniques do not seem to be applicable to each other. While a detailed comparison has yet to be done, Zombie appears to have an order of magnitude lower communication cost (proof size) and computation (prover time) in the public policy case.

The other concurrent work, `zkreg` [89], compiles a large collection of regular expressions (mostly string matches) into an Aho-Corasick automaton [2], encodes this automaton as an arithmetic circuit, and then uses a custom Commit-and-Prove scheme [20] to prove membership and non-membership in zero knowledge on extremely large dictionaries of strings. For example, they consider proofs involving an automaton with 19 million states and over 300 million transitions. To handle an automaton this large, they represent it as a multiset of transitions and handle transition checking partially using set membership. This incurs a significantly higher computational overhead than our transition checking, but it scales far better for large automata (for which it is explicitly designed). Future work is to investigate ways of combining relevant techniques in `zkreg` with Zombie to efficiently support larger policies.

Middlebox architectures. Many proposed middlebox architectures aim to enforce policies on encrypted traffic. For helpful surveys, we refer the reader to Sherry [99] and Naylor et al. [78]. Work prior to ZKMB [49] largely falls into two broad categories:

Trusted hardware. ETTM [31] first proposed shifting policy enforcement logic from middleboxes to network users (end hosts) themselves. This requires trusted hardware to assert that a virtual machine run by the end host is faithfully checking that the plaintext is policy-compliant. Endbox [44] refined this vision using the then-emerging *trusted execution environment* (TEE) abstraction, specifically using Intel's SGX implementation. An obvious limitation is that all users must have a TEE to take advantage of this approach.

mbTLS [78] proposes relying on a TEE at the middlebox itself, acting as a middleperson (MITM) between a TLS session established with the client machine and one with the server (which can also be extended to multiple hops). This undermines the typical end-to-end nature of TLS, but if the TEE remains secure users can trust that their plaintext will only be used by the TEE for policy checks. Another approach is to shift policy enforcement from a local middlebox to a TEE run on a cloud server [51, 87, 105]. Other works, too, rely on trusted hardware [32, 45, 50, 59, 118].

We wish to avoid trusted hardware, given the cavalcade of exploits demonstrated against real-world TEE implementations [36, 47, 77, 80, 92, 102, 109–112], using information like power consumption or electromagnetic emanations to extract secrets from an enclave non-invasively. One might note that zero-knowledge prover implementations can also be subject to side-channel attacks [40, 106]. However, there is a subtle but essential difference: TEE-based middleboxes inherently require running code with access to secrets in an enclave *that is placed under the direct control of an adversary*, for example a policy-enforcing enclave hosted by a client machine (where a TEE break undermines integrity) or an enclave with access to decryption keys hosted by the middlebox (where a TEE break undermines privacy). Either setup makes side-channel attacks considerably easier to mount, versus the ZKMB paradigm, in which the relevant adversary (for example, the network administrator) is *remote from the party executing the relevant algorithm* (for example, the ZK prover).

TLS modifications. Several proposals to reconcile widespread TLS adoption with network policy enforcement envision modifying TLS to make it "middlebox-aware," with middleboxes gaining the ability to read and/or modify some (but not necessarily all) of the plaintext data sent in a TLS connection and users typically getting some visibility into the process [10, 12, 64, 65, 78, 79, 121]. An example is "multi-context TLS" or mcTLS [79], with different middleboxes on

the network path receiving context-specific keys based on the permissions the client and server are willing to grant. In the case of DNS filtering, a middlebox might require read-only access to the request body of a DNS query. While this approach enables finer-grained tradeoffs than disabling encryption completely, it is still a blunt instrument that sacrifices user privacy considerably; in the DNS example users fully give up privacy of their query history. It also requires server-side changes.

Blindbox [98] proposed modifying TLS to support policy enforcement by middleboxes. Specifically, Blindbox supplements the standard, semantically-secure symmetric encryption used in TLS with *searchable encryption*. This second ciphertext, along with techniques from circuit garbling and oblivious transfer, allows middleboxes to obliviously execute policy checks on ciphertext, specifically tailored to searching for keywords in text. A rich line of follow-up work extends this basic model [35, 56, 61–64, 66, 81, 82, 88, 122].

All of these works use some variant of *functional encryption*, which allows middleboxes to compute a limited function of the underlying plaintext, with different proposals tailored to different functionality. These works all face the challenging requirement of changing TLS servers, as well as relying on servers to check consistency of the TLS plaintext and that of the supplemental functional encryption (without this check, clients might send policy-violating traffic over TLS but append a functional encryption of benign traffic to satisfy the middlebox). A key goal in our work is not to require changes to, or participation of, existing TLS servers (§1).

8 Discussion

Zombie is 3× cheaper than its progenitor ZKMB [49], with an overhead of 406 ms for enforcing a blocklist on DNS requests over TLS (§6.1). Assuming client idle time, this drops to 256 ms, via precomputation. This number may be tolerable for DNS filtering; by comparison, traditionally satellite Internet connections have added at least 600 ms of latency [13] (though, to be fair, modern low-earth-orbit satellite Internet service offer significantly lower latency, as low as 25 ms [70]). Regardless, under optimistic approval, there is zero online overhead (§6.1). Zombie is thus plausibly practical for clients. Also, our regular expression techniques allow for new policies in the ZKMB paradigm, reducing the cost of enforcing complex policies to the point where the dominant cost is decryption (§6.3).

Although the communication overhead of 30 KB per proof (§6.1) is 120× larger than 255-byte DNS requests, the proof size is small when compared to the average website size of 2–3 MB [5]. Furthermore, these proofs are transmitted only from client to middlebox, which are typically on the same local network. Moreover, Zombie is best geared to settings where most packets do not need proofs (see below), so the overall increase in bandwidth is expected to be small. Memory requirements at prover and verifier are an issue but are mitigated by small packets (§6.3).

The real snag is the middlebox's resource requirements, independent of configuration. Despite batching, which can experimentally increase throughput by almost 5×, the middlebox still requires at least 38 ms per packet on a single thread (§6.2), which is too high for most applications.

Consequently, Zombie is not truly practical outside of a few specific use cases: policy-relevant traffic must be a small fraction of all traffic, with small packets, and ideally occurs over multi-packet flows, to amortize channel opening (§3). Two promising examples are enforcing policy over DNS requests and over search engine queries.

Apart from performance, a number of concerns remain for real-world deployment. Zombie only supports public, stateless, read-only policies. Although confidential policies with zero-knowledge middleboxes are possible in principle [49], they require extra round-trips, and composing them with batching and asynchrony is an open challenge. Supporting stateful and write-based policies in the ZKMB framework is also conceptually possible, but we leave this to future work.

To deploy and update Zombie, clients need to learn the setup, such as the proving algorithm and the subcomputations (§3), when they first connect to the network. This material can be supplied or stored in browsers or as a required download to use the network.

To avoid barring honest clients from the network, the middlebox needs to handle dropped packets and proofs gracefully. One option is for the middlebox to send an acknowledgment when it receives a proof. If the client does not receive a timely acknowledgment, it will know to resend the proof. Another concern is soundness under load: the middlebox cannot simply skip verification, say if it runs out of storage (§6.2). Instead it must drop proofs and packets, and expect clients to resend. In the asynchronous setting, the middlebox can go so far as to request batch proofs from specific clients when it has freed up space for them.

Finally, the middlebox needs a way to identify, and deal with, clients who violate policy. This concern is not specific to the ZKMB paradigm, and is left to network administrators.

Ultimately, despite the sometimes equivocal performance results, Zombie has taken a substantial step forward in demonstrating the possible practicality of the ZKMB paradigm.

Acknowledgments

We thank the anonymous reviewers for constructive comments that improved this work. This research was supported by DARPA under Agreement No. HR00112020022. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Government or DARPA.

The code for Zombie is available at
https://github.com/PepperSieve/Zombie

A Details of SpartanBatch

A.1 Spartan protocol details

This section provides a sketch of the Spartan algorithm, focusing on the parts relevant to our extension of it to support batching.

For an R1CS instance $\mathcal{C}$ with matrices (A,B,C) and public input $\mathbf{X}$, public output $\mathbf{Y}$, and witness $\mathbf{W}$, Spartan works by transforming the validity check for $(\mathbf{X},\mathbf{Y})$ into a polynomial that equals zero at every point if (and only if) $(\mathbf{X},\mathbf{Y})$ satisfies the constraints $\mathcal{C}$. This polynomial is large, and some of its coefficients are elements of $\mathbf{W}$, so the verifier cannot do this check itself; instead, the prover and verifier engage in a subprotocol that lets the verifier check whether the polynomial is zero efficiently, by evaluating it at a random point. The details of this process are unimportant for us, save for one: in the last step of the subprotocol, the verifier must evaluate special polynomial encodings (a *multilinear extension*) $\tilde{A},\tilde{B},\tilde{C}$ of each R1CS matrices A,B,C at a random point. These evaluations are the most expensive part of the protocol for the verifier; in fact, they are asymptotically as expensive as rerunning the entire computation. The random point that the polynomial encodings are to be evaluated on is of the form (r_x,r_y) where $r_x,r_y \in \mathbb{F}^{\log n}$ and each element of the r_x and r_y is a random value provided by the verifier in some step of the protocol.

In the non-interactive version of Spartan, these are chosen by hashing prefixes of the proof as the prover generates it (that is, via the Fiat-Shamir transform [37]). We observe that, since these expensive evaluations: $\tilde{A}(r_x,r_y)$, $\tilde{B}(r_x,r_y)$, and $\tilde{C}(r_x,r_y)$, depend only on the R1CS statement and not the input (in our setting, the ciphertext), they can be done just once for a batch of proofs as long as each of their respective subprotocols "coordinate", that is, use the same r_x,r_y values. We ensure this by having the prover hash the prefixes of each proof in the batch together, instead of separately. (Some hashing steps in Spartan generate randomness that is not part of r_x or r_y; we do not batch generate randomness for these steps.) We call the resulting protocol SpartanBatch.

Below, we show SpartanBatch retains the security guarantees of Spartan; in particular, we show that a malicious client has about the same (very low) probability of proving a false statement with SpartanBatch as it does with Spartan. Our analysis is based on analogous results for AND-composition in Σ-protocols and related results [95, 113].

A.2 SpartanBatch and its security proof

We can define (interactive) SpartanBatch from (interactive) Spartan below. Applying the standard Fiat-Shamir heuristic results in the non-interactive version described above. (Interactive) SpartanBatch is b parallel instances of (interactive) Spartan with the verifier following two different methods depending on the step involved:

- **Coordinated** steps are those where the verifier provides a random value that is an element of r_x or r_y (which are each in $\mathbb{F}^{\log n}$), and thus part of the evaluation point (r_x,r_y) for the polynomials $\widetilde{A},\widetilde{B},\widetilde{C}$ in the final step of Spartan's verification algorithm. In these steps, the SpartanBatch Verifier provides a *single* random value that is taken to be the response to all b parallel instances.

- All other steps are **uncoordinated**. Here, the verifier provides a b-tuple of independent responses, one for each parallel proof.

The analysis below is based on that of similar techniques applied in other proof systems [95, 113].

Theorem 1 *SpartanBatch is a succinct non-interactive argument of knowledge for the language $\mathcal{L}^b$, where b is the batch size.*

Proof:

We analyze the *interactive* version of SpartanBatch, noting that all the properties proven below are retained when using the standard Fiat-Shamir heuristic to obtain non-interactivity.

Completeness: The verifier for SpartanBatch can be seen as performing the checks of b separate Spartan verifiers. Completeness is thus immediate from the completeness of Spartan.

Soundness: Using an argument similar to standard AND-composition analysis in Σ-protocols, we show that the soundness error of SpartanBatch is *at most* the soundness error (ε) of Spartan. The proof proceeds by contradiction. Assume that there exists a false instance $x^* \notin \mathcal{L}$ and a SpartanBatch prover P_B that produces a convincing proof of a batch of statements $X^* = \{x^*, x_2, \ldots, x_b\}$ with probability $\geq 1-\varepsilon$ (we place the false instance in the first position without loss of generality). We use P_B to construct a Spartan prover P that convinces a Spartan verifier V of the same false statement x^* with the same probability.

In this reduction, P doubles as the SpartanBatch verifier when interacting with P_B: that is, $\langle P,V\rangle$ run an instance of Spartan on input x^* while $\langle P_B,P\rangle$ run an instance of SpartanBatch on input X^*. The reduction proceeds as follows:

- When P_B provides a tuple of values, P forwards the value corresponding to the false instance (here, the first one) to V.

- When V sends randomness r, P forwards the following to P_B based on the step:

  - In coordinated steps: P forwards r.
  - In uncoordinated steps: P sample randomness $r_2,\ldots,r_b$ and forwards $(r,r_2,\ldots,r_b)$.

Thus, if P_B passes the SpartanBatch verification checks, P must pass the Spartan verification checks. This is a contradiction as P_B was assumed to pass with probability $\geq 1-\varepsilon$.

Zero-knowledge: Like Spartan, SpartanBatch being a public-coin interactive protocol allows us to leverage existing compilers to satisfy zero-knowledge [116].

Knowledge soundness: We prove the stronger notion of witness-extended emulation. As this property is satisfied by Spartan, we have an emulator E that interacts with any Spartan prover P as an oracle and is allowed to rewind P to any step and resume with new verifier randomness. Using E, we construct $E_B^{P_B}$ that runs on input $X = \{x_1, \ldots, x_b\}$ interacting with a SpartanBatch prover P_B as follows:

For $i \in 0 \ldots b$:

- E_B runs emulator E on input x_i.
- When E sends randomness r to its oracle, E_B sends the following values to its oracle P_B based on the step:
  - In coordinated steps, E_B forwards value r.
  - In uncoordinated steps, E_B forwards a b-tuple with value r in position i and freshly sampled randomness in all other positions.
- When P_B responds with a tuple of values, E_B forwards the value at position i to E as a response to E's oracle query.
- When E rewinds its prover P to a step, E_B rewinds P_B (and thus rewinding all parallel instances in the batch) to that step, as well.

This way, E_B accurately simulates the required oracle for E and thus has it extract witness w_i for all inputs x_i in the batch. As E_B sequentially runs E on b inputs, E_B also runs in expected polynomial time when b is a constant.

References

[1] BNF converter. http://bnfc.digitalgrammars.com/.

[2] Efficient string matching: an aid to bibliographic search. *Communications of The ACM*, 18(6):333–340, 1975.

[3] zk-regex. https://github.com/zkemail/zk-regex, 2023.

[4] Shashank Agrawal, Chaya Ganesh, and Payman Mohassel. Non-Interactive Zero-Knowledge Proofs for Composite Statements. In *CRYPTO*, 2018.

[5] HTTP Archive. Web almanac HTTP archive's annual state of the web report. https://almanac.httparchive.org/en/2022/page-weight#request-bytes.

[6] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof Verification and the Hardness of Approximation Problems. *Journal of the ACM*, 45(3), May 1998.

[7] Sanjeev Arora and Shmuel Safra. Probabilistic Checking of Proofs: A New Characterization of NP. *Journal of the ACM*, 45(1), January 1998.

[8] László Babai and Lance Fortnow. Arithmetization: A new method in structural complexity theory. *Computational Complexity*, 1:41–66, March 1991.

[9] László Babai, Lance Fortnow, Leonid A Levin, and Mario Szegedy. Checking Computations in Polylogarithmic Time. In *ACM STOC*, 1991.

[10] Joonsang Baek, Jongkil Kim, and Willy Susilo. Inspecting TLS anytime anywhere: a new approach to TLS interception. In *Asia CCS*, 2020.

[11] Daniel J. Bernstein. Curve25519: new diffie-hellman speed records. https://cr.yp.to/ecdh/curve25519-20060209.pdf, 2006.

[12] Karthikeyan Bhargavan, Ioana Boureanu, Antoine Delignat-Lavaud, Pierre-Alain Fouque, and Cristina Onete. A formal treatment of accountable proxying over TLS. In *IEEE Symposium on Security and Privacy*, 2018.

[13] Anas A Bisu, Alan Purvis, Katharine Brigham, and Hongjian Sun. A framework for end-to-end latency measurements in a satellite network environment. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2018.

[14] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable Delay Functions. In *CRYPTO*, 2018.

[15] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. In *IEEE Symposium on Security and Privacy*, pages 947–964. IEEE, 2020.

[16] Benjamin Braun. Compiling computations to constraints for verified computation. UT Austin Honors thesis HR-12-10, December 2012.

[17] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *ACM SOSP*, 2013.

[18] Broadcom Near Real-Time Scan. https://techdocs.broadcom.com/us/en/symantec-security-software/endpoint-security-and-management/cloud-workload-protection-for-storage/1-0/Scan_Configuration_7/about-near-real-time-scan-v123769597-d4995e65807.html, 2023.

[19] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. In *Financial Crypto*, 2020.

[20] Matteo Campanelli, Dario Fiore, and Anaïs Querol. LegoSNARK: Modular design and composition of succinct zero-knowledge proofs. In *ACM CCS*, page 2075–2092, 2019.

[21] Weikeng Chen, Alessandro Chiesa, Emma Dauterman, and Nicholas P. Ward. Reducing Participation Costs via Incremental Verification for Ledger Systems. Cryptology ePrint Archive, Paper 2020/1522, 2020.

[22] Cisco. Snort intrusion detection system. https://www.snort.org/.

[23] Cisco Umbrella. https://umbrella.cisco.com/, 2023.

[24] Jeremy Clark and Paul C Van Oorschot. SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements. In *IEEE Symposium on Security and Privacy*, 2013.

[25] Russ Cox. Regular expression matching can be simple and fast. https://swtch.com/ rsc/regexp/regexp1.html, 2007.

[26] Santiago Cuéllar, Bill Harris, James Parker, Stuart Pernsteiner, and Eran Tromer. Cheesecloth: Zero-Knowledge Proofs of Real-World Vulnerabilities. *arXiv preprint arXiv:2301.01321*, 2023.

[27] Gaby G Dagher, Benedikt Bünz, Joseph Bonneau, Jeremy Clark, and Dan Boneh. Provisions: Privacy-preserving Proofs of Solvency for Bitcoin Exchanges. In *ACM CCS*.

[28] George Danezis, Cedric Fournet, Markulf Kohlweiss, and Bryan Parno. Pinocchio Coin: building Zerocoin from a succinct pairing-based proof system. In *ACM workshop on Language Support for Privacy-Enhancing Technologies*, 2013.

[29] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, and Titouan Tanguy. Limbo: Efficient zero-knowledge MPCitH-based arguments. In *ACM CCS*, CCS '21, page 3022–3036, New York, NY, USA, 2021. Association for Computing Machinery.

[30] Tim Dierks and Eric Rescorla. RFC 5246: The transport layer security (TLS) protocol version 1.2. RFC 5246, 2008.

[31] Colin Dixon, Hardeep Uppal, Vjekoslav Brajkovic, Dane Brandon, Thomas Anderson, and Arvind Krishnamurthy. ETTM: A scalable fault tolerant network manager. In *USENIX NSDI*, 2011.

[32] Huayi Duan, Xingliang Yuan, and Cong Wang. Lightbox: SGX-assisted secure network functions at near-native speed. *arXiv preprint arXiv:1706.06261*, 2017.

[33] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[34] Jacob Eberhardt and Stefan Tai. ZoKrates - Scalable Privacy-Preserving Off-Chain Computations. In *IEEE Conference on Internet of Things (iThings)*, 2018.

[35] Jingyuan Fan, Chaowen Guan, Kui Ren, Yong Cui, and Chunming Qiao. Spabox: Safeguarding privacy during deep packet inspection at a middlebox. *IEEE/ACM Transactions on Networking*, 25(6), 2017.

[36] Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. Security vulnerabilities of SGX and countermeasures: A survey. *ACM Computing Surveys*, 54(6), 2021.

[37] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, 1986.

[38] Fortra Digital Guardian. https://www.digitalguardian.com/, 2023.

[39] Open Information Security Foundation. Suricata intrusion detection system. https://suricata.io/.

[40] Sanjam Garg, Abhishek Jain, and Amit Sahai. Leakage-Resilient Zero Knowledge. In *CRYPTO*, 2011.

[41] Oded Goldreich. Probabilistic proof systems – a primer. *Foundations and Trends in Theoretical Computer Science*, 3(1), 2008.

[42] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. Delegating computation: interactive proofs for muggles. *Journal of the ACM*, 62(4), 2015.

[43] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1), 1989.

[44] David Goltzsche, Signe Rüsch, Manuel Nieke, Sébastien Vaucher, Nico Weichbrodt, Valerio Schiavoni, Pierre-Louis Aublin, Paolo Cosa, Christof Fetzer, Pascal Felber, et al. Endbox: Scalable middlebox functions using client-side trusted execution. In *IEEE/IFIP DSN*, 2018.

[45] Deli Gong, Muoi Tran, Shweta Shinde, Hao Jin, Vyas Sekar, Prateek Saxena, and Min Suk Kang. Practical verifiable in-network filtering for DDoS defense. In *IEEE ICDCS*, 2019.

[46] Google. Google public DNS. https://developers.google.com/speed/public-dns.

[47] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[48] Jens Groth. On the size of pairing-based non-interactive arguments. In *IACR Eurocrypt*, 2016.

[49] Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish. Zero-Knowledge Middleboxes. In *USENIX Security*, 2022.

[50] Juhyeng Han, Seongmin Kim, Daeyang Cho, Byungkwon Choi, Jaehyeong Ha, and Dongsu Han. A Secure Middlebox Framework for Enabling Visibility Over Multiple Encryption Protocols. *IEEE/ACM Transactions on Networking*, 28(6), 2020.

[51] Juhyeng Han, Seongmin Kim, Jaehyeong Ha, and Dongsu Han. SGX-Box: Enabling visibility on encrypted traffic using a secure middlebox module. In *Asia-Pacific Workshop on Networking*, 2017.

[52] Paul E. Hoffman and Patrick McManus. DNS Queries over HTTPS (DoH). RFC 8484, 2018.

[53] Zi Hu, Liang Zhu, John Heidemann, Allison Mankin, Duane Wessels, and Paul E. Hoffman. Specification for DNS over Transport Layer Security (TLS). RFC 7858, 2016.

[54] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *ACM STOC*, STOC '07, page 21–30, New York, NY, USA, 2007. Association for Computing Machinery.

[55] Dmitry Khovratovich, Mary Maller, and Pratyush Ranjan Tiwari. MinRoot: Candidate Sequential Function for Ethereum VDF. Cryptology ePrint Archive, Paper 2022/1626, 2022.

[56] Jongkil Kim, Seyit Camtepe, Joonsang Baek, Willy Susilo, Josef Pieprzyk, and Surya Nepal. P2DPI: Practical and Privacy-Preserving Deep Packet Inspection. *AsiaCCS*, 2021.

[57] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. In *IEEE Symposium on Security and Privacy*, 2016.

[58] Feodor Kulishov. DFA-based and SIMD NFA-based regular expression matching on cell BE for fast network traffic filtering. In *2nd Intl. Conference on Security of Information and Networks (SIN)*. ACM Press, 2009.

[59] Dmitrii Kuvaiskii, Somnath Chakrabarti, and Mona Vij. Snort intrusion detection system with Intel software guard extension (Intel SGX). *arXiv preprint arXiv:1802.00508*, 2018.

[60] SSL Labs. SSL Pulse. https://www.ssllabs.com/ssl-

pulse/.

[61] Shangqi Lai, Xingliang Yuan, Joseph K Liu, Xun Yi, Qi Li, Dongxi Liu, and Surya Nepal. OblivSketch: Oblivious Network Measurement as a Cloud Service. In *Network and Distributed System Security Symposium*, 2021.

[62] Shangqi Lai, Xingliang Yuan, Shifeng Sun, Joseph K. Liu, Ron Steinfeld, Amin Sakzad, and Dongxi Liu. Practical Encrypted Network Traffic Pattern Matching for Secure Middleboxes. *IEEE TDSC*, 2021.

[63] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: Securely outsourcing middleboxes to the cloud. In *USENIX NSDI*, 2016.

[64] Hyunwoo Lee, Zach Smith, Junghwan Lim, Gyeongjae Choi, Selin Chun, Taejoong Chung, and Ted Taekyoung Kwon. maTLS: How to make TLS middlebox-aware? In *Network and Distributed System Security Symposium*, 2019.

[65] Jie Li, Rongmao Chen, Jinshu Su, Xinyi Huang, and Xiaofeng Wang. ME-TLS: Middlebox-enhanced TLS for Internet-of-Things devices. *IEEE Internet of Things Journal*, 7(2), November 2019.

[66] Cong Liu, Yong Cui, Kun Tan, Quan Fan, Kui Ren, and Jianping Wu. Building generic scalable middlebox services over encrypted protocols. In *IEEE INFOCOM*, 2018.

[67] Yanbing Liu, Li Guo, Ping Liu, and Jianlong Tan. Compressing regular expressions' dfa table by matrix decomposition. In Michael Domaratzki and Kai Salomaa, editors, *Implementation and Application of Automata*, pages 282–289, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[68] Ning Luo, Chenkai Weng, Jaspal Singh, Gefei Tan, Ruzica Piskac, and Mariana Raykova. Privacy-preserving regular expression matching using nondeterministic finite automata. Cryptology ePrint Archive, Paper 2023/643, 2023. https://eprint.iacr.org/2023/643.

[69] Chad Mayfield. my-pihole-blocklists/pi_blocklist_porn_all.list. https://github.com/chadmayfield/my-pihole-blocklists, 2021.

[70] François Michel, Martino Trevisan, Danilo Giordano, and Olivier Bonaventure. A First Look at Starlink Performance. In *Proceedings of the Internet Measurement Conference*, 2022.

[71] Microsoft. Data loss prevention. learn.microsoft.com/en-us/microsoft-365/compliance/dlp-learn-about-dlp.

[72] Microsoft. U.S. bank account number. https://learn.microsoft.com/en-us/microsoft-365/compliance/sit-defn-us-bank-account-number.

[73] Microsoft. U.S. drivers license number. https://learn.microsoft.com/en-us/microsoft-365/compliance/sit-defn-us-drivers-license-number.

[74] Microsoft. U.S. individual taxpayer identification number. https://learn.microsoft.com/en-us/microsoft-365/compliance/sit-defn-us-individual-taxpayer-identification-number.

[75] Microsoft. U.S. social security number. https://learn.microsoft.com/en-us/microsoft-365/compliance/sit-defn-us-social-security-number.

[76] Microsoft. U.S./U.K. passport number. https://learn.microsoft.com/en-us/microsoft-365/compliance/sit-defn-us-uk-passport-number.

[77] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. In *IEEE Symposium on Security and Privacy*, 2020.

[78] David Naylor, Richard Li, Christos Gkantsidis, Thomas Karagiannis, and Peter Steenkiste. And Then There Were More: Secure Communication for More Than Two Parties. In *ACM CoNEXT*, 2017.

[79] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R. López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. Multi-context TLS (McTLS): Enabling secure in-network functionality in TLS. In *ACM SIGCOMM*, 2015.

[80] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A survey of published attacks on Intel SGX. *arXiv preprint arXiv:2006.13598*, 2020.

[81] Jianting Ning, Xinyi Huang, Geong Sen Poh, Shengmin Xu, Jia-Chng Loh, Jian Weng, and Robert H Deng. Pine: Enabling privacy-preserving deep packet inspection on TLS with rule-hiding and fast connection establishment. In *ESORICS*, 2020.

[82] Jianting Ning, Geong Sen Poh, Jia-Ch'ng Loh, Jason Chia, and Ee-Chien Chang. PrivDPI: privacy-preserving encrypted traffic inspection with reusable obfuscated rules. In *ACM CCS*, 2019.

[83] OpenWrt. OpenWrt table of hardware. https://openwrt.org/toh/views/toh_extended_all.

[84] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. CirC: compiler infrastructure for proof systems, software verification, and more. In *IEEE Symposium on Security and Privacy*, 2022.

[85] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, 2013.

[86] PerfOps. Dns performance analytics and comparison. https://www.dnsperf.com/#!dns-resolvers.

[87] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Safebricks: Shielding network functions in the cloud. In *USENIX NSDI*, 2018.

[88] Geong Sen Poh, Dinil Mon Divakaran, Hoon Wei Lim, Jianting Ning, and Achintya Desai. A Survey of Privacy-Preserving Techniques for Encrypted Traffic Inspection over Network Middleboxes. *arXiv preprint arXiv:2101.04338*, 2021.

[89] Michael Raymond, Gillian Evers, Jan Ponti, Diya Krishnan, and Xiang Fu. Efficient zero knowledge for regular language. Cryptology ePrint Archive, Paper 2023/907, 2023. https://eprint.iacr.org/2023/907.

[90] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, 2018.

[91] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *IEEE Symposium on Security and Privacy*, 2014.

[92] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *DIMVA*, 2017.

[93] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *IACR CRYPTO*, 2020.

[94] Srinath Setty, Benjamin Braun, Victor Vu, Andrew J. Blumberg, Bryan Parno, and Michael Walfish. Resolving the conflict between generality and plausibility in verified computation. In *Eurosys*, 2013.

[95] Srinath Setty, Richard McPherson, Andrew Blumberg, and Michael Walfish. Making argument systems for outsourced computation practical (sometimes). 2012.

[96] Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security*, 2012.

[97] Adi Shamir. IP = PSPACE. *J. ACM*, 39(4):869–877, oct 1992.

[98] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. BlindBox: Deep packet inspection over encrypted traffic. In *ACM SIGCOMM*, 2015.

[99] Justine M. Sherry. *Middleboxes as a Cloud Service*. PhD thesis, University of California, Berkeley, 2016.

[100] Adhiraj Singh, Madhavan Malolan, and Abhilash Inumella. Reclaim Protocol: Privacy preserving consensus to export reputation from webservers. https://www.reclaimprotocol.org/, 2022.

[101] Michael Sipser. *Introduction to the Theory of Computation*. Boston, MA, third edition, 2013.

[102] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W Fletcher. Microscope: Enabling microarchitectural replay attacks. In *ISCA*, 2019.

[103] Justin Thaler. Proofs, Arguments, and Zero-Knowledge. http://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html, 2020.

[104] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of The ACM*, 11(6):419–422, June 1968.

[105] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. Shieldbox: Secure middleboxes using shielded execution. In *Symposium on SDN Research*, 2018.

[106] Florian Tramèr, Dan Boneh, and Kenny Paterson. Remote side-channel attacks on anonymous transactions. In *USENIX Security*, 2020.

[107] Nirvan Tyagi, Ben Fisch, Andrew Zitek, Joseph Bonneau, and Stefano Tessaro. VeRSA: Verifiable Registries with Efficient Client Audits from RSA Authenticated Dictionaries. In *ACM CCS*, 2022.

[108] Ioanna Tzialla, Abhiram Kothapalli, Bryan Parno, and Srinath Setty. Transparency Dictionaries with Succinct Proofs of Correct Operation. In *Network and Distributed System Security Symposium*, 2022.

[109] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, 2018.

[110] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *IEEE Symposium on Security and Privacy*, 2020.

[111] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. SGAxe: How SGX fails in practice. https://sgaxeattack.com/, 2020.

[112] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. In *IEEE Symposium on Security and Privacy*, 2021.

[113] Victor Vu, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. A hybrid architecture for interactive verifiable computation. In *IEEE Symposium on Security and Privacy*, 2013.

[114] W3Schools. Chrome statistics. https://www.w3schools.com/browsers/browsers_chrome.asp.

[115] Riad S. Wahby, Srinath Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *Network and Distributed System Security Symposium*, 2015.

[116] Riad S. Wahby, Ioanna Tzialla, abhi shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *IEEE Symposium on Security and Privacy*, 2018.

[117] Michael Walfish and Andrew J. Blumberg. Verifying computations without reexecuting them: from theoretical possibility to near practicality. *Communications of the ACM*, 58(2), February 2015.

[118] Juan Wang, Shirong Hao, Yi Li, Zhi Hong, Fei Yan, Bo Zhao, Jing Ma, and Huanguo Zhang. TVIDS: Trusted virtual IDS with SGX. *China Communications*, 16(10), 2019.

[119] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan: A fast multi-pattern regex matcher for modern CPUs. In *USENIX NSDI*, 2019.

[120] Yu Wang, Yang Xiang, Wanlei Zhou, and Shunzheng Yu. Generating regular expression signatures for network traffic classification in trusted network management. *Journal of Network and Computer Applications*, 35(3):992–1000, 2012. Special Issue on Trusted Computing and Communications.

[121] Florian Wilkens, Steffen Haas, Johanna Amann, and Mathias Fischer. Passive, transparent, and selective TLS decryption for network security monitoring. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, 2022.

[122] Xingliang Yuan, Huayi Duan, and Cong Wang. Assuring string pattern matching in outsourced middleboxes. *IEEE/ACM Transactions on Networking*, 26(3), 2018.

[123] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. DECO: Liberating web data using decentralized oracles for TLS. In *ACM CCS*, 2020.

[124] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *IEEE Symposium on Security and Privacy*, 2017.