LibFTE: A Toolkit for Constructing Practical, Format-Abiding Encryption Schemes

Daniel Luchaup, *University of Wisconsin—Madison;* Kevin P. Dyer, *Portland State University;*
Somesh Jha and Thomas Ristenpart, *University of Wisconsin—Madison;*
Thomas Shrimpton, *Portland State University*

https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/luchaup

This paper is included in the Proceedings of the
23rd USENIX Security Symposium.

August 20–22, 2014 • San Diego, CA

ISBN 978-1-931971-15-7

Open access to the Proceedings of
the 23rd USENIX Security Symposium
is sponsored by USENIX

LibFTE: A Toolkit for Constructing Practical, Format-Abiding Encryption Schemes

Daniel Luchaup[1] *luchaup@cs.wisc.edu*
Kevin P. Dyer[2] *kdyer@cs.pdx.edu*
Somesh Jha[1] *jha@cs.wisc.edu*
Thomas Ristenpart[1] *rist@cs.wisc.edu*
Thomas Shrimpton[2] *teshrim@cs.pdx.edu*

[1]*Department of Computer Sciences, University of Wisconsin-Madison*
[2]*Department of Computer Science, Portland State University*

Abstract

Encryption schemes where the ciphertext must abide by a specified format have diverse applications, ranging from in-place encryption in databases to per-message encryption of network traffic for censorship circumvention. Despite this, a unifying framework for deploying such encryption schemes has not been developed. One consequence of this is that current schemes are ad-hoc; another is a requirement for expert knowledge that can disuade one from using encryption at all.

We present a general-purpose library (called libfte) that aids engineers in the development and deployment of format-preserving encryption (FPE) and format-transforming encryption (FTE) schemes. It incorporates a new algorithmic approach for performing FPE/FTE using the nondeterministic finite-state automata (NFA) representation of a regular expression when specifying formats. This approach was previously considered unworkable, and our approach closes this open problem. We evaluate libfte and show that, compared to other encryption solutions, it introduces negligible latency overhead, and can *decrease* diskspace usage by as much as 62.5% when used for simultaneous encryption and compression in a PostgreSQL database (both relative to conventional encryption mechanisms). In the censorship circumvention setting we show that, using regular-expression formats lifted from the Snort IDS, libfte can reduce client/server memory requirements by as much as 30%.

1 Introduction

Both in practice and in the academic literature, we see an increasing number of applications demanding encryption schemes whose ciphertexts abide by specific formatting requirements. A small industry has emerged around the need for in-place encryption of credit-card numbers, and other personal and financial data. In the case of credit-card numbers, this means taking in a string of 16 decimal digits as plaintext and returning a string of 16 decimal digits as ciphertext. This is an example of *format-preserving encryption* (FPE). NIST is now considering proposals for standardized FPE schemes, such as the FFX mode-of-operation [7], which is already used in some commercial settings [3]. On a totally different front, a recent paper [11] builds a *format-transforming encryption* scheme. It takes in plaintext bit strings (formatted or not) and returns ciphertexts formatted to be indistinguishable, from the point of view of several state-of-the-art network monitoring tools, from real HTTP, SMTP, SMB or other network protocol messages. This FTE scheme is now part of the Tor Project's Browser Bundle, and is being integrated into other anti-censorship systems.

It seems clear that FPE and FTE have great potential for other applications, too. Unfortunately, developers will find a daunting collection of design choices and engineering challenges when they try to use existing FPE or FTE schemes in new applications, or to instantiate entirely new schemes. To begin with, there isn't a standard way to specify the formats that plaintexts or ciphertexts must respect. There are no established guidelines, and certainly no automated tools, to help developers understand whether they should be targeting deterministic schemes or randomized ones, or how their chosen formats might affect runtime performance and memory usage. (In the case of FTE, it can be difficult to tell if a given input and output format will result in a scheme that operates properly.) There are no established APIs, and no reference implementations or open-source libraries to aid development.

Making FPE/FTE More Approachable: libfte. In this work, we offer a unifying framework for building and deploying FPE and FTE schemes. We design and implement an algorithm library, libfte, and include in it developer-assistance tools. A paramount goal of

our effort is *ease-of-use*: our library exposes an interface in which formats for plaintexts and ciphertexts are easily specified via Perl-compliant regular expressions (regexes), and it relieves the programmer of the burdens of making good algorithm and parameter choices.

Some of what we do is to make existing algorithms (e.g., FFX) significantly easier to use. But some of the engineering and deployment challenges demand entirely new approaches to both FPE and FTE. Perhaps most notably, we solve an open problem regarding how to build regular-expression-based schemes using a regex's nondeterministic finite automaton (NFA) representation, as opposed to its DFA representation. This is desirable because it can lead to significantly more space-efficient schemes, but the approach was previously thought to be unworkable [5, 11]. We dispel this thought, and experimentally observe the resulting boost in efficiency.

To summarize the main contributions of this work, we:

- *Design and implement a library and toolkit* to make development and deployment easy. The libfte library exposes simple interfaces for performing FPE/FTE over regex formats specified by the user. We provide a configuration tool that guides developers towards good choices for the algorithms that will instantiate the scheme, and that provides concrete feedback on expected offline and online performance and memory usage.
- *Develop new FTE schemes* that take regular-expression formats, but can work directly with their NFA representation. This was previously thought to be an unworkable approach [5], due to a PSPACE-hardness result, but we show how to side-step this via a new encoding primitive called *relaxed ranking*. The result is FTE schemes that handle a larger class of regexes, and impose smaller offline/online memory requirements.
- *Detail a general, theoretical framework* that captures existing FPE/FTE schemes as special cases, and surfaces potentially useful new constructions, e.g., deterministic FTE that encrypts and compresses simultaneously. Due to space constraints, the formalisms appear mostly in the full version [16].

In addition, the libfte library will be made publicly available as free and open-source software[1], with APIs for Python, C++ and JavaScript.

Applications. We exercise libfte by applying it to a variety of application settings. Table 1 gives a summary of the diversity of formats required across these various applications.

We first show how to use libfte to perform FPE of SQL database fields, a classic motivational setting for FPE, but one that has (to the best of our knowledge) never been reported upon. We show that performance loss compared to conventional encryption is negligible. We also show how to leverage the flexibility of libfte to *improve* performance, by using a (deterministic) FTE scheme that simultaneously encrypts and compresses fields (in a provably secure manner).

We then use libfte to build a proof-of-concept browser plugin that encrypts form data on websites such as Yahoo! Contacts. This uses a variety of FPE and FTE schemes, and allows one to abide by a variety of format restriction checks performed by the website.

Finally, we show that our NFA-based algorithms in libfte enable significant memory savings, specifically for the case of using FTE in the network-monitor-avoidance setting [11]. Using a corpus of 3,458 regular expressions from the Snort monitor we show that we can reduce memory consumption of this FTE application by 30%.

Deployment Setting	Examples	
	Type	Constraint
Databases	credit card number	16-digit string
	datefield	YYYYMMDD
	account balance	32-bit integers
Web Forms	email address	contains @ symbol, ends with {.com,...}
	year, month, day	YYYY, MM, DD
	URL	starts with http(s)
Network Monitors	HTTP GET request	"GET /..."
	Browser X	"...User-Agent: X ..."
	SSH traffic	"SSH-..."

Table 1: Example deployment settings and constraints for FPE/FTE schemes.

2 Previous Approaches and Challenges

We review in more detail some of the main results in the areas of format-preserving and format-transforming encryption, and then discuss some of the challenges presented when one attempts to implement and use these in practice. As we shall see, existing tools fall short for the types of applications we target. Table 2 provides a summary.

Format-preserving encryption. In many settings the format of a plaintext and its encryption must be the same, and the tool used to achieve this is format-preserving encryption (FPE). Work on FPE predates its name, with various special cases such as length-preserving encryption of bit strings for disk-sector encryption (c.f., [14, 15]), ciphers for integral sets [8], and elastic block ciphers [10] including de novo constructions such as the hasty pudding cipher [21]. For an overview of work on

[1] https://libfte.org/

Paper	Builds	Formats	Schemes	Implementation	Comments
[7]	FPE	slice of Σ^*	deterministic	none	proposed NIST standard
[5]	FPE	slice of chosen regular language	deterministic	none	first FPE paper, theory only, requires regex-to-DFA conversion
[11]	FTE	slice of chosen regular language	randomized	open source, but domain specific	input format fixed as bitstrings, control of output format, requires regex-to-DFA conversion
This Work	**FPE/ FTE**	**range-slice of chosen regular language**	**deterministic/ randomized**	**open source, configuration toolchain, non domain specific**	**control of input and output format, NFA and DFA ranking, regex-to-DFA conversion not required**

Table 2: Analysis of prior works, and a comparison of features.

FPE, see Rogaway [20].

FPE was first given a formal cryptographic treatment by Bellare, Ristenpart, Rogaway and Spies (BRRS) [5]. In their work, BRRS suggested an approach to FPE called the "rank-encipher-unrank" construction. First, they show how to build a cipher that maps $\mathbb{Z}_N$ to $\mathbb{Z}_N$, for an arbitrary fixed number N. (Recall that $\mathbb{Z}_N = \{0, 1, \ldots N-1\}$.) Now say that X is a set of strings that all fit some specified format, and one desires an encryption scheme mapping X to X. A classic algorithm due to Goldberg and Sipser (GS) [12] shows that, given a DFA for X, there exists an efficiently computable function $\mathsf{rank} : X \to \mathbb{Z}_N$, where $|X| = N$ and $\mathsf{rank}(x)$ is defined to be its position (its "rank") in the shortlex ordering of X. In addition, rank has an efficiently computable inverse $\mathsf{unrank} : \mathbb{Z}_N \to X$, so that $\mathsf{unrank}_L(i)$ is the i-th string in the ordering of L. Then to encrypt a string $x \in X$: (1) rank the input x to yield a number $a \leftarrow \mathsf{rank}(x)$, (2) encipher a, giving a new number b, then (3) unrank b to yield the ciphertext $y \leftarrow \mathsf{unrank}(b)$, which is an element of X.

BRRS focus on FPE for sets X that are a *slice* of a language L, that is $X = L \cap \Sigma^n$ for some n and where Σ is the alphabet of L. Relatedly, we define a *range-slice* of a language L as $X = L \cap (\Sigma^n \cup \Sigma^{n+1} \cup \cdots \cup \Sigma^m)$, for $n \leq m$. The latter is superior because it offers greater flexibility, although not explored by BRRS. Still, extending BRRS to an FPE scheme over the entire (regular) language is possible, by establishing a total ranking one slice at a time. The main disadvantage of the BRRS scheme is that it requires a DFA to represent the set X. For most users, this is an unnatural way to specify languages, or slices thereof.

We quickly note that the BRRS algorithm may be susceptible to timing-based side-channel attacks, since rank is not constant time. Timing information may therefore leak partial information about plaintexts. We leave to future work exploration of this potential security issue, which extends to libfte and other non-constant-time message encodings as well.

The FFX scheme. Bellare, Rogaway, and Spies [7] specify the FFX mode of operation, which is a specific kind of FPE scheme and is based on the BRRS work [5]. FFX takes a parameter $2 \leq r \leq 2^{16}$, the radix, and encrypts a plaintext $P \in L = \bigcup_\ell \{0, 1, \ldots, r-1\}^\ell$ to a ciphertext in L with $|L| = |P|$. The length ℓ ranges between a minimum value of 2 (or 10, if $r \geq 10$) and $2^{32} - 1$. For example, FFX[10] enciphers strings of decimal digits to (same length) strings of decimal digits; FFX[8] does likewise for octal strings. In addition, FFX has an extra "tweak" input, making it a length-preserving tweakable cipher, in the sense of [17]. The tweak allows FFX to support associated data.

We are aware of no public, open-source implementations of FFX, though there do exist proprietary ones [3]. Even given such an implementation, the formats supported by FFX are not as general as we might like. For example, the scheme does not support domain $\mathbb{Z}_N$ when N is not expressible as r^ℓ for the supported radices r. One can rectify this using cycle walking [8] but the burden is on developers to properly do so, hindering usability. Moreover, the user is left to determine how best to map more general formats into the set of formats that FFX supports.

Format-transforming encryption. Dyer, Coull, Ristenpart and Shrimpton (DCRS) [11] introduced the notion of format-*transforming* encryption, and gave a purpose-built scheme that mapped bitstring plaintexts to ciphertexts belonging to a specified regular language. Their FTE scheme was built to force protocol misidentification in state of the art deep-packet-inspection (DPI) systems used for Internet censorship.

The DCRS scheme is randomized, which lets it target strong privacy goals for the plaintexts (namely, semantic security [13]), and also naturally aligns with using standard encryption schemes as building blocks. The scheme itself is similar in spirit to BRRS: the plaintext bitstring is encrypted using an authenticated encryption scheme, the resulting intermediate ciphertext interpreted as a number, and this number is then unranked into the target language. Like BRRS, this scheme works on slices of a given regular language.

DCRS observe that regular expressions provide a friendlier programming interface for specifying inputs. But to use the GS scheme for ranking/unranking, they must first convert the given regular expression to an NFA and then from an NFA to a DFA. The last step often leads to a large blowup in the number of states, sometimes rendering the process completely intractable. (Examples of such regexs, and the associated NFA and DFA sizes, are given in Table 6 in Section 6.) Even when the process is tractable, the precomputed tables that DCRS and BRRS use to implement ranking require space that scales linearly in the number of states in the DFA. Many of the formats used by DCRS require several megabytes of memory; in one case, 383 MB. This is prohibitive for many applications, especially if one wants to keep several potential formats in memory.

Thus, in many instances it would be preferable to use the NFA representation of the given regex, but BRRS showed that ranking given just the NFA representation of a regular language is PSPACE-hard. Building any FPE or FTE scheme that works directly from an NFA has remained an open problem.

We also note that developers might hope for a general purpose FTE scheme, that takes arbitrary regular expressions for the input and output formats, and that can be built from existing deterministic cryptographic primitives (e.g., wideblock tweakable blockciphers) or randomized ones (e.g., authenticated encryption schemes). But actually instantiating such a scheme presents an array of algorithmic and engineering choices; in the current state of affairs, expert knowledge is required.

Summary. While a number of approaches to FPE and FTE exist, there is a gap between theory and developer-friendly tools. Implementations are non-existent, and even expert developers encounter challenges when implementing schemes from the literature, including: understanding and managing memory requirements, developing a "good" construction, or engineering the plaintext/ciphertext format pair. Finally, there exist fundamental performance roadblocks when using some classes of regular expressions. This is compounded by the fact that, a priori, it isn't obvious when a given regex will raise these roadblocks.

3 Overview of libfte

To aid adoption and usage of FPE and FTE, we developed a set of tools known collectively as libfte. At a high level, libfte has two primary components (see Figure 3): a standalone tool called the *configuration assistant*, and a library of algorithms (implemented in a mixture of Python and C/C++) that exposes an API for encryption and decryption via a number of underlying FPE/FTE schemes. Loosely, the API takes a configuration, describing what algorithms to use, and some key inputs for those algorithms, while the assistant helps developers determine good configurations. Let us start by talking about the assistant.

Configuration assistant. A *format* is a tuple $\mathcal{F} = (R, \alpha, \beta)$, where R is a regular expression, and $\alpha \leq \beta$ are numbers. A format defines a set of strings $L(\mathcal{F}) = \{\ s \in L(R) \ \big|\ \alpha \leq |s| \leq \beta\ \}$, where $L(R)$ is the set of strings matched by R. Following traditional naming conventions, we call $L(\mathcal{F})$ the language of the format. Because of its wide-spread use, in libfte the input R is specified in Perl-Compatible Regular Expression syntax. However, we note that PCRE syntax allows expressions that have no equivalent, formal regular expression. For instance, PCRE expressions using $\backslash 1, \backslash 2, ...$ (where $\backslash 1$ is a back-reference to the first capture group; see [1]) are not even context free, let alone regular. Thus, libfte accepts expressions built from a subset of the full PCRE syntax.

Our configuration assistant takes as input two formats, one describing the format of plaintext strings ($\mathcal{F}_P$), and one describing the desired format of ciphertext strings ($\mathcal{F}_C$). It also accepts some "preference parameters", for example specifying the maximum memory footprint of any scheme considered by the assistant, but these are set to some reasonable default value if not specified. It then runs a battery of tests, in an effort to determine which configurations will result in FPE/FTE scheme that abide by the user's inputs. Concretely, the assistant outputs a table listing various possible configurations (some configurations may not be possible, given the user's input), along with information pertaining to expected performance and memory usage. Given the user's preferences, the table lists the best option first. In the case that no available configuration is possible, the assistant provides information as to why, so that the user can alter their inputs and try again.

The encryption API. The algorithm library exposes an encryption API that takes as input an *encryption configuration*, which consist of a plaintext format, a ciphertext format, and a configuration identifier. The latter is a string that specifies the desired methods for performing ranking, unranking, encryption and decryption. The library performs all necessary precomputations (initialize rankers, build look-up tables, etc.) in an initialization function and returns a handle to an object that can perform encryption and decryption, according to the specified configuration. Currently, ten configurations are supported by libfte (see Section 6 for descriptions).

Roadmap. In Sections 4 and 5 we describe in detail the algorithms that result in these configurations. In Section 4 we detail a new type of ranking algorithm, what

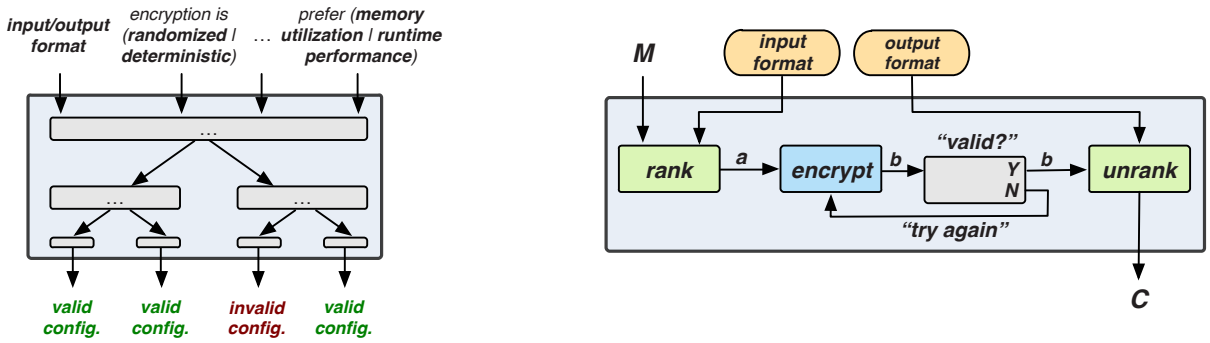

Figure 3: **Left:** The libfte configuration assistant (built against the library) helps users create formats that meet their specific performance requirements. The assistant takes an input/output format pair and uses a decision-tree process to determine if the formats are valid. If the formats are deemed valid, performance statistics are reported for the instantiated scheme(s). **Right:** The library implements APIs for FPE/FTE schemes. Shown is a diagram of the basic flow of our FPE/FTE schemes. As input it takes an input/output format and message M and returns a ciphertext C.

we call *relaxed ranking*, that allows us to work more directly with regular expressions (in particular, their equivalent NFAs), and sidestep the PSPACE-hardness obstacle. In Section 5, we lay out methods of combining relaxed ranking with standard cryptographic primitives to build both deterministic and randomized FPE and FTE schemes. For deterministic schemes, we leverage a technique called *cycle walking*, and for randomized schemes, we employ *rejection sampling*.

Then in Section 6 we describe specific instantiations of these schemes, and explain how the configuration assistant works in more detail. Finally, in Section 7 we show how these schemes can be put to work in three different use cases: database encryption, web form encryption, and network monitor circumvention.

4 Fast, Relaxed Ranking

The rank-encipher-unrank method for constructing FPE/FTE schemes needs efficient techniques for mapping strings in a regular language L to positive integers as well as computing the inverse operation (mapping positive integers back to strings in the language). Existing techniques are often impractical for two main reasons. First, the traditional DFA-based ranking requires the construction of a DFA corresponding to a regular expression. DFAs for some regular expressions can be very large. For instance, the minimum DFA for the regex $(a|b)^*a(a|b)\{20\}$ has $1 + 2^{21}$ states. Second, the numbers involved in ranking can be very large (for languages with many strings) and operations on these integers can therefore be computationally expensive. As an extreme example, ranking a $10,000$-byte long element accepted by the regex $.*$ requires numbers of up to $(2^8)^{10000}$ bits, or $10,000$ bytes. This section tackles these two challenges.

4.1 Relaxed Ranking

We introduce a framework for building FPE and FTE schemes directly from NFAs. The resulting algorithms will often use significantly less memory than the DFA approach, thus enabling general-purpose regex-based ranking in memory-constrained applications. For instance, the NFA for the regex $(a|b)^*a(a|b)\{20\}$ has 48 states.

A key insight is that we can circumvent the negative result about NFA ranking if we shift to a *relaxed ranking* approach, which we formally define in a moment. This will require, in turn, constructing FPE and FTE schemes given only relaxed ranking which we address in Section 5.

4.1.1 Relaxed Ranking Schemes

Informally, a relaxed ranking of a language $\mathcal{L}$ relaxes the requirement for a bijection from $\mathcal{L}$ to $\mathbb{Z}_{|\mathcal{L}|}$.

Formally, a *relaxed ranking scheme* for $\mathcal{L}$ is a pair of functions $\mathsf{Rank}_{\mathcal{L}}$ and $\mathsf{Unrank}_{\mathcal{L}}$, such that:

1. $\mathsf{Rank}_{\mathcal{L}} : \mathcal{L} \to \mathbb{Z}_i$ is injective, $i \geq |\mathcal{L}|$ (Note that we capitalize 'Rank' to distinguish relaxed ranking from ranking.)
2. $\mathsf{Unrank}_{\mathcal{L}} : \mathbb{Z}_i \to \mathcal{L}$ is surjective; and
3. For all $X \in \mathcal{L}$, $\mathsf{Unrank}_{\mathcal{L}}(\mathsf{Rank}_{\mathcal{L}}(X)) = X$.

The last condition means that we can correctly invert points in the image of $\mathcal{L}$, denoted $Img(\mathcal{L}) \subseteq \mathbb{Z}_i$. Note that a ranking is a relaxed ranking with $i = |\mathcal{L}|$.

DFA-based ranking revisited. As a thought experiment, one can view the traditional GS DFA-based ranking for regular languages as follows: let $\mathcal{I}$ be the set of all

accepting paths in a DFA. First, one maps a string $X \in \mathcal{L}$ to its accepting path $\pi_X \in \mathcal{I}$. Then, one maps π_X to an integer via an (exact) ranking. The composition of these two functions yields a ranking function for all strings in $\mathcal{L}$. In the DFA ranking algorithms of [5, 12], these two steps are merged.

A two-stage framework. We can use this two-step procedure to build efficient relaxed ranking algorithms. Suppose we desire to build a relaxed ranking function $\mathsf{Rank}_{\mathcal{L}}$ from a given set $\mathcal{L}$ into $\mathbb{Z}_i$. We first identify three components:

1. an *intermediate* set $\mathcal{I}$ for which we can efficiently perform ranking, i.e., there is an efficient algorithm for $\mathsf{rank}_{\mathcal{I}} : \mathcal{I} \to \mathbb{Z}_i$ where $i = |\mathcal{I}|$;
2. an injective function $\mathsf{map} : \mathcal{L} \to \mathcal{I}$; and
3. a surjective function $\mathsf{unmap}\colon \mathcal{I} \to \mathcal{L}$ such that for all $X \in \mathcal{L}$ it holds that $\mathsf{unmap}(\mathsf{map}(X)) = X$.

We then define

$$\begin{aligned} \mathsf{Rank}_{\mathcal{L}}(X) &= \mathsf{rank}_{\mathcal{I}}(\mathsf{map}(X)) \\ \mathsf{Unrank}_{\mathcal{L}}(Y) &= \mathsf{unmap}(\mathsf{unrank}_{\mathcal{I}}(Y)) \end{aligned}$$

Should unmap additionally be injective, then $\mathsf{Rank}_{\mathcal{L}}$ is a bijection, and we have (strict) ranking.

At first glance, this framework may seem to not have accomplished much as we rely on a strict ranking to realize it. But we will ensure that the language $\mathcal{I}$ allows for strict ranking, and so the framework allows us to transform the problem of ranking from a difficult set ($\mathcal{L}$) to an easier one ($\mathcal{I}$).

4.1.2 Relaxed Ranking Using NFAs

We construct relaxed ranking for NFAs using the approach above. We use as intermediate set $\mathcal{I}$ the set of all accepting paths in the NFA. To map into this set, for each string in $\mathcal{L}$ we deterministically pick an accepting path (a process called *parsing*). To rank on $\mathcal{I}$ we define a path ordering, and generalize the Goldberg-Sipser ranking algorithm for DFAs to count paths based on this ordering.

Recall that an NFA is a 5-tuple $\mathrm{M} = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is the alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation[2], $q_0 \in Q$ is the start state, and $F \subseteq Q$ is the set of final (or accepting) states. If $(q, a, q') \in \delta$ then M may transition from state q to state q' when the current input symbol is a. We also write a transition $\tau = (q, a, q') \in \delta$ as $q \xrightarrow{a} q'$, where q is the source and q' is the destination of τ.

A *path* π in M is a sequence of transitions

$$q_{i_0} \xrightarrow{a_{j_1}} q_{i_1} \xrightarrow{a_{j_2}} q_{i_2} \cdots q_{i_{n-1}} \xrightarrow{a_{j_n}} q_{i_n} .$$

[2]We assume that there are no ϵ-transitions, but this is without loss of generality as there are standard methods to efficiently remove them from an NFA.

Path π can also be expressed as a sequence of transitions $\tau_1\tau_2 \cdots \tau_n$, where $n = |\pi|$ is the length of π. The suffix π^1 of the path π is $\tau_2 \cdots \tau_n$, and we have $\pi = \tau_1\pi^1$. The sequence of characters in the path is $\pi|_\Sigma = a_{j_1}a_{j_2}...a_{j_n}$.

The intermediate set $\mathcal{I}$. An accepting path is one that ends in an accepting state. Let $\mathrm{Acc}_{\mathrm{M}}(q)$ be the set of accepting paths starting from state q. We let $\mathcal{I} = \mathrm{Acc}_{\mathrm{M}}(q_0)$.

The functions map **and** unmap**.** We must map from $\mathcal{L}$ to $\mathcal{I}$ and back. The latter is simpler: define $\mathsf{unmap}(\pi)$ to be the word $\pi|_\Sigma$. This is fast to compute, in time linear in $|w|$. The forward direction $\mathsf{map}(w)$ requires a deterministic choice for an accepting path for w. This is called *parsing*. Any suitable parsing algorithm will work, but we note that the most obvious algorithms may be quite inefficient. For example, simply recording all accepting paths while running the NFA runs in time exponential in $|w|$ in the worst case.

Linear-time parsing. We now give the (to the best of our knowledge) first algorithm for determining a compact representation of all of an NFA's accepting paths for a string w. Then $\mathsf{map}(w)$ simply runs this algorithm for w and outputs the lexicographically least accepting path. Our algorithm constructs an implicit representation of a directed-acyclic graph (DAG) representing all accepting paths for w. The lexicographically least accepting path for w can then be found using a simple traversal of the DAG. Next we describe the algorithm in detail.

Let $\mathrm{M} = (Q, \Sigma, \delta, q_0, F)$ be an NFA, $Q' \subseteq Q$, and $c \in \Sigma$. We denote by $\delta(Q', c)$ the set of states q such that $(q', c, q) \in \delta$ for some $q' \in Q'$, and by $\delta^{-1}(Q', c)$ the set of states q such that $(q, c, q') \in \delta$ for some $q' \in Q'$.

Consider a string $w = c_1c_2...c_n$. Traditional NFA matching starts with a frontier of states $\mathcal{F}_0 = \{q_0\}$, and at every position k in w it computes $\mathcal{F}_k = \delta(\mathcal{F}_{k-1}, c_k)$. The string is accepted if $\mathcal{F}_n \cap F \neq \emptyset$. However, this does not allow easy recovery of an accepting path, even if all $\mathcal{F}_k$ sets are saved. The main reason for this is that there might be states in the frontiers that do not lead to an accepting state. To work around this, we also scan the input backwards, maintaining a backwards frontier set of states where $\mathcal{B}_n = F$, and $\mathcal{B}_{k-1} = \delta^{-1}(\mathcal{B}_k, c_k)$. Given the sequences $\{\mathcal{F}_k\}$ and $\{\mathcal{B}_k\}$, with $k = 0, ..., n$, we compute $\{\mathcal{S}_k\}$ where $\mathcal{S}_k = \mathcal{F}_k \cap \mathcal{B}_k$. The set $\mathcal{S}_k$ contains all states reachable from the start state following transitions on $c_1...c_k$ such that $c_{k+1}c_{k+2}...c_n$ is an accepting path. Together, $\{\mathcal{S}_k\}$ and the NFA transitions of the form (q, c_k, q') with $q \in \mathcal{S}_{k-1} \wedge q' \in \mathcal{S}_k$, form an implicit Direct Acyclic Graph (DAG) representation of all accepting paths for w. Finally, we traverse this DAG starting from $q_0 \in \mathcal{S}_0$ and following the lexicographically smallest transitions, which yields $\mathsf{map}(w)$.

NFA path ranking. All that remains is to give a strict

ranking algorithm for $\mathcal{I}$, the set of accepting paths in the NFA. Here, we can adapt techniques from the DFA-based ranking by Goldberg and Sipser. Their algorithm can be viewed as a recursive procedure for counting the number of accepting DFA paths that precede a given path in lexicographical order.

Let $T(q, n)$ be the number of paths of length n in $\mathrm{Acc}_{\mathrm{M}}(q)$. Note that, for all $q \in Q$ and $0 \leq i \leq n$, the value of $T(q, i)$ can be computed in polynomial time using a simple dynamic-programming algorithm.

Assume that the NFA transitions are enumerated according to a total ordering, and that $\tau \lessdot \tau'$ means that τ precedes τ' according to this order. The ordering on transitions induces a lexicographical ordering '$\prec$' on paths (which are sequences of transitions). Formally, if $\pi_1 = \tau_1\pi_1^1$ and $\pi_2 = \tau_2\pi_2^1$, then this order is:

$$\pi_1 \prec \pi_2 \iff \tau_1 \lessdot \tau_2 \vee \left(\tau_1 = \tau_2 \wedge \pi_1^1 \prec \pi_2^1\right) \quad (1)$$

Let $\mathsf{rank}(\pi)$ be the number of accepting paths $\pi' \prec \pi$ that precede π in the lexicographical order on paths. It follows that, $\mathsf{rank}(\epsilon) = 0$ (the rank of the empty string is 0), and for any $\pi = \tau\pi^1 \in \mathrm{Acc}_{\mathrm{M}}(q)$, we have:

$$\mathsf{rank}(\pi) = \mathsf{rank}(\pi^1) + \sum_{(q,c',q') \lessdot \tau} T(q', n-1) \quad (2)$$

Note that the sum is over transitions $\tau' = (q, c', q') \in \delta$ that precede τ in transition order, $\tau' \lessdot \tau$. In words, we are summing over all outgoing edges from q that lead to paths that are lexicographically smaller than the paths that follow the transition τ. Unrolling the recursion gives us an iterative procedure for ranking accepting paths of length n that can be efficiently implemented via dynamic programming.

To conclude, the relaxed ranking for a string w accepted by an NFA is $\mathsf{Rank}(w) = \mathsf{rank}(\mathsf{map}(w))$, and the reverse is $\mathsf{Unrank}(r) = \mathsf{unmap}(\mathsf{unrank}(r))$.

4.2 Large Integer DFA/NFA Optimization

We present a simple but effective optimization that speeds up both NFA and DFA-based ranking. In practice, ranking efficiency depends on how fast we evaluate the sum in equation (2), and this depends on the precise definition of the transition order. We define this order so that we can replace multiple large-integer additions with a single multiplication. Our experiments confirmed that this replacement indeed resulted in faster code.

Observe that equations (1) and (2) used for path ranking depend only on transition (edge) order and structure of the automaton. This observation is valid for both NFA and DFA. Previous, traditional, DFA ranking is given by these equations and standard lexicographical ordering, using character order: $(q, c', q') \lessdot (q, c'', q'') \Longleftrightarrow (c' < c'')$. In a DFA $c' = c'' \implies q' = q''$. But equation (1) does not have to use standard lexicographical ordering.

Our idea is to give *priority to states over characters.* We assume a state and character order given by an arbitrary but fixed enumeration of Q and Σ, and use the following order for transitions originating from the same state q: $(q, c', q') \lessdot (q, c'', q'')$ if-and-only-if $(q' < q'')$ or $q' = q''$ and $c' < c''$. This specific order allows for precomputation in equation (2). In equation (2) we can replace all the terms $T(q', n-1)$ which correspond to transitions $(q, c', q') \lessdot \tau$ with $n[q, q'] \times T(q', n-1)$, where the precomputed value $n[q, q']$ represents the number of transitions from q to q'. Similarly, all the terms corresponding to edges $\tau' = (q, c', q'')$, where $\tau' \lessdot \tau = (q, c'', q'')$, can be replaced by $r[q, c'', q''] \times T(q'', n-1)$, where $r[q, c'', q'']$ is the number of such transitions. These optimizations have benefit, because the numbers $T(q, n)$ can be very large multiple precision integers.

5 Building FTE Schemes

Now we turn to building FTE schemes, treating FPE in passing as a special case of FTE. We specifically give two methods for composing relaxed-ranking algorithms with an underlying cryptographic primitive to make an FTE scheme. For deterministic FTE, the cryptographic component is a tweakable cipher (e.g. FFX), and we call the composition *cycle-walking FTE*. For randomized FTE, the cryptographic component is an authenticated encryption scheme, and we call the composition method *rejection-sampling FTE.* (Impatient readers can look ahead to Figure 4 for the pseudocode.) We delay specific instantiations of the schemes until Section 6.1.

Informal FTE scheme syntax. We provide a formal treatment of FTE scheme syntax in the full version. We provide a simpler, more informal discussion of it here; this will suffice for what follows. An FTE scheme is a pair of algorithms $(\mathsf{Enc}, \mathsf{Dec})$. The FTE encryption algorithm Enc takes as inputs

- a key K
- a pair of formats $(\mathcal{F}_P, \mathcal{F}_C)$ that describe the language $L(\mathcal{F}_P)$ of plaintext inputs, and the language $L(\mathcal{F}_C)$ of ciphertext outputs
- a plaintext string $M \in L(\mathcal{F}_P)$
- associated data, and encryption parameters (both optional)

and outputs a ciphertext string $C \in L(\mathcal{F}_C)$, or a special "failure" symbol $\perp$. Associated data is data that must be bound to the underlying plaintext, but whose privacy is not required. (For example, metadata meant to provide context for the use or provenance of the plaintext.) We allow for encryption parameters to help enforce spe-

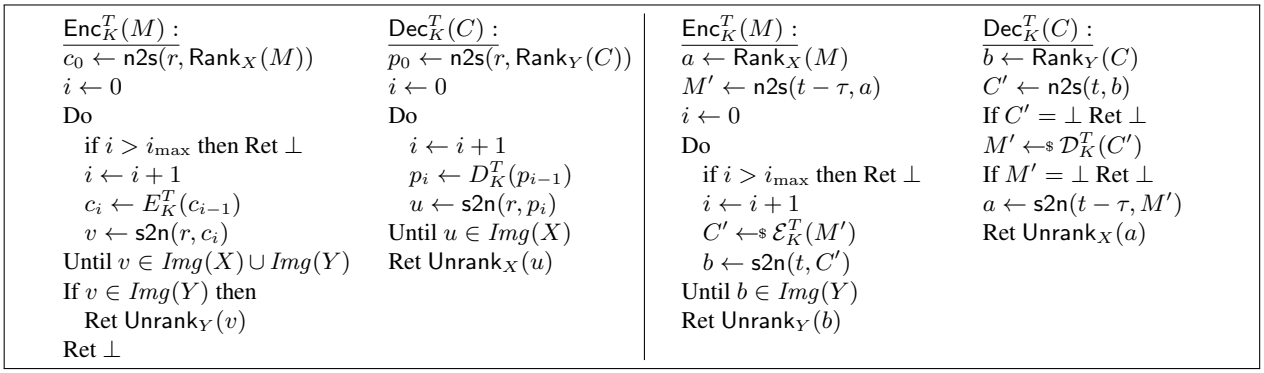

Figure 4: **Left:** Cycle-walking deterministic FTE. $\mathsf{n2s}(r, a)$ returns the string representing number a in radix r, and $\mathsf{s2n}(r, b)$ returns the number whose radix r representation is b. The parameter $i_{\max}$ determines the maximum number of iterations. **Right:** Rejection-sampling randomized FTE.

cific failure criteria, which will become clear when we describe our schemes. We write $\mathsf{Enc}_K^{T,P}(M)$ for FTE encryption of message M, under key K, using associated data T and parameters P. To ease the burden of notation (slightly), we typically do not explicitly list the parameters as inputs. The encryption algorithm may be randomized, meaning that fresh randomness is used for each encryption.

The FTE decryption algorithm Dec takes as input $(\mathcal{F}_P, \mathcal{F}_C), K$, a ciphertext C, and the associated data T (if any), and returns a plaintext M or $\perp$. The decryption algorithm is always deterministic.

Unlike conventional encryption schemes, we do not demand that $\mathsf{Enc}_K^{T,P}(M)$ always yield a valid ciphertext, or always yield $\perp$, when T, P and K are fixed. Instead, we allow encryption to "fail", with some small probability, to produce a ciphertext for a any given plaintext in its domain. Doing so will permit us to give simple and natural FTE schemes that would be ruled out otherwise.

In general, the formats can change during the lifetime of the key, even on a per-plaintext basis. (Of course, changes must be synchronized between parties.) When we talk about an FTE scheme being over some given formats, or their languages, we implicitly have in mind some notion of a format-session, during which the formats do not change.

5.1 Cycle-walking (deterministic) FTE

To build deterministic FTE schemes we take inspiration from BRRS rank-encrypt-unrank. However, accommodating format transformations and, especially, NFA-based language representations introduces new challenges.

To begin, let $X = L(\mathcal{F}_P)$ and $Y = L(\mathcal{F}_C)$. Assume that we perform relaxed ranking using the two-stage framework in Section 4.1.1, with the intermediate sets $\mathcal{I}(X)$ for X and $\mathcal{I}(Y)$ for Y. If Rank_X and Rank_Y are the corresponding relaxed-ranking functions, let $Img(X)$ be the image of X under Rank_X, and likewise $Img(Y)$ be the image of Y under Rank_Y. Define $N_X = |\mathcal{I}(X)|$ and $N_Y = |\mathcal{I}(Y)|$. (Recall that if we are using NFA-based ranking over either X or Y, these values can be significantly larger than $|X|$ or $|Y|$.) We assume that both N_X, N_Y are finite.

Say one has a tweakable cipher[3] E that natively supports strings over a variety of radices, e.g. FFX. (At a minimum, there are many constructions of secure tweakable ciphers that support radix 2, e.g. [9, 14, 15].) Now, fix integers $r \geq 2$ and $t \geq \lceil \max\{\log_r(N_X), \log_r(N_Y)\} \rceil$, so that a string of t symbols from $\{0, 1, \ldots, r-1\}$ suffices to represent the relaxed-rankings of X and Y. Then if E can encipher the set of strings $\{0, 1, \ldots, r-1\}^t$, we can encrypt a plaintext $M \in X$ as shown on the left side of Figure 4.

Cycle walking. A well-known fact about permutations is that they can be decomposed into a collection of disjoint cycles: starting from any element a in the domain of the permutation π, repeated application of π will result in a sequence of distinct values that eventually ends with a. Black and Rogaway [8] were the first to exploit this fact to build ciphers with non-standard domains, and we use it, too. For any fixed K and T, the mapping induced by E_K^T is a permutation. Thus, inside the Do-loop, the distinct strings $c_0, c_1 \leftarrow E_K^T(c_0)$, $c_2 \leftarrow E_K^T(E_K^T(c_0))$, and so on form a sequence that eventually must return to c_0. Intuitively, if we want a ciphertext that belongs to a particular subset $S \subseteq \{0, 1, \ldots, r-1\}^t$, we can walk the cycle until we hit a string $c_i \in S$.

There are, however, two important details to consider. The first is that encryption is not guaranteed to hit *any*

[3] If the FTE scheme does not need to support associated data, then the underlying cipher need not be tweakable, and references to T in the pseudocode can be dropped.

string $c_i \in S$. For example, if the subset is small, or the cycle is very short. So encryption must be equipped with test that tells it when this has happened, and $\perp$ should be returned. The second is that there must be a test that uniquely identifies the starting string c_0. This is because decryption should work by waking the cycle in reverse. Absent a test that uniquely identifies c_0, it may not be clear when the reverse cycle-walk should stop.

Our implementation deals with both of these issues. In particular, c_0 is the t-symbol string that results from relaxed-ranking our FTE plaintext input M. By definition, c_0 is a string that, when viewed as a radix-r integer, is in $Img(X)$. We desire to find a c_i that, when viewed as an integer, is in $Img(Y)$, since this is the set of integers that yield ciphertexts in Y that will be properly decrypted. Intuitively, the walk should halt on the first i for which this is true. But then, if any of $c_1, \ldots, c_{i-1}$ represent integers that are in $Img(X)$, proper decryption is not possible (because we do not know how many steps to go from c_i back to c_0). Thus our cycle-walking encryption checks, at each step, to see if the current walk should be terminated because decryption will not be possible, or because we have found a c_i that will yield a ciphertext Y that will decrypt properly. We also allow cycle-walking FTE to take a maximum-number-of-steps parameter $i_{\max}$, and encryption fails if that number of steps is exceeded.

Efficiency. The standard security assumption for a tweakable cipher is that, for any secret key K, and any associated data T, the mapping induced by E_K^T is indistinguishable from that of a random permutation. Modeling E_K^T as such, the expected number of steps before the cycle-walk terminates is at most $r^t/|Img(X) \cup Img(Y)|$ (a conservative bound) and never more than $i_{\max}$. Assuming the walk terminates before $i_{\max}$ steps, then the probability that the encryption succeeds is $p_s = |Img(Y)|/|Img(X) \cup Img(Y)|$. Since relaxed ranking is injective, $|Img(X)| = |X|$ and $|Img(Y)| = |Y|$, so $p_s \geq 1/(1 + |X|/|Y|)$. Thus we expect that p_s is quite close to 1 if $|Y| \gg |X|$.

Each step of the cycle-walk requires checking $v \in Img(X) \cup Img(Y)$, which can be done by checking $v \in Img(X)$ first (signaling termination of the walk), and then $v \in Img(Y)$ (signaling successful termination). A straightforward way to implement the last is to test if $v = \mathsf{Rank}_Y(\mathsf{Unrank}_Y(v))$ or, using our two-stage viewpoint on relaxed ranking, $\mathsf{map}(\mathsf{Unrank}(v)) = \mathsf{unrank}_{\mathcal{I}}(v)$, which may be faster. Checking $v \in Img(X)$ can be done likewise.

Recall that the NFA representation of a regex, unlike a DFA representation, may have many accepting paths for a given string in its language. This can lead to $N_X \gg |X| = Img(X)$ or $N_Y \gg |Y| = Img(Y)$, hence, potentially, $r^t \gg |Img(X) \cup Img(Y)|$. When this happens, the resulting in cycle-walking scheme may be prohibitively inefficient in some applications.

Simplifications. We note that the cycle-walking technique is used in [5], as well, but they restrict to the much simpler case that $X = Y$. More generally when we know that $Img(X) \subseteq Img(Y)$, we can simplify our construction. One may still need to cycle-walk in this case if $r^t > |Y|$. For example, say one desires to use $r = 2$ (binary strings) but the larger of $|X|, |Y|$ is not a power of two. But when $Img(X) \subseteq Img(Y)$ we know that, if the encryption cycle-walk terminates before $i_{\max}$ steps, then it always finds a point in $Img(Y)$, i.e. $p_s = 1$. Also, the expected number of steps is at most $r^t/|Img(Y)| = r^t/|Y|$, again modeling E_K^T as a random permutation. Finally, we note that the walk termination test can be simplified to $v \in Img(Y)$, and encryption can thereafter immediately return $\mathsf{Unrank}_Y(v)$.

Security. We mentioned, above, that the standard security assumption for a tweakable cipher is that, when the key K is secret, every associated data string T results in $E_K^T(\cdot)$ being indistinguishable from a random permutation. Under this assumption, it is not hard to see that the cycle-walking construction outputs (essentially) random elements of the set $Y = L(\mathcal{F}_c)$, when it does not output $\perp$. Intuitively, each $E_K^T(c_{i-1})$ in the cycle-walk is a random string (subject to permutivity), so the corresponding number v represented by the string is random, too. Thus, if $v \in Img(Y)$, it is a random element of this set, resulting in a random element of Y being chosen when v is unranked.

In the full version we formally define a security notion for deterministic FTE schemes, and give a theorem stating the security of our construction relative to this security notion.

5.2 Rejection-Sampling (randomized) FTE

We now turn our attention to building randomized FTE schemes. Let $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a conventional, randomized, authenticated-encryption scheme with support for associated data (AEAD). We assume that this scheme has a fixed ciphertext stretch τ; this is typical of in-use AEAD schemes. To build a randomized FTE scheme using a generalized ranking scheme, we use a rejection-sampling approach. Let t be the least integer such that both of the following are true: (1) $|\mathcal{I}(X)| \leq 2^{t-\tau}$, and (2) $|\mathcal{I}(Y)| \leq 2^t$. Then to encrypt $M \in X$, or decrypt $C \in Y$, under key K and associated data T, we do as shown on the right side of Figure 4.

A standard security assumption for AEAD schemes is that its ciphertexts are indistinguishable from strings (of the same length) that are uniformly random. Under this assumption, treating each C' as a random t-bit string, the

Sub-Component	Written in...	Lines of Code
Regular Expression Parser	C/C++/Flex/Bison	2,057
DFA Minimizer	C/C++	1,166
NFA/DFA Ranking	C/C++	2,752
FFX	C++	842
FPE/FTE	C++	870
Configuration Assistant	C++/Python	731

Table 5: The sub-components of libfte.

expected number of invocations of $\mathcal{E}_K^T$ is $2^t/|Img(Y)| = 2^t/|Y|$. (And certainly no more than $i_{\max}$.)

Under this standard security assumption, it is intuitive that any element of $Y = \mathcal{F}_c$ returned by our rejection-sampling FTE is a uniform one. If each C' is indistinguishable from a random string, then the corresponding number b represented by C' is random, too. Hence if $b \in Img(Y)$, then it is a random element of that set, and so the element of Y that results from unranking b will be random.

We give a formal security notion for randomized FTE, and a security theorem for rejection-sampling-based FTE, in the full version.

6 Realizing LibFTE

In Section 5 we explored strategies for constructing FPE/FTE schemes in theory. Now, let's concretely describe the schemes implemented in libfte.

Implementation. The libfte implementation is a hybrid of C, C++, Python, Flex and Bison. We present a detailed breakdown of the sub-components in Table 5. We engineered a custom regular expression parser because off-the-shelf solutions did not expose the appropriate data structures necessary to implement our NFA relaxed-ranking algorithm.

In addition to a native C++ interface, we also provide interfaces in Python and JavaScript for libfte. The Python interface is exposed through a manually-written wrapper around the C++ implementation. The JavaScript interface is provided through C++-to-JavaScript compilation.

6.1 Schemes Implemented in LibFTE

We use a shorthand notation to refer to types of libfte instantiations. As an example, T-ND-$ is a an F**T**E scheme that uses **N**FA-based ranking (Section 4.1) for the input format, and **D**FA-based ranking (Section 4.2) for the output format, and is randomized (**$**); T-ND denotes the same, but the scheme is deterministic. FPE constructions are similarly named, but begin with P.

For deterministic schemes (those without the final $) we use the cycle-walking construction, with FFX[2] as the underlying tweakable cipher. For randomized schemes, we use the rejection-sampling construction. As the underlying encryption scheme, we employ the Bellare-Rogaway "encode-then-encipher" paradigm [6], prepending the result of $\mathsf{Rank}_X(M)$ (interpreted as a fixed-length bitstring) with the appropriate number of random padding bits, and applying FFX[2] to this. Because our particular application of randomized FTE does not need support for associated data, the FFX tweak was fixed to an all-zeros string, and we do not need redundancy padding in our encode-then-encipher scheme.

We note that, although we fixed specific instantiations of FPE/FTE schemes for the sake of a concrete evaluation, there is no reason to restrict to these. In the randomized scheme, for example, one could use CTR-AES (with a random IV) and HMAC-SHA256 in an "encrypt-then-mac" composition [4, 18] (including any associated data in the mac-scope) for the underlying primitive.[4]

6.2 The LibFTE Configuration Assistant

We now turn our attention towards the implementation details of the libfte *configuration assistant*. We divide the internal workflow of the configuration assistant into three steps. First, we gather requirements from the user, this is done by the user passing parameters to a command-line interface. Then, we start with an initial set of all possible FPE/FTE schemes (i.e., P-xx, T-xx, T-xx-$) that one could instantiate, and use a decision tree algorithm to eliminate schemes from the initial set that do not satisfy user requirements. Finally, the configuration assistant analyzes the set of all schemes that were not eliminated in stage two, performs a battery of tests on them, and returns the results to the user. We provide a sample output of this tool in Figure 7.

Collecting requirements from the user. The command-line configuration assistant (see Figure 7) takes two required parameters, the input and output formats. In addition to the required parameters, the configuration assistant takes optional parameters, most notably: the memory threshold for the configuration assistant to determinize regexs, and the memory threshold for FPE/FTE scheme instantiation.

Identifying feasible schemes. Next, the configuration assistant's job is to eliminate schemes that fall outside the user-specified requirements. It starts with a set of all possible FPE/FTE schemes that one could construct (i.e.,

[4] One should keep in mind the interaction between the ciphertext length overheads of AEAD and the expected number of steps in rejection-sampling.

Scheme	Input/Output Format			DFA/NFA States	Memory Required	Encrypt (ms)	Decrypt (ms)
	R	α	β				
P-DD	`(a|b)*`	0	32	2	4KB	0.18	0.18
	`(a|b)*a(a|b){16}`	16	32	131,073	266MB	0.25	0.21
	`(a|a|b){16}(a|b)*`	16	32	18	36KB	0.19	0.18
	`(a|b){1024}`	1,024	1,024	1,026	34MB	1.2	1.2
P-NN	`(a|b)*`	0	32	3	6KB	0.36	0.35
	`(a|b)*a(a|b){16}`	16	32	36	73KB	0.61	0.60
	`(a|a|b){16}(a|b)*`	16	32	51	103KB	1,340	1,340
	`(a|b){1024}`	1,024	1,024	2,049	68MB	6.6	6.6

Table 6: Performance benchmarks for P-DD and P-NN constructions, based on our Python API. The regular expressions have been selected to highlight the strengths and weaknesses of the constructions. Recall that α and β are upper- and lowerbounds (respectively) on the length of strings used in a range slice.

```
$ ./configuration-assistant \
>  --input-format "(a|b)*a(a|b){16}" 0 32 \
>  --output-format "[0-9a-f]{16}" 0 16

==== Identifying valid schemes ====
WARNING: Memory threshold exceeded when
         building DFA for input format
VALID SCHEMES: T-ND, T-NN,
               T-ND-$, T-NN-$

==== Evaluating valid schemes  ====
SCHEME ENCRYPT DECRYPT ... MEMORY
T-ND   0.32ms  0.31ms  ... 77KB
T-NN   0.39ms  0.38ms  ... 79KB
...
```

Figure 7: A sample execution of our configuration assistant for building an FTE scheme. The tool failed to determinize the regex of the input format, and notifies the user that that T-ND, T-NN, T-ND-$ and T-NN-$ constructions are possible. Then reports on the performance of these schemes.

P-xx, T-xx, T-xx-$). If the DFA can't be built (within the user-specified memory thresholds) for the input format, then we eliminate P-Dx, T-Dx and T-Dx-$ schemes from consideration. We repeat this process for the output format. Then we perform a series of additional checks — involving the sizes of $L(\mathcal{F}_P)$, $L(\mathcal{F}_C)$, the sizes of the intermediate representations, the minimum ciphertext stretch of underlying cryptographic primitives, etc. — to cull away schemes that should not be considered further.

Evaluating feasible schemes. Finally, we consider the set of schemes that remain from the previous step. If there are none, we output an error. Otherwise, we iterate through the set of schemes and perform a series of functional and performance tests against them. We have fourteen quantitative tests, such as: the average time elapsed to perform encryption/decryption, the (estimated) probability that encryption returns $\perp$, and memory requirements. The final result of the tool is a table output to the user, each row of the table represents one scheme and the columns are results from the quantitative tests performed. The method for sorting (i.e., prefer memory utilization, prefer runtime performance, etc.) is a user-configurable parameter.

6.3 Performance

We conclude this section with benchmarks of our libfte implementation. All benchmarks were performed on Ubuntu 12.04, with an Intel Xeon E3-1220 at 3.10GHz and 32GB of memory. Numbers reported are an average over 1,000 trials for calls to our libfte Python API. For memory utilization of each scheme, we measure the memory required at runtime to use the specified formats. For encrypt benchmarks we select a random string in the language of the input format and measure the time it takes for encrypt to return a ciphertext. For decrypt benchmarks we select a random plaintext, encrypt it, then measure the time it takes for decrypt to recover the plaintext.

In Table 6 we have the performance of libfte for P-DD and P-NN schemes. Note that `(a|b)*a(a|b){16}` requires roughly four orders of magnitude less memory using a P-NN scheme, compared to a P-DD scheme. With the P-NN scheme for `(a|a|b){16}(a|b)*`, the high encrypt cost is completely dominated by cycle walking, we do roughly 720 FFX[2] encrypt calls per P-NN call. (The configuration assistant would inform the user of this, and the user would have the opportunity to re-write the expression as `(a|b){16}(a|b)*`.) For `(a|b)*`, the two constructions (i.e., P-DD, P-NN) require a comparable amount of memory but the P-DD construction is twice as fast for encryption/decryption.

Due to space constraints we omit benchmarks for T-xx and T-xx-$ schemes in this section, and defer their analysis to Section 7.

7 Exploring LibFTE Use Cases

We turn our attention to integrating libfte into third-party applications. Our goal is to show that libfte is easy to use, flexible, and in some cases *improves* performance, compared to other cryptographic solutions. In this section we consider three use cases: database encryption, web form encryption, and encryption using formats lifted from a network monitoring device.

7.1 Databases

We start with integration of libfte into a PostgreSQL database. For our database we used PostgreSQL 9.1 in its default configuration. Our server was Ubuntu 12.04 with an Intel Xeon E3-1220 v3 @ 3.10GHz and 32GB of memory. We use the pgcrypto library that is included in PostgreSQL's contrib directory as our baseline cryptographic library. We performed all tests with the PostgreSQL client and server on the same machine, such that network latency did not impact our performance results.

The integration of libfte into our database as PostgreSQL stored procedures required 53 lines of plpython code.

Pgbench is tool included with PostgreSQL to measure database performance. As input, pgbench takes a description of a database transaction and runs the transaction repeatedly, emulates concurrent users, and measures statistics such as transactions per second and response latency. We used pgbench's default database schema and transaction set for testing libfte's impact on PostgreSQL performance. The default pgbench testing schema simulates a simple bank and has four tables: accounts, tellers, branches, and history. The default transaction set for testing includes three query types: SELECT (S), UPDATE (U) and INSERT (I). There are three different transaction types that can be selected using pgbench: S, USI, and USUUI — for each transaction type the acronym represents the type and order of queries executed.

In order to test the performance impact libfte has on PostgreSQL, we created four configurations for populating and accessing data in the database:

- **PSQL:** The default configuration and schema used by the pgbench utility for its simple bank application. No encryption is employed.
- **+AES:** The default schema, with the following modifications: the balance columns in accounts, tellers, and branches are changed from type `integer` to type `bytea`. To secure these fields we use AES128 in ECB mode with PKCS padding.
- **+AE:** We use the same schema as **+AES**, but we change our encryption algorithm to pgcrypto's recommended encrypt function `pgp_sym_encrypt`, which provides privacy and integrity of data.
- **+FPE:** We use the default schema, but employ a libfte P-DD scheme with the format $R \leftarrow \backslash\text{-}[0\text{-}9]\{9\}$, $(R, 9, 10)$, to encrypt account balances (in accounts, tellers, and branches) in-place.

Account Balance Queries				
Transaction Type	**Transactions Per Second**			
	PSQL	+AES	+AE	+FPE
S	38,500	30,246	8,380	8,280
USI	2,380	2,259	1,580	1,540
USUUI	99.2	97.5	97.2	96.5

Table 8: A comparison of throughput (transactions/second) for our four database configurations.

We note that in our evaluation we did not have the option to compare libfte to a scheme that provides the same functionality or security, as no prior scheme exists. We compare to **+AES** because it provides a baseline performance that we would not expect libfte to exceed. The comparison to **+AE**, which provides privacy and integrity, can be used as a baseline for the performance of a cryptographic primitive implementation in a widely-used, mature database product.

Performance For each configuration and transaction type we executed pgbench for five minutes with 50 active customers, leaving all other pgbench parameters as default. In all configurations except PSQL, when performing modifications to the database we perform an encrypt when storing the account balance. When retrieving the account balance we recover the plaintext via a call to the decryption algorithm.

In Table 8 we have the have the benchmark results for transactions per second carried out by the server for our four database configurations and three transaction types. For the most complex transaction type (USUUI) our +FPE configuration reduces the number of transactions per seconds by only 0.8%, compared to the +AE configuration. For the simplest query type (S), +FPE reduces the transactions-per-second rate by only 1.1%, compared to +AE. Compared to the +AES configuration the +AE and +FPE reduce the transactions per second by roughly 72%. This is, in fact, not surprising as under the hood the +FPE configuration relies on FFX, which in turn calls AES at least ten times.

In Table 9 we have the average latency for each of the five different query types. This measures the amount of time elapsed between a client request and server response. Compared to the +AE configuration, +FPE introduces no substantial latency.

Simultaneous encryption and compression. As one final test, we deploy a T-DD scheme in our PostgreSQL

Account Balance Queries				
	Average Latency (ms)			
Query	PSQL	+AES	+AE	+FPE
(U) accounts	0.6	1.2	2.1	2.1
(S) accounts	0.4	0.5	1.0	1.0
(U) tellers	412	412	415	420
(U) branches	78	80	80	84
(I) history	0.2	0.2	0.2	0.2

Table 9: Average latency per query for each database configuration.

Credit Card Number Queries					
	PSQL	+AES	+AE	+FPECC	+FTECC
Table Size	50MB	65MB	112MB	50MB	**42MB**
Query Avg.	74ms	92ms	112ms	125ms	110ms

Table 10: FTE for simultaneous encryption and compression. The table size is the amount of space required on disk to store the table. We also present the average query time (over 1,000 trials) for selecting (and decrypting) 100 credit card numbers at random from 1 million records.

database to provide simultaneous privacy and compression. We augment the default pgbench database schema to add a new table for credit card numbers. This table has two columns: an account number of type `integer` and a credit card number field of type `bytea`. (Following the structure of the pgbench schema, we do not add any indexes to this table.) We start with the four configurations we presented in our initial benchmarks. However, we change our +FPE configuration to a P-DD scheme that encrypts 16-digit credit card number in-place and call this +FPECC. Then we introduce a new configuration, +FTECC, a T-DD scheme where our input format is a 16-digit credit card number and our output format is the set of all 7-byte strings.

In each configuration we populated our database with 1 million random credit card numbers. For each database configuration, we have a breakdown (Table 10) of the query cost to retrieve 100 credit card numbers at random (and decrypt, if required) as well as the total size of the new credit card table. Compared to the +AES and +AE configuration, our +FTECC configuration requires 35% and 62.5% less space, respectively. We note that it may be possible to employ additional compression in the PSQL, +FPECC settings (e.g., an int to bitstring conversion). However, such optimizations are not possible in the +AES and +AE configurations.

We also highlight that, with respect to query times (Table 10) our +FTECC configuration modestly outperforms the +AE configuration. Compared to +AES, +FTECC introduces a 19.5% increase in query cost.

7.2 Web Forms

Next, we present a Firefox extension powered by libfte. The job of this browser extension is to encrypt sensitive contact information, client-side, in a Yahoo address book contact form, prior to submission to the remote Yahoo servers.

Browser extension. We tested our libfte-extension with Firefox version 26, using our C++-to-JavaScript compiled libfte API. In addition to libfte , in roughly 200 lines of code, we implemented logic that was responsible for locating page elements to encrypt/decrypt. The Yahoo-specific logic was minimal and consisted of mappings between form fields and FPE/FTE schemes.

Fields were manually specified using unique identifiers (e.g., CSS id tags) and mapped to their corresponding P-DD FPE scheme in JavaScript. There are many options for determining what input/output formats to use for a given scheme. For this proof-of-concept we started with a set of formats we considered to be reasonable, then progressively relaxed/increased constraints on the formats appropriately until they were accepted by Yahoo's server-side validation. As a couple examples, we found that the email address field is required to have an @ symbol, and all dates are required to be valid date ranges. (e.g., month must between 1 and 12 inclusive) We expect that such constraints could be identified programmatically, at scale, using a browser automation tool such as Selenium [2].

Our Firefox extension exposes an "encrypt/decrypt" drop-down menu to the user. Prior to saving a new contact to their address book, the user can press the "encrypt" button to automatically libfte-encrypt all fields in the form. To recover the encrypted data, they user visits the page for a contact, then presses the libfte extension's decrypt button to recover the plaintext data. With further engineering efforts this encryption/decryption process could be transparent to the user. We present a screenshot of our extension in Figure 11.

7.3 Network Monitors

Finally, we turn our attention to building T-xx-$ schemes (as used in [11]) by lifting regular expressions from the Snort IDS [19]. As far as these authors are aware, this Snort IDS corpus of regular expressions is the largest and most diverse (publicly available) set of regexes used for deep-packet inspection.

In the initial exploration of FTE by Dyer et al. [11] a fundamental limitation to their construction was the need to perform a regex-to-DFA conversion for formats. Unfortunately, this creates a natural asymmetry: systems

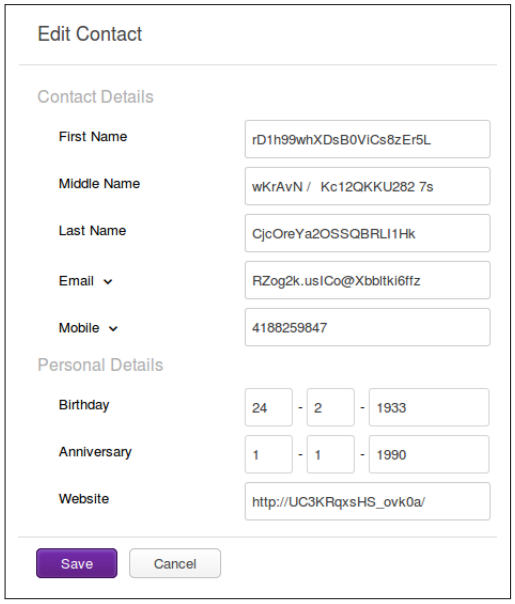

Figure 11: Screenshot from our Firefox Browser Extension that encrypts the data fields of our Yahoo address book, client-side, prior to transmission to the Yahoo servers.

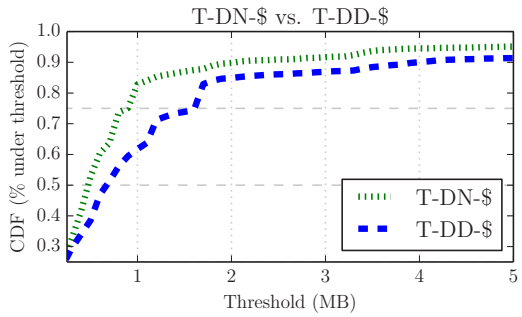

Figure 12: The CDF representing the fraction of the Snort corpus that can be instantiated for a given memory threshold. The CDF graph has a long tail and reaches 100% at 143MB for T-DN-$. We were unable to calculate the threshold for T-DD-$ to reach 100%, due to memory constraints on our test system.

such as Snort are able to perform network monitoring directly from an NFA representation, but the construction presented by Dyer et al. requires regex-to-DFA conversion. In this section we show that we've overcome this limitation — regular expressions that were intractable using the construction by Dyer et al. are no longer an obstacle, given our new NFA ranking algorithm.

Snort IDS regex corpus. To build our corpus, we started with the official Snort ruleset, version 2955, released Jan 14, 2014. Each rule in the ruleset contains a mixture of values, including static strings or regular expressions to match against traffic. From each rule we extracted all regular expressions (as defined by the `pcre` field) which resulted in 6,277 expressions. Of these, 3,458 regular expressions compiled with our regular expression parser[5]. For all regular expressions that compiled, if we were able to instantiate their precomputation table in memory, we were able to successfully utilize them for encryption.

Corpus evaluation. For each regular expression R in the Snort corpus we attempted to build a T-DD-$ and T-DN-$ scheme with an output format $\mathcal{F} \leftarrow (R, 0, 256)$, and input format that is a $\lfloor \log_2 |L(\mathcal{F})| \rfloor$-bit string. This choice of libfte scheme and α and β is motivated by the construction in [11].

In Figure 12 we plot the CDF of the fraction of the corpus that can be instantiated when constrained by a given memory threshold, for each scheme. At 1MB, roughly 60% of the corpus can be instantiated using T-DD-$ ranking, compared to roughly 85% of the corpus with T-DN-$ ranking. At 5MB, T-DN-$ is at roughly 97% and T-DD-$ is at roughly 92%. If we increase the threshold to 143MB (beyond the focus of the graph) we can instantiate 100% of the corpus using T-DN-$. Yet, at at threshold of 1GB, we are able to instantiate only 97.0% of the corpus using T-DD-$. In fact we were unable to construct some schemes (due to memory constraints) using T-DD-$, so we don't know the exact threshold required to reach 100% instantiation.

As a final test we measured the *total* memory utilization for instantiating the complete Snort corpus. Initially, we instantiated all regular expressions in the corpus using T-DD-$, which required a cumulative 8.8GB of memory. Then we considered a "best case" scenario, where, over the 97% of tractable regexs (those that we could construct a T-DD-$ scheme) we took the minimum of the T-DD-$ or T-DN-$ memory utilization. Using the best-case approach we reduced memory utilization to 6.2GB, a reduction of roughly 30%. The best-case scenario is, of course, biased against T-DN-$, as 3% of the corpus couldn't be instantiated with T-DD-$.

8 Conclusion

In this paper we presented a unifying approach for deploying format-preserving encryption (FPE) and format-transforming encryption (FTE) schemes. The approach is realized via a library we call libfte, which has two components: an offline configuration assistant to aid engineers in developing formats and understanding their system-level implications, and an API for instantiating

[5]We don't support greedy operators `*?` or case insensitivity `(?i...)`, which accounted for the majority of compilation failures. Greedy operators are used for parsing, not for language definition, and we do not support extended patterns of the form `(?...)` in general.

and deploying FPE/FTE schemes. In the development of libfte we overcame a number of obstacles. Most notably, we developed a new approach to perform FPE/FTE directly from the NFA representation of a regular expression, which was previously considered to be impractical. This significantly increases the expressiveness of regular languages for which FTE can be made useful in practice, and generally improves system efficiency, potentially making FTE a viable cryptographic option in contexts where it previously was not. We studied libfte performance empirically in several application contexts, finding that it typically introduces negligible performance overhead relative to conventional encryption. In some cases (e.g. simultaneous compressions and encryption) even enables substantial performance improvements.

Our work surfaces many avenues for future research. To name a few: investigate the security of libfte's algorithms (and FTE implementations, in general) in the face of side-channel attacks; integrate FTE into additional applications, and report on newly found algorithmic and engineering challenges; and explore efficiency improvements for specific classes of regular expressions that are in wide use. To promote further research and development, we will make libfte open source and publicly available at `https://libfte.org/`.

References

[1] Perl regular expressions man page. http://perldoc.perl.org/perlre.html, February 2014.

[2] Seleniumhq: Browser automation. http://www.seleniumhq.org/, February 2014.

[3] Voltage security. http://www.voltage.com/, February 2014.

[4] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. pages 531–545, 2000.

[5] Mihir Bellare, Thomas Ristenpart, Phillip Rogaway, and Till Stegers. Format-preserving encryption. In *Selected Areas in Cryptography*, pages 295–312. Springer-Verlag, 2009.

[6] Mihir Bellare and Phillip Rogaway. Encode-then-encipher encryption: How to exploit nonces or redundancy in plaintexts for efficient cryptography. pages 317–330, 2000.

[7] Mihir Bellare, Phillip Rogaway, and Terence Spies. The ffx mode of operation for format-preserving encryption draft 1.1, 2010.

[8] John Black and Phillip Rogaway. Ciphers with arbitrary finite domains. In *Topics in Cryptology–CT-RSA 2002*, pages 114–130. Springer Berlin Heidelberg, 2002.

[9] Debrup Chakraborty and Mridul Nandi. An improved security bound for HCTR. pages 289–302, 2008.

[10] Debra L. Cook, Angelos D. Keromytis, and Moti Yung. Elastic block ciphers: the basic design. pages 350–352, 2007.

[11] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. Protocol misidentification made easy with format-transforming encryption. In *Proceedings of the 20th ACM Conference on Computer and Communications Secuirty (CCS 2013)*, November 2013.

[12] A Goldberg and M Sipser. Compression and ranking. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, STOC '85, pages 440–448, New York, NY, USA, 1985. ACM.

[13] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[14] Shai Halevi. EME*: Extending EME to handle arbitrary-length messages with associated data. pages 315–327, 2004.

[15] Shai Halevi and Phillip Rogaway. A tweakable enciphering mode. pages 482–499, 2003.

[16] Daniel Lachaup, Kevin P. Dyer, Somesh Jha, Thomas Ristenpart, and Thomas Shrimpton. LibFTE: A toolkit for constructing practical, format-abiding encryption schemes (full version), 2014. Available from authors' websites.

[17] Moses Liskov, Ronald L Rivest, and David Wagner. Tweakable block ciphers. In *Advances in Cryptology-CRYPTO 2002*, pages 31–46. Springer, 2002.

[18] Chanathip Namprempre, Phillip Rogaway, and Thomas Shrimpton. Reconsidering generic composition. In *Advances in Cryptology – EUROCRYPT '14*, LNCS, pages 257–274. Springer-Verlag, 2014.

[19] Martin Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration*, LISA '99, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association.

[20] Phillip Rogaway. A synopsis of format-preserving encryption. Unpublished manuscript, March 2010.

[21] Rich Schroeppel. An overview of the hasty pudding cipher, July 1998.