Zombie: Middleboxes that Don't Snoop

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

[*]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, provide a factor of 3.5× speedup in total computation done by client and middlebox, lowering the critical path overhead for a DNS filtering application to less than 300ms (on commodity hardware) or (in the asynchronous configuration) to 0.

As an additional contribution that is likely of independent interest, Zombie introduces a portfolio of techniques to efficiently encode regular expressions in probabilistic (and zero knowledge) proofs; these techniques offer significant asymptotic and constant factor improvements in performance over a standard baseline. 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 this approach: by design, middleboxes 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 completely.

Resolving this conflict has been a goal of the network security research community for some time. While many approaches exist, they have thus far fallen into two categories, each with significant downsides. 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 [60, 76, 95] (§7). Changing TLS is a huge task though: it took ten years of extensive design effort to go from TLS 1.2 [27] to TLS 1.3 [87]. 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 [57]. Furthermore, implementing TLS securely is notoriously complex and subtle [21], meaning that protocol changes are risky. Second, by contrast, are middleboxes designed to work with standard TLS-encrypted traffic but require users to disclose keys to trusted hardware enclaves (TEEs) to enforce policy [28, 40]. Unfortunately, relying on trusted hardware is an increasingly unappealing risk given the litany of attacks on TEE implementations [43, 74, 89, 99, 105–108].

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 push code updates to the vast majority of users within months [110].

Zero-knowledge middleboxes. We build on top of the recently proposed ZKMB paradigm [45]. With ZKMBs, clients can prove (in zero knowledge [39]) to the middlebox that the plaintext underlying their encrypted traffic is policy-compliant. Middleboxes verify these proofs, ensuring that only policy-compliant traffic is allowed to pass, while learning nothing about the underlying plaintext beyond the fact that it is policy-compliant. ZKMBs require no changes to existing encryption protocols, no trusted hardware, and are extensible in principle to verifying any network policy. Thus they promise an elegant solution to the policy vs. privacy conflict. Unfortunately, the initial 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 of checking DNS queries against a static block/allow list.

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 contributes a set of context-specific techniques to reduce end-to-end delay (both client proving costs and middlebox verifying costs on the critical path). First, Zombie splits the computation to be proved, moving one part off the critical path, to be precomputed (and pre-proved) 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? The crucial observation is that TLS 1.3 uses only stream ciphers (AES-CTR and ChaCha20). This enables clients to compute the key stream before plaintext is available, and only compute an XOR of the key stream with the plaintext on the critical path.

Zombie's next performance enhancement is *optimistic approval* (§3.2). Middleboxes can perform proof validation *after* forwarding packets and therefore off the critical path. 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, with consequences (for example, future blocklisting) if clients don't supply a valid proof. A similar approach, *near-real-time verification*, is already taken by some real-world middleboxes [15, 20, 35].

Finally, Zombie supports *batch proof verification* by the middlebox, reducing the overall verification burden by amortizing it across proofs for multiple packets (§3.3). Optimistic approval complements batch proof verification, allowing the middlebox to postpone verification until a larger batch has been assembled. Batch verification offers significant savings, increasing middlebox throughput by almost 5× in our experiments (§6).

Another set of contributions enables Zombie to handle policies based on regular expression matching, a crucial building block in various kinds of middleboxes, including intrusion detection systems (IDS), network traffic classification, and data loss prevention (DLP). Regular expression matching is tricky to implement efficiently in proof frameworks. The core challenge is that, in order to represent *any* computation to be proved, one must translate it to arithmetic circuits or constraints, an inefficient and inhospitable formalism (§2). 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, a new finite automaton formalism, optimizations to preprocess regular expressions to target the new encodings, and an efficient way to perform context-dependent matches. These techniques apply well beyond regular-expression matching and are very likely to be of independent interest for other applications of probabilistic proofs.

We implement Zombie for TLS 1.3 with the ChaCha20/Poly1305 cipher suite (§5). The result of all this work is near-practicality for some ZKMB uses (§6). For example, in the precomputation regime, Zombie adds less than 300 ms of delay to DNS queries. This may be tolerable; by comparison, traditional satellite Internet connections[1] typically add at least 600 ms of latency [10]. Furthermore, in the optimistic approval regime, there is almost no critical path delay from Zombie.

Limitations. While our implementation shows that Zombie can offer practical performance in some application scenarios, this carries several important caveats. First, its proofs are large: for DNS, roughly ten times the size of DNS query traffic itself (although these proofs never leave the local network). Zombie also relies on bursty workloads (providing enough downtime to precompute and post-verify proofs). Optimistic approval requires state-keeping by the middlebox. Also, our implementation is heavily tailored to TLS 1.3; our stream cipher precomputation technique would not work for TLS 1.2. Optimizing for other end-to-end encrypted protocols is an important open problem. Zombie also inherits some of the general limitations of the ZKMB paradigm. As examples, identifying policy-relevant traffic can be difficult for some policies, and using local commitments to middlebox state to apply policies that span multiple packets is open work.

2 Background

We present some basic background on zero-knowledge proofs (ZKPs). There is a deep cryptographic literature on ZKPs; for a general overview, we refer the reader to Thaler [100].

Overview of 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 $(\mathbf{X},\mathbf{Y})$. 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

[1] Modern low-earth-orbit satellite Internet service offers significantly lower latency, as low as 25 ms [67].

zero-knowledge have precise cryptographic definitions that we elide here; Zombie inherits these properties directly from the underlying cryptographic tools.

A concrete example. Consider using ZKPs to prove that an encrypted packet does not contain a DNS query for a blocked domain [45, §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 included in the blocklist. Note that, by design, this statement cannot be efficiently checked using just $\mathbf{X}$ and $\mathbf{Y}$ without knowing $\mathbf{W}$ (the decryption key).

Zero-knowledge proof pipelines. Most generic ZKP schemes decompose into a *front-end* and a *back-end*. The front-end takes 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:

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

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 [13, 14, 82, 91, 93, 111, 120].

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` [93, Appx D]:

$$\begin{Bmatrix} y \cdot x = 0 \\ W \cdot x = 1 - y \end{Bmatrix}$$

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:

$$\begin{array}{c} \begin{matrix} x & y & 1 & W \end{matrix} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ A \end{array} \quad \begin{array}{c} \begin{matrix} x & y & 1 & W \end{matrix} \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \\ B \end{array} \quad \begin{array}{c} \begin{matrix} x & y & 1 & W \end{matrix} \\ \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 \end{bmatrix} \\ C \end{array}$$

Spartan ZKP. As its back-end, Zombie uses Spartan [90], specifically the SpartanNIZK variant (which we refer to as just Spartan for simplicity). Spartan is a non-interactive ZKP protocol that strikes an attractive balance among prover time, verifier time, and proof size. It also has a *transparent setup*, meaning that there are no secrets required during parameter generation, only a secure source of public randomness to seed the setup algorithm. A transparent setup makes it feasible to establish global public parameters that provers (clients) can use across different verifiers (networks).

For an R1CS instance $\mathcal{C}$ with 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 a special polynomial encoding (a *multilinear extension*) of each R1CS matrix at a random point. These evaluations are the single most expensive part of the protocol for the verifier; in fact, they are asymptotically as expensive as re-running the entire computation.

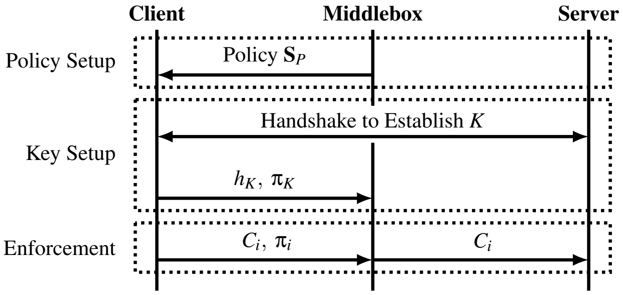


Figure 1: The ZKMB paradigm with amortized key setup [45] divided into 3 phases. The *policy setup* step occurs once when the client connects to the middlebox; the middlebox sends the policy $\mathbf{S}_P$ to the client. The *key setup* step occurs once per session; it involves a handshake between the client and the server, a commitment to a session key K, and a proof of this commitment via $\mathbf{S}_{E.1}$. 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.

3 Zombie's protocol

Zombie is built using the zero-knowledge middlebox (ZKMB) paradigm [45]. In this section, we begin by describing the existing ZKMB approach and establishing notation, as context for what follows. We then describe Zombie's enhancements: precomputation (§3.1), asynchronous verification mode (§3.2), and batching (§3.3).

Figure 1 sketches the protocol flow for this paradigm. The middlebox's goal is to ensure that clients are following some usage policy P that pertains to the traffic that they send to a server; a client's goal is to communicate with a server using some encryption protocol E, such as TLS 1.3.

The middlebox begins with protocol E, content type F, and policy P with the goal of enforcing P on all traffic of type F that is sent via E. It creates the following subcomputations:

(1) From E, it creates a *channel-opening* subcircuit $\mathbf{S}_E$ that takes as input a packet and the information required to re-derive a session key, and outputs the decrypted packet.

(2) From F, it creates a *parse-and-extract* subcircuit $\mathbf{S}_F$ that takes as input the decrypted packet and outputs a snippet of policy-relevant data from the packet.

(3) From P, it creates a *policy-check* subcircuit $\mathbf{S}_P$ that 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).

The middlebox sends $\mathbf{S}_P$ to each client when they join the network [45]. We follow the amortized ZKMB model, which reuses the expensive work of channel opening over multiple per-packet proofs. There are two broad phases after the middlebox communicates the policy to the client, as outlined in Figure 1. To facilitate these two phases, the TLS 1.3 channel opening subcomputation, $\mathbf{S}_E$, is split into two parts $\mathbf{S}_{E.1}$ and $\mathbf{S}_{E.2}$.

When a client wants to communicate with a particular server, it first negotiates the shared key K using a protocol known as the *handshake*, the transcript of which is public but also involves secrets known only to the client and the server. The first subcomputation part, $\mathbf{S}_{E.1}$ (*derive-and-commit*), re-derives this session key K by taking the handshake transcript as public input and the client's secrets as witness, and then hashes it to produce h_K. The client sends to the middlebox the proof π_K of this statement $\mathbf{S}_{E.1}$, along with h_K. This proof convinces the middlebox that h_K is the commitment to some key that is consistent with the handshake.

The second part, $\mathbf{S}_{E.2}$ (*decrypt*), takes the packet's ciphertext C and the key commitment h_K as public inputs, and the session key K as witness and outputs the decrypted packet after verifying that K hashes to h_K. Note that this decrypted packet is not revealed to the middlebox, but instead it serves as an input for the subcircuit $\mathbf{S}_F$. Then, whenever the client wants to send this server an encrypted packet C, it needs to convince the middlebox that C is valid with respect to h_K and the composition of the subcomputations $\mathbf{S}_{E.2}$, $\mathbf{S}_F$, and $\mathbf{S}_P$. It does so with a proof π. When the middlebox receives (C, h_K, π), it verifies π, and only then forwards C.

Zombie's enhancements. Departing from prior work, Zombie introduces three important changes to the existing ZKMB paradigm: precomputation, asynchronous verification, and batching. 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. Zombie's asynchronous verification mode (§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 expensive computations across all verify operations in a batch of proofs created by the client. Both precomputation and batching are most useful for settings where Zombie's workload is bursty.

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 $\mathbf{S}_{E.2a}$ before 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. Before continuing, we note that this technique may be more broadly relevant; it is orthogonal (and complementary) to the split between $\mathbf{S}_{E.1}$ and $\mathbf{S}_{E.2}$.

Let K be the session key output by the TLS 1.3 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, and outputs an ℓ-byte pseudorandom pad pad. For an ℓ-byte message M, its TLS 1.3 ciphertext is $\mathsf{pad} \oplus M$. This is a simplification; it omits some operations, such as computing a MAC, whose details are unimportant here.

We make two key observations. First, the inputs to PadGen are independent of the message, so the value pad can be computed by the client before M is known. Second, computing PadGen is the most expensive part of the channel-opening subcircuit $\mathbf{S}_E$. This is because PadGen involves legacy cryptographic algorithms (specifically AES or ChaCha20), which are difficult to represent in the constraint formalism used for ZKPs (§2). In our evaluation of a DNS filtering application [45] (§6), we find that PadGen contributes to over 40% of the proving time incurred by the client in each per-packet proof.

Zombie uses these observations to move PadGen into the subcomputation $\mathbf{S}_{E.2a}$ (*pad-commit* in Figure 2), out of the critical path of operations which must be performed before packets can leave the network. This involves running PadGen to produce the pad corresponding to the next sequence number SN using the secret key K provided as witness, and then computing its hash $h_{\mathsf{pad}_{\mathsf{SN}}}$. Also, K is hashed to verify that it corresponds to the hash h_K provided as a public input. The client computes the 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. Thus, when the next packet needs to be sent, the client can provide the required pad as a witness and have it be verified against the respective stored hash instead of re-running PadGen. The client can precompute any number of such proofs for future sequence numbers. As we'll see in Section 3.3, Zombie's batching technique allows the client to batch together multiple such proofs for faster verification.

The second part of the protocol, run once the Zombie client receives the plaintext M, is as follows: the client first encrypts M with the pad 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 2), which takes $\mathsf{pad}_{\mathsf{SN}}$ as the witness, verifies that it hashes to the stored hash $h_{\mathsf{pad}_{\mathsf{SN}}}$ corresponding to the right sequence number, and then passes the decrypted $M = \mathsf{pad} \oplus C$ to the next stages of the proof statement (that is, the $\mathbf{S}_F$ and $\mathbf{S}_P$ subcomputations).

The security of Zombie's precomputation reduces to the soundness and zero-knowledge properties of the proof protocol, and the hiding and binding properties of the hash function H. Neither the hash h_{pad} nor the precomputation proof reveal the pad, and the soundness of the proof system prevents the client from lying about the pad (and thus equivocating about the sent message M).

```
$pad\text{-}commit(h_K, \mathsf{SN}, \ell\,; K)$:
$\mathsf{pad}_{\mathsf{SN}} \leftarrow \mathsf{PadGen}(K, \mathsf{SN}, \ell)$
$h_{\mathsf{pad}_{\mathsf{SN}}} \leftarrow H(\mathsf{pad}_{\mathsf{SN}})$
Return $(h_K = H(K))\ ?\ h_{\mathsf{pad}_{\mathsf{SN}}} : \perp$

$decrypt\text{-}from\text{-}pad(C, h_{\mathsf{pad}_{\mathsf{SN}}}, \mathsf{SN}\,; \mathsf{pad})$:
$M \leftarrow \mathsf{pad} \oplus C$
Return $(h_{\mathsf{pad}_{\mathsf{SN}}} = H(\mathsf{pad}))\ ?\ M : \perp$
```

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

3.2 Asynchronous verification

While precomputation can greatly reduce the per-packet delay incurred by proof generation (down to a quarter of a second (§6)), it may still be too slow for latency-sensitive applications like web browsing.

In this section, we describe how Zombie can generate and verify proofs asynchronously to reduce delay. Namely, Zombie can be configured to perform the ZKP-related parts of its protocol independent of the flow of non-ZKP network traffic. Thus, client traffic can be handled more or less as it would be in a non-ZKMB network, but the middlebox can still detect policy violations retroactively.

In Zombie's asynchronous variant, first the client encrypts its packet to get the ciphertext C. The client immediately sends C to the middlebox, which forwards it to the server without delay, but notes that it expects a proof for C to be received soon. Then the client generates π and sends it to the middlebox. The middlebox checks π; if the proof is valid the middlebox does nothing. If the proof is invalid, or not received by a deadline, the middlebox may take some action such as adding the client's MAC address to a blocklist. We can view asynchronous mode as *optimistic approval* in that the middlebox forwards client packets with no delay, assuming that valid proofs will almost always be received.

In addition to reducing delay, asynchronous mode lets Zombie take full advantage of batching (that is, generating and verifying multiple proofs at once; see §3.3). Though batching can be done in Zombie even in synchronous mode, in async mode clients and middleboxes can wait for larger batches without affecting delay.

A performance drawback of asynchronous verification is an increase in the amount of state stored on the middlebox. First, the middlebox must track which packets have been optimistically forwarded and when corresponding proofs are expected. Second, recall that proof verification requires the ciphertext C as input; the proof π pertains to the contents of a specific C. Since asynchronous verification occurs after the middlebox has already sent C, it needs to remember C until it receives and verifies π. An alternative (which we do not implement) is for the middlebox to store a hash $H(C)$ instead of the full C (the hash serving as a cryptographic commitment). This would require clients to store and re-send the full value C along with π. We did not pursue this approach as it doubles bandwidth requirements on the local network (with the advantage of decreasing the middlebox's memory requirements.

Security. Asynchronous verification also changes the security model in a fundamental way. By design, with asynchronous verification the middlebox cannot prevent non-compliant packets from leaving the network; it will only eventually learn if this has occurred given if these packets are followed up with invalid or non-existent proofs. Observe that clients may also receive response packets from servers before sending proofs.

We claim, though, that this relaxed security is sufficient for many applications: for example, for DNS filtering, the goal of the policy is to prevent users from browsing blocked sites. Even if the user is able to learn 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) a file prefix is successfully uploaded. Other context-specific policies may be appropriate, for example a middlebox might optimistically send packets out but hold 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 all b underlying plaintexts are policy-compliant. Importantly, this proof is much more efficient for the middlebox to verify than b separate proofs would be. Batching is especially useful in Zombie's asynchronous mode (§3.2), because the Zombie client can wait to collect many packets before generating their proofs. It is orthogonal, but complementary, to precomputation (§3.1), as multiple $\pi_{E.2a}$ proofs can be batched.

At a high level, batching works by modifying Zombie's underlying ZKP protocol, Spartan [90]. The modification allows the MB to re-use the most expensive part of the Spartan verification algorithm for each proof in the batch. We explain the details in Appendix A.

4 Regular expressions in Zombie

This section describes how Zombie supports middlebox functionality based on regular expressions. Regular expressions feature in real-world policies for data loss prevention (DLP) [68], intrusion detection (IDS) [19, 36], and network traffic classification [115, 116]. For example, DLP systems might use a regular expression 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 regular expressions. This policy (§2) 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 policy is satisfied. The policy can be as simple as whether any substring of the payload matches the given regular expression (which we sometimes call a *regexp*). 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 P is satisfied. Thus, the prover can, and will be expected to, prove both positives and negatives. For example, if P is a simple regular expression match, and the payload does *not* contain a substring that matches the regular expression, then $\mathcal{C}_P$ will be satisfiable—and thus a back-end proof is possible—if and only if the prover correctly claims that the output of $\mathbf{S}_P$ is 0.

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.

Accordingly, unlike prior work on regular expressions [115, 116], we do not focus on making *matching* fast. Matching, for us, is the step where the prover executes its recipe to identify a satisfying assignment to the constraints. In our context, the costs of this step are swamped by the cost of proving and verifying. Those steps, particularly proving, depend on the number of constraints (§2). Accordingly, and following Section 2, our metric will be 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, which significantly reduce the size of the generated constraints (§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 $\mathbf{S}_R$ subcomputations.

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). One can think of $\mathbf{S}_R$ and its output as the usual unrolling that happens when translating a looping computation to constraints.

$\mathbf{S}_P$ 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: $R \rightarrow FA \rightarrow IR \rightarrow \mathcal{C}_R$, 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

Imagine 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 $R \rightarrow IR \rightarrow \mathcal{C}_R$. 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 [93, Appx D]):

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

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 $b_i^{(\ell)} = b_{i+1}^{(\ell)} \cdot \texttt{EQUALS-ZERO}(T[\ell-i]-\texttt{a})$, as follows:

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

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:

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

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).

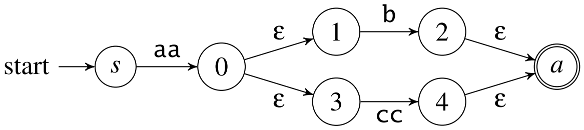
The loose limits are determined by the size of the alphabet and the size of the field that the constraints are expressed over. Assuming an alphabet of size $|\Lambda|$ and a field of size q, the maximum length of a pattern that can be compiled into a single `STRING-MATCH` is $\lfloor \log_{|\Lambda|}(q) \rfloor$. For our application, we consider the alphabet of ASCII characters ($|\Lambda| = 2^8$) and a 255-bit prime field (the base field of curve25519 [9]). This means that with our parameters, 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-MATCH`s, connected by `AND` ($\wedge$).

4.3 From regular expressions to constraints

Real-world systems [22, 55, 64] translate regular expressions to executable code in two steps. First, they produce a nondeterministic finite automaton (NFA), via Thompson's algorithm [101]. Second, they determinize the NFA to get a DFA [98, Ch. 1]. The second step represents the DFA's state transition function as a table: an entry for every state and every character. This approach makes execution very fast. However, in our context, the entire exponentially-sized table would turn into constraints, dramatically slowing the prover's and verifier's running time.

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.

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

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 either `b` or `c`), then the number of constraints is unchanged (assuming the loose limits on k given earlier).

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 [7, 82, 91–94]:

$r := p \wedge q$	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 variables) cost no constraints. We will use this fact over and over again.

Consequently, `OR` has become mostly free: addition of degree-1 terms doesn't require constraints (because the sum is a linear combination, as above). We say *mostly* because, for this to work, $p+q$ must be prohibited from overflowing, that is, wrapping around the finite field modulus and becoming 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. The specific constraints are `NOT-EQUALS-ZERO` [93, 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 the example above 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. This point could be of independent interest, as it 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 to rewrite the regular expression 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):

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

Whereas the formulation in (2) costs 8 constraints for each character position ℓ, expression (3) costs 4 constraints (two for each `STRING-MATCH`, with f_a being a linear combination and thus not costing a constraint).

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*:

$$\begin{aligned} t_0[\ell] &:= \texttt{MATCH-CLASS}(T[\ell], \texttt{[0-9]}) \\ f_a(\ell) &:= \texttt{STRING-MATCH}(t_0[\ell-2]T[\ell-1]t_0[\ell], 1_{\mathbb{F}}\texttt{a}1_{\mathbb{F}}) \end{aligned} \tag{4}$$

$1_{\mathbb{F}}$ 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, one might wonder whether we could use deduplication on that earlier example. The answer is no, because the union components were different lengths.

As another optimization, Zombie exploits structure in the character class. For example, Zombie encodes the regexp `[A-Za-z]` with only 25 constraints (fewer than the number of characters in the class!). Here is an example of the basic idea:

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

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 [93], which is 9 if Λ is the 8-bit ASCII characters. Given a large character class, the idea of using inclusion, exclusion, and range encoding is beneficial. The Zombie compiler attempts to optimize the encoding of a particular class using the best arithmetic tools possible; for example, treating this class `[0-9]` as a range with `<=` and `>=` operators is not worthwhile.

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 matching and non-matching expressions. Suppose the higher-level policy of interest, P, is whether there is a match somewhere in the payload, and in that case $\mathbf{S}_P$ returns 1 and 0 otherwise. Recall that $\mathbf{S}_R$ is already encoded as constraints for $\{f_a(\ell)\}_{\ell=0,\ldots,L-1}$. $\mathbf{S}_P$, then, is $\texttt{NOT-EQUALS-ZERO}(f_a(0) \vee f_a(1) \vee \cdots \vee f_a(L_T - 1))$. Assuming no overflow (in which case $\vee$ is free; see §4.4), the overhead of $\mathbf{S}_P$ (beyond $\mathbf{S}_R$) is two constraints total, stemming from `NOT-EQUALS-ZERO` (§4.4)). If overflow is possible, the Zombie compiler handles that as in Section 4.4.

Context and proximity. In the context of 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).

Perhaps surprisingly, Zombie can handle such context-dependent policies with very little overhead beyond the cost of simply matching the individual regular expressions. 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 $f_{r_1}^d$, which for each position ℓ holds a Boolean indicating whether there is a match within d characters of ℓ. Concretely,

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

Because each $f_{r_1}^d[\ell]$ is a linear combination of existing variables, there is no cost in constraints to produce it. Second, $\mathbf{S}_P$ checks whether the entrywise product of $f_{r_1}^d$ 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.

5 Implementation

Our implementation of Zombie has two main components: a client and a middlebox (MB). We currently support two classes of applications. The first is DNS filtering [45] (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. Currently, our implementation supports policy scans for text files, not more complex formats like PDF.

5.1 ZKP implementation

Circuits. The circuits used for Zombie's ZKPs were specified in the ZoKrates domain-specific language (DSL) [31] and compiled to R1CS using CirC [81], a ZKP compiler framework. The circuits comprise 1832 hand-written lines of ZoKrates code and 630 lines automatically generated by our own regexp compiler. The handwritten code was optimized and features a large improvement in the encoding of $\mathbf{S}_F$, resulting from replacing RAM with a barrel shifter over bytes of the plaintext.

The regexp compiler takes as input a policy specified as a list of regular expressions and a list of pairs of indices to generate proximity constraints, and it outputs ZoKrates code realizing this high-level policy, which can be integrated with the rest of Zombie, benchmarked in isolation, or integrated with other projects. This compiler uses a built-in understanding of the constraint-level costs of ZoKrates' semantics and performs several optimization passes to minimize the number of constraints accordingly. These passes apply 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 important improvements to CirC and the Spartan implementation. First, we created an adapter that integrates CirC with Spartan. We have also configured Spartan to use curve25519 [9] as its underlying cryptographic group, a standard choice believed to offer ≈128 bits of security.

Our CirC improvements focused on making witness generation more efficient. This is important, because CirC needs to generate a satisfying assignment for the R1CS instance before Spartan can begin proving. In early experiments, this witness generation was in fact slower than proof generation. We modified internal CirC data structures to prevent unnecessary memory copying, which greatly improved performance.

Our Spartan improvements focused on rewriting both the 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 [42]) over TLS (DoT [49]) or HTTPS (DoH [48]). 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 same session for as long as the recursive resolver will allow (up to five minutes in our testing). The client generates proofs via CirC's interface to Spartan, described in the preceding section. It sends proofs and forwards traffic to the middlebox; we ensure this via routing tables. For the application of arbitrary policies, the client acts as a workload generator and hasn't yet been integrated into an actual application like a browser.

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 proofs when the proxy is idle. As described in Figure 2, there are two free parameters for the client to choose: the number of pads to generate (m) and the length of the pad (ℓ). In our implementation, each time the child process is triggered, it generates $m = 16$ pads with proofs; a higher m value would be less likely to be exhausted by a burst of traffic, but risks performing excess precomputation that may not actually be used. Choosing ℓ depends on expected packet sizes. For the DNS application, packets are fairly short, so our implementation sets $\ell = 255$.

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 the Linux netfilter-queue library. The Zom-

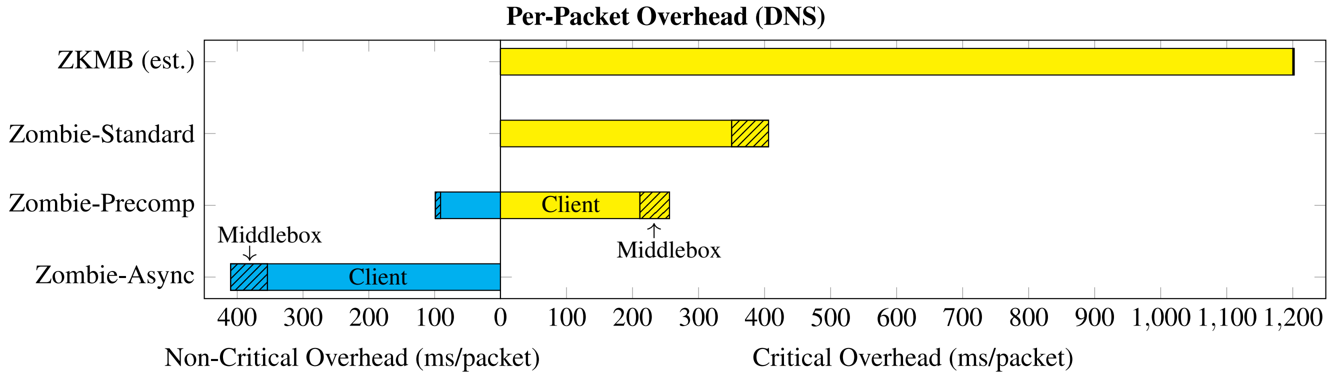


Figure 3: Per-Packet critical and non-critical overheads for enforcing a blocklist on DNS requests over TLS in zero-knowledge with ZKMB [45] and with Zombie-Standard, Zombie-Precomputation, and Zombie-Asynchronous configurations.

bie MB takes packets off this queue and performs the following steps. First, it determines whether they are policy-relevant or not. If so, the MB 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 MB 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 of Zombie?

(2) What are the individual performance contributions of the techniques in Zombie (§3.1–§3.3)?

(3) How close does Zombie's regexp framework (§4) bring real-world zero-knowledge DLP applications to practicality?

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 [45]. 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 [45, 66] (with 2 million domains). The DLP benchmarks are policies from Microsoft DLP Purview [68] that are expressed using combinations of regular expressions. Additionally, we quantify the impact of precomputation (§3.1), asynchronous mode (§3.2), batching (§3.3), and our regexp encoding (§4) on Zombie's performance.

Our experiments that require networking run on CloudLab [30] 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. Our benchmarks that run on AWS use similarly powerful instances.

We measure performance by taking the average of multiple runs unless otherwise specified.

6.1 Computational overhead and delay

We begin by measuring the overhead of Zombie in different configurations. We use the DNS filtering benchmark. The workload generator sends a DNS request every five seconds.

For the ZKMB baseline (which we call "ZKMB (est.)"), we provide conservative estimates of the prior work extrapolated from microbenchmarks we ran on the same hardware.

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 don't include the combination of precomputation and asynchrony. Asynchrony already moves all overheads to the non-critical path, so precomputation would have no effect in this case. Batching can be applied to all of the above, so we benchmark it independently (below).

Figure 3 depicts the additional latency incurred by ZKMB [45] and by Zombie. As expected, *Zombie-Precomputation* moves a non-negligible portion of the critical overhead to the non-critical path while *Zombie-Asynchronous* successfully moves all of the critical overhead to the non-critical path. All three Zombie configurations show similar overheads, with Zombie-Precomputation being slightly faster. This is expected because, as described in Section 3.1, the client batches 16 *pad-commit* proofs for faster verification, and at the same time, as noted in Section 5, generating multiple proofs together achieves higher parallelism, reducing the average proving cost per DNS request. Overall, both Zombie-Precomputation and Zombie-Asynchronous significantly reduce the latency observed by the user when compared to Zombie-Standard and ZKMB [45].

The average additional latency is about 400 ms for Zom-

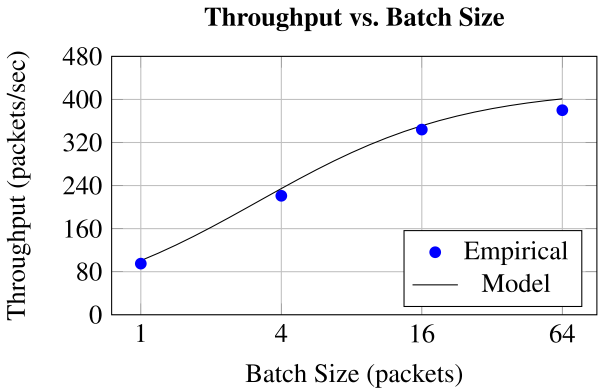


Figure 4: Middlebox throughput vs batch size for *Zombie-Async-batch*

bie with no optimizations. Of that, proof generation time is approximately 350 ms and verification time is about 50 ms. With precomputation, the average response time is lower, about 250 ms. While this is much better than the estimated performance of prior work, it is still much worse than the average latency for a DNS request, which is about 20 ms [83]. Thus, Zombie in the synchronous configuration has about an order of magnitude latency increase over a setting that doesn't enforce policy.

When in asynchronous mode, Zombie introduces no additional latency. This suggests that Zombie-Asynchronous is plausibly practical. Of course, asynchronous mode brings the requirement to buffer packets at the middlebox; we cover that when evaluating batching, as the effects on storage are more pronounced under batching.

A notable weakness of Zombie is the communication overhead of sending proofs for packets. In the synchronous setting with precomputation, each online proof is approximately 30 KB. Though these proofs are not large when compared to the average website size of 2–3 MB [4], they are 12–20× larger than the 255-byte DNS packets themselves. Because these proofs are transmitted only from client to middlebox, which are typically on the same local network, in a setting where most packets do not need proofs (for example, only DNS requests), the overall increase in required bandwidth is expected to be small. We leave a thorough treatment of this question to future work.

Middlebox resources. We now investigate middlebox resource requirements and to what degree batching (§3.3) eases these requirements. We investigate the effect of this technique on the throughput (number of (proof, packet) pairs processed per second) and the storage requirements of the middlebox.

To investigate this technique, we run a new *Zombie-Async-batch* configuration, which is parameterized by batch size. We model each client as a Poisson process, and for a fixed batch size, we vary offered load until we reach a target offered load. To reach this target offered load, we scale the parameter of the Poisson process 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.

Figure 4 depicts both a theoretical model of maximum capacity and the empirical maximum capacity for various batch sizes. For the model, we measured the time to verify a single Zombie DNS proof on one thread on CloudLab and decomposed this time into a marginal cost of 38 ms per proof and a fixed cost of 121 ms. When running on multiple cores, the performance scales almost perfectly with respect to these parameters.

When we used this model to predict the maximum throughput of the batch verifier for different batch sizes and compared it to the actual achievable throughput, we see experimentally that the divergence between the model and the actual throughput is around 5%. This discrepancy owes to lower-order middlebox costs related to packet forwarding, listening, and proof parsing. The maximum throughput we observe is 380 packets per second, at a batch size of 64.

This is too low for practicality if every packet needs to be verified, but for the DNS filtering case, we believe the implied overhead (roughly 38 ms per amortized verification on a single core measured on CloudLab) is tolerable, because DNS packets are a small fraction of all traffic. While waiting to batch 64 packets might be suitable for the asynchronous setting, it is less practical for synchronous clients. Despite this, we see that even a modest batch size of 4 offers a substantial improvement in throughput over no batching.

Batching increases throughput, but this comes at the cost of additional storage requirements in the middlebox. This is because the middlebox needs to accumulate ciphertexts in order to batch-validate proofs about them later. Despite this, storage capacity is not a limiting factor in this context due to both the small size of DNS packets and the observed capacity 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 [80].

6.2 Regular expressions

We use Microsoft Purview [68] as a source of real-world policy information in addition to several standard regexp benchmarks for more general comparisons. We implement five policies designed to detect sensitive information in the US locale. These are: bank account number [69], driver's license

Benchmark	Payload Size	Constraints	Prover Time	Verifier Time
DNS Blocklist [66], Isolated	255 B	40295	190 ms	33 ms
DNS Blocklist [66], Zombie	255 B	128702	345 ms	44 ms
Date [62], Isolated	100 B	2117	40 ms	18 ms
	2000 B	43917	122 ms	32 ms
Email [62], Isolated	100 B	489	27 ms	14 ms
	2000 B	9989	62 ms	25 ms
URI [62], Isolated	100 B	5019	46 ms	20 ms
	2000 B	105719	207 ms	38 ms
URI | Email [62], Isolated	100 B	5493	50 ms	21 ms
	2000 B	117593	255 ms	44 ms
Microsoft DLP [68], Isolated	100 B	20080	114 ms	26 ms
	2000 B	490966	5263 ms	357 ms
Microsoft DLP [68], Zombie	100 B	64438	193 ms	33 ms
	2000 B	1186241	6658 ms	453 ms

Figure 5: Constraints, prover time, and verifier time, for Zombie DNS, Zombie DLP and Isolated DNS, DLP, and pure regular expression circuits with various payload sizes.

Optimizations	Cost	Prover Time	Verifier Time
Baseline	*1566 / B*	*18763 ms*	*1420 ms*
+ `STRING-MATCH`	*996 / B*	*11417 ms*	*857 ms*
+ Alt Arithmetization	*901 / B*	*10192 ms*	*764 ms*
+ Regexp Preprocessing	*873 / B*	*9831 ms*	*736 ms*
+ Additional Tapes	*288 / B*	*2292 ms*	*159 ms*
+ Optimized Classes	242 / B	1705 ms	38 ms

Figure 6: Approximate Constraints per Byte, Prover Time, and Verifier Time for our combined Microsoft DLP policy with various levels of optimization in the order they are introduced in Section 4. The final line is the result of running CirC on the zok file produced from all optimizations. The other lines are estimates from the compiler with all prior optimizations enabled and all subsequent optimizations still disabled. Prover and Verifier Time come from applying the policy to a 1 KB payload.

number [70], taxpayer number (ITIN) [71], social security number [72], and passport number [73]. These policies individually combine substring matches, regular expressions, and proximity checks. We combine these policies so that a message must pass all five for a proof to be produced; for brevity, we refer to this combination simply as our benchmark Microsoft DLP, and we benchmark the overhead of enforcing this combination of policies in zero knowledge.

We encode this policy using Zombie's regexp pipeline (§4–§5) and benchmark it on HTTP POST messages of varying sizes. We run these experiments once since there is little experimental variation across trials. Specifically, we perform macro-benchmarks to compare it to the balance of Zombie (Figure 5) and micro-benchmarks of the impact of our regexp optimizations in isolation (Figure 6). This setup approximates a DLP setting where the network administrator wants assurance that US PII is not being uploaded to a cloud storage service.

Figure 5 allows one to directly compare the overheads of enforcing the Microsoft DLP policy to the DNS policy and to non-policy overheads in Zombie. The DLP policy is cheaper than the DNS policy and the non-policy overheads on a per-byte basis, owing to the optimizations described in Section 4. The complexity of Zombie for DLP (constraint count, prover time, and verifier time) scales roughly linearly with the size of the message. Despite being competitive with the DNS policy and non-policy overheads, the DLP use case is still not quite practical given the larger sizes of HTTP POST messages, when compared with DNS.

Figure 6 depicts the results of our regular expression optimizations on the per-byte overheads of enforcing the policy. The most substantive improvements come from our `STRING-MATCH` primitive (§4.2) and from creating multiple tapes (§4.6). The combination of our regular expression optimizations takes the cost of enforcing the policy from completely dominating the rest of Zombie's DLP circuit to something competitive with (and in some cases smaller than) the overheads not related to policy.

7 Related work

Systems built using probabilistic proofs. Probabilistic proofs are a foundational concept in complexity theory with a deep and rich literature [5,6,8,38,39]; for a survey, we recommend Goldreich [37]. 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 [113] or Thaler [100].

Zombie is built on several strands of probabilistic proof work. As noted earlier, its back-end (§2) is Spartan [90], and its front-end (§2) is CirC [81] with the ZoKrates language [31]. For comparison, the earlier ZKMBs work used the Groth16 [44] back-end and xJsnark [54] to compile circuits. While Spartan has a more expensive verifier, proof computation is faster, and there is no trusted setup, only transparent setup (§2).

Zombie is part of a growing line of work applying probabilistic proofs to solve practical problems, such as privacy-preserving payments [25, 88] private smart contracts [12, 16, 53], proofs of solvency [3, 24], verifiable delay functions [11, 51], proofs of software vulnerability [23] or cryptographic transparency logs [18, 103, 104]. Of particular relevance to our work are DECO [119] and Reclaim [97], which employ probabilistic proofs about TLS plaintext. However, both systems aim to prove some statement 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. One of the contributions of Zombie is a portfolio of techniques for encoding regular expressions in probabilistic proofs. We are aware of only two other works, both concurrent with Zombie, that address this problem; like Zombie, both target network security applications.

Exciting work by Luo et al. [65] transforms a regular expression to a Thompson NFA [101], like Zombie does (§4.3). Unlike Zombie, Luo et al. transform the NFA to a Boolean circuit and then use MPC-in-the-head [26, 50]. 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 arithmetic over large finite fields, our techniques are not applicable to their system and their Boolean circuit techniques are not applicable to our system. 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` [86], 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 [17] 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 automatons (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 systems proposed novel middlebox architectures which aim to enforce policies on encrypted traffic. For helpful surveys, we refer the reader to Sherry [96] and Naylor et al. [75]. Work prior to ZKMB [45] largely falls into two broad categories:

Trusted hardware. ETTM [28] 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 [40] refined this vision using the then-emerging *trusted execution environment* (TEE) abstraction, built using Intel's SGX implementation. An obvious limitation is that all users must have a TEE to take advantage of this approach.

mbTLS [75] 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 [47, 84, 102]. All of these works (and many others [29, 41, 46, 56, 114]) inherently rely on trusted hardware; however, we wish to avoid trusted hardware, given the growing cavalcade of exploits demonstrated against real-world TEE implementations [33, 43, 74, 77, 89, 99, 105–108]

TLS modifications. Early proposals to reconcile widespread TLS adoption with network policy enforcement envisioned 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. An example is "multi-context TLS" or mcTLS [76], 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 which still sacrifices user privacy considerably; in the DNS example users fully give up privacy of their query history. It also requires server-side changes.

Blindbox [95] 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 [32, 52, 58–61, 63, 78, 79, 85, 117, 118].

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.

A Details of Spartan Batch

A.1 Protocol details

Recall from Section 2 that Spartan works by transforming the statement about the validity of $(\mathbf{X},\mathbf{Y})$ for $\mathbf{S}$ into a statement about an associated polynomial being zero. The associated polynomial contains polynomial encodings $\tilde{A},\tilde{B},\tilde{C}$ of the R1CS matrices A,B,C. In the last part of the subprotocol that convinces the verifier this polynomial is zero, the verifier must compute three expensive polynomial evaluations: $\tilde{A}(r_x,r_y)$, $\tilde{B}(r_x,r_y)$, and $\tilde{C}(r_x,r_y)$, each at random points $r_x,r_y \in \mathbb{F}^{\log n}$ chosen by hashing prefixes of the proof as the prover generates it (that is, via the Fiat-Shamir transform [34]). We observe that, since these expensive operations 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 [92, 109].

A.2 Security proof

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 with a verifier party providing randomness whenever required. We can remove interactivity while retaining all desired properties proven below by using the standard Fiat-Shamir heuristic. Recall that (interactive) SpartanBatch is identical to 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 the random values that determine 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 Verifier provides a single random value that is taken to be the response to all parallel instances.
- All other steps are **uncoordinated**. Here, the verifier provides a b-tuple of responses, one for each parallel proof.

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 $X^* = \{x^*, x_2, \dots, x_b\}$ 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 $r_2,\dots,r_b$ and forwards $(r,r_2,\dots,r_b)$.

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 [112].

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 each i, 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.

Acknowledgments

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.

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

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

[5] 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), 1998.

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

[7] László Babai and Lance Fortnow. Arithmetization: A new method in structural complexity theory. *Computational Complexity*, 1:41–66, 03 1991.

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

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

[10] 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.

[11] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable eelay functions. In *CRYPTO*, 2018.

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

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

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

[15] 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.

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

[17] Matteo Campanelli, Dario Fiore, and Anaïs Querol. LegoSNARK: Modular design and composition of succinct zero-knowledge proofs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2075–2092, New York, NY, USA, 2019. Association for Computing Machinery.

[18] 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.

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

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

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

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

[23] 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.

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

[25] 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.

[26] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, and Titouan Tanguy. Limbo: Efficient zero-knowledge MPCitH-based arguments. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 3022–3036, New York, NY, USA, 2021. Association for Computing Machinery.

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

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

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

[30] 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.

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

[32] 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.

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

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

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

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

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

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

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

[40] 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.

[41] 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 *2019 IEEE ICDCS*, 2019.

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

[43] 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.

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

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

[46] 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.

[47] 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.

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

[49] 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.

[50] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '07, page 21–30, New York, NY, USA, 2007. Association for Computing Machinery.

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

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

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

[54] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. xJsnark: a framework for efficient verifiable computation. In *IEEE Security & Privacy*, 2018.

[55] 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.

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

[57] SSL Labs. Ssl pulse. https://www.ssllabs.com/ssl-pulse/.

[58] 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 *ISOC NDSS*, 2021.

[59] 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.

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

[61] Hyunwoo Lee, Zach Smith, Junghwan Lim, Gyeongjae Choi, Selin Chun, Taejoong Chung, and Ted Taekyoung Kwon. maTLS: How to make TLS middlebox-aware? In *ISOC NDSS*, 2019.

[62] Heng Li. Benchmark of regex libraries. https://lh3lh3.users.sourceforge.net/reb.shtml.

[63] 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.

[64] 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.

[65] 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.

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

[67] François Michel, Martino Trevisan, Danilo Giordano, and Olivier Bonaventure. A First Look at Starlink Performance. In *IMC*, 2022.

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

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

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

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

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

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

[74] 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 Security & Privacy*, 2020.

[75] 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.

[76] 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. *ACM SIGCOMM Computer Communication Review*, 45(4), 2015.

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

[78] 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.

[79] 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.

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

[81] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. CirC: compiler infrastructure for proof systems, software verification, and more. In *IEEE S&P*, 2022.

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

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

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

[85] 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.

[86] 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.

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

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

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

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

[91] 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.

[92] Srinath Setty, Richard McPherson, Andrew Blumberg, and Michael Walfish. Making argument systems for outsourced computation practical (sometimes). 01 2012.

[93] 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.

[94] Adi Shamir. Ip = pspace. *J. ACM*, 39(4):869–877, oct 1992.

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

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

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

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

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

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

[101] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, jun 1968.

[102] 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.

[103] 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.

[104] Ioanna Tzialla, Abhiram Kothapalli, Bryan Parno, and Srinath Setty. Transparency Dictionaries with Succinct Proofs of Correct Operation. In *NDSS*, 2022.

[105] 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.

[106] 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 Security & Privacy*, 2020.

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

[108] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. In *S&P*, 2021.

[109] Victor Vu, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. A hybrid architecture for interactive verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 223–237, 2013.

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

[111] Riad S. Wahby, Srinath Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *ISOC NDSS*, 2015.

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

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

[114] 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.

[115] 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 *Symposium on Networked Systems Design and Implementation*, 2019.

[116] 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.

[117] Florian Wilkens, Steffen Haas, Johanna Amann, and Mathias Fischer. Passive, transparent, and selective TLS decryption for network security monitoring. *arXiv preprint arXiv:2104.09828*, 2021.

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

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

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