Set It and Forget It! Turnkey ECC for Instant Integration

Dmitry Belyavsky
Cryptocom Ltd.
Moscow, Russian Federation
beldmit@cryptocom.ru

Billy Bob Brumley
Tampere University
Tampere, Finland
billy.brumley@tuni.fi

Jesús-Javier Chi-Domínguez
Tampere University
Tampere, Finland
jesus.chidominguez@tuni.fi

Luis Rivera-Zamarripa
Tampere University
Tampere, Finland
luis.riverazamarripa@tuni.fi

Igor Ustinov
Cryptocom Ltd.
Moscow, Russian Federation
igus@cryptocom.ru

ABSTRACT

Historically, Elliptic Curve Cryptography (ECC) is an active field of applied cryptography where recent focus is on high speed, constant time, and formally verified implementations. While there are a handful of outliers where all these concepts join and land in real-world deployments, these are generally on a case-by-case basis: e.g. a library may feature such X25519 or P-256 code, but not for all curves. In this work, we propose and implement a methodology that fully automates the implementation, testing, and integration of ECC stacks with the above properties. We demonstrate the flexibility and applicability of our methodology by seamlessly integrating into three real-world projects: OpenSSL, Mozilla's NSS, and the GOST OpenSSL Engine, achieving roughly 9.5x, 4.5x, 13.3x, and 3.7x speedup on any given curve for key generation, key agreement, signing, and verifying, respectively. Furthermore, we showcase the efficacy of our testing methodology by uncovering flaws and vulnerabilities in OpenSSL, and a specification-level vulnerability in a Russian standard. Our work bridges the gap between significant applied cryptography research results and deployed software, fully automating the process.

KEYWORDS

applied cryptography; public key cryptography; elliptic curve cryptography; software engineering; software testing; formal verification; GOST; NSS; OpenSSL

1 INTRODUCTION

In 1976, Whitfield Diffie and Martin Hellman published the first key-exchange protocol [15] (based on Galois field arithmetic) that provides the capability for two different users to agree upon a shared secret between them. In 1985, Miller [32] and Koblitz [28] proposed public-key cryptosystems based on the group structure of an elliptic curve over Galois fields; from these works, an Elliptic Curve Diffie-Hellman (ECDH) variant arose. In 1994, Scott Vanstone proposed an Elliptic Curve Digital Signature Algorithm (ECDSA) variant (for more details see [25]). However, the main advantage of using Elliptic Curve Cryptography (ECC) is the smaller keys compared to their Galois field DH and DSA initial proposals.

From the birth of ECC, which was focused on its mathematical description, the study, analysis, and improvement of elliptic curve arithmetic to achieve performant, constant-time, *exception-free*, and formally verified ECC implementations are clear research trends. Nevertheless, practice sometimes misaligns with theory, and by integrating theoric works into real-world deployments, vulnerabilities arise and compromise the given ECC scheme security.

Motivation. On the practice side, there is no shortage of examples of this misalignment. Brumley and Hakala [10] published the first (microarchitecture) timing attack on OpenSSL's ECC implementation in 2009, with countermeasures by Käsper [27] and later Gueron and Krasnov [20]. But OpenSSL supports over 80 named curves, and the scope of these countermeasures is only three: NIST curves P-224, P-256, and P-521, even later augmented with formal verification guarantees [35] after patching defects [31]. CVE-2018-5407 "PortSmash" [3] finally led to wider countermeasures [42] a decade later, but small leakage persists in the recent "LadderLeak" attack [4]. Still, even if current solutions hedge against timing attacks, the question of functional correctness remains: CVE-2011-1945 from [9] is the only real-world bug attack [8] we are aware of, deterministically recovering P-256 keys remotely by exploiting a carry propagation defect.

BoringSSL approaches the constant time and functional correctness issues by narrowing features, only supporting P-224, P-256, and X25519, leveraging formal verification guarantees for Galois field arithmetic from Fiat [19]. Mozilla's NSS approach is similar, removing support for the vast majority of curves—two of which (P-256, X25519) leverage the formal verification results from HACL* [43], while others still use generic legacy code with no protections or guarantees. Stripping support is not a viable option for fuller-featured libraries, OpenSSL being one example but generally any project with even a slightly larger scope. *How can these projects retain features yet provide constant-time and functional correctness confidence?*

Contributions. Our main contribution focuses on fully automatic implementation, testing, and integration of ECC stacks on real-world projects like OpenSSL, Mozilla's NSS, and GOST OpenSSL Engine. Our full-stack ECC implementations achieve about 9.5x, 4.5x, 13.3x, and 3.7x speedup for key generation, key agreement, signing, and verifying, respectively. Furthermore, our flexible and applicable proposal can be easily adapted to any curve model. To our knowledge, this is the first hybrid ECC implementation between *short Weierstrass* and *Twisted Edwards* curves, which has been integrated to OpenSSL. Additionally, our methodology allowed us to find and fix very special vulnerabilities on development versions of OpenSSL and official Russian standards for cryptography.

Outline. Section 2 gives the elliptic curve background concepts related to curve models and cryptosystems; in particular, Section 2.1 describes GOST and the related OpenSSL GOST Engine. Section 3 introduces our library-agnostic unit and regression testing framework for ECC implementations (ECCKAT); while Section 4 presents our dynamic ECC layer generation (ECCKiila) and performance results. Finally, Section 5 draws conclusions.

2 BACKGROUND

An elliptic curve E defined over a Galois field $GF(p)$, is usually described by an equation of the following form

$$E\colon y^2 = x^3 + ax + b, \quad a \in GF(p),\ b \in GF(p); \tag{1}$$

called a short Weierstrass curve. Furthermore, a point on the curve E is a pair (x, y) satisfying (1), but there is also a *point at infinity* denoted $\mathcal{O}$, which plays the role of the neutral element on E. Additionally, given a positive integer k, point multiplication is the computation of k times a given point P denoted by $[k]P$. The order of a point P on E corresponds with the smallest positive integer q such that $[q]P$ gives $\mathcal{O}$. In our work, we assume the cardinality of E is equal to $h \cdot q$ where q is a prime number with $\lg(q) \approx \lg(p)$, and $h \in \{1, 4, 8\}$.

When 4 divides h, there is a *Twisted Edwards curve*

$$E_t : eu^2 + v^2 = 1 + du^2v^2 \tag{2}$$

having the same cardinality, and each point on E_w (1) maps into E_t, and viceversa, using the mappings

$$(x, y) \mapsto (u, v) := \left(\frac{x - t}{y}, \frac{x - t - s}{x - t + s}\right), \text{ and} \tag{3}$$

$$(u, v) \mapsto (x, y) := \left(\frac{s(1 + v)}{1 - v} + t, s\frac{(1 + v)}{(1 - v)u}\right), \tag{4}$$

where $s = (e-d)/4 \bmod p$, $t = (e+d)/6 \bmod p$, $a = (s^2-3t^2) \bmod p$, and $b = (2t^3 - t \cdot s^2) \bmod p$.

Projective points on the short Weierstrass curve. We choose to work with *projective* points $(X : Y : Z)$ satisfying $ZY^2 = X^3 + aXZ^2 + bZ^3$ where the *affine* point $(X/Z, Y/Z)$ belongs to E_w. Moreover, the projective representation of $\mathcal{O}$ is $(0 : 1 : 0)$, which does not satisfy the *affine* curve equation of E_w.

Because of the nature of the short Weierstrass curves, one needs to handle some exceptions when: (i) adding or doubling points with $\mathcal{O}$; (ii) adding points when $P = \pm Q$. In particular, any *mixed* point addition takes as inputs a *projective* point and an *affine* point, which implies no exception-free implementation will be possible for this *mixed* point addition—$\mathcal{O}$ has no *affine* representation!

Failure to use *exception-free* formulas could lead to successful *exceptional procedure attacks* [24], implying a possible break of ECC security. Still, apart from theoretical attacks there is the question of functional correctness. For example, CVE-2017-7781 affected Mozilla's NSS, failing to account for the $P = \pm Q$ exceptions in textbook mixed Jacobian-affine point addition—a bug present in their codebase for over a decade.

Projective points on the Twisted Edwards curve. To achieve efficient curve arithmetic, we choose to work with *extended projective* points $(X : Y : T : Z)$ satisfying $eX^2Z^2 + Y^2Z^2 = Z^4 + dX^2Y^2$, where the *affine* point $(X/Z, Y/Z)$ belongs to E_t and $T = XY/Z$. The main advantage of using *Twisted Edwards curves* is the "cheap" exception-free formula for point addition; in particular, $(0 : 1 : 0 : 1)$ represents $\mathcal{O}$ and corresponds with the *affine* point $(0, 1)$ on E_t.

The main blocks of ECC cryptosystem implementations consist of (i) key generation, (ii) key agreement procedure, and (iii) digital signature algorithm.

Key generation. Given an order-q point g the user randomly and uniformly chooses a secret key α from $\{1, \ldots, q-1\}$, and computes the public key $P = [\alpha]g$.

Key agreement with cofactor clearing. Assume the users *Alice* and *Bob* need to agree a secret shared key; thus, Alice generates her private key $\alpha_a \in \{1, \ldots, q-1\}$ and a public key $P_a = [\alpha_a]g$ by using the key generation block; similarly, Bob generates α_b and $P_b = [\alpha_b]g$. Next, Alice and Bob compute $s_{ab} = [h \cdot \alpha_a]P_b$ and $s_{ba} = [h \cdot \alpha_b]P_a$, respectively. Consequently,

$$s_{ab} = [h \cdot \alpha_a]P_b = [h \cdot \alpha_a \cdot \alpha_b]g = [h \cdot \alpha_b \cdot \alpha_a]g = [h \cdot \alpha_b]P_a = s_{ba}$$

is the secret shared key. The multiplication by h is called *cofactor clearing* and ensures the protocol fails if P_a or P_b are adversarially in the order-h subgroup. When $h = 1$, ECC CDH [1] and classical ECDH variants are equivalent.

Digital signature algorithm (ECDSA). The user generates a private key $\alpha \in \{1, \ldots, q-1\}$ and a public key $P = [\alpha]g$ by using the key generation block; using an approved hash function Hash(), the signature (r, s) on message m is computed by

$$r = ([k]g)_x \bmod q, \quad s = k^{-1}(\hat{m} + \alpha r) \bmod q \tag{5}$$

where k is a nonce chosen uniformly from $\{1, \ldots, q-1\}$, and $\hat{m}$ denotes the representation of Hash(m) in $GF(q)$. The ECDSA signature successfully verifies if $u_1 = \hat{m} \cdot s^{-1} \bmod q$ and $u_2 = r \cdot s^{-1} \bmod q$ satisfy

$$([u_1]g + [u_2]P)_x = r \bmod q. \tag{6}$$

ECDSA is the ECC-equivalent of DSA that instead operates with the multiplicative group of a Galois field and pre-dates the ECDSA variant by at least a decade.

Security. Mathematically speaking, the security of ECC relies on the hardness of computing an integer k given $[k]P$ called the *Elliptic Curve Discrete Logarithm Problem (ECDLP)*. In certain instances, ECDLP solves by using the *small-subgroup* [30] when the curve cardinality is smooth, and *invalid-curve* [7] attacks when the input point P does not satisfies the curve equation.

As a consequence, ECC implementations often seek to be secure against *combined* attacks that use *small-subgroup* attacks with *invalid-curve* attacks using the *twist* curve E' determined by the equation $y^2 = x^3 + ax - b$. The *twist* curve E' has cardinality $h' \cdot q' = p + 1 + t_E$ where $h \cdot q = p + 1 - t_E$ is the cardinality of E and t_E is a curve constant (the *Frobenius trace*). However, a curve E is *twist secure* if h' is a small integer and $q' \approx p$ is a large prime number. For example, the following two GOST curves are *twist secure*:

- the curve `id_tc26_gost_3410_2012_256_paramSetA` has
```
q = 0x3FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF027322037
      8499CA3EEA50AA93C9F265,
q'= 0x400000000000000000000000000000000FD8CDDFC8
      7B6635C115AF556C360C67,
```

and both h and h' equal 4;

- the curve `id_tc26_gost_3410_2012_512_paramSetC` has

```
q = 0x400000000000000000000000000000000000000000
      0000000000000000000000003673245B9AF954FFB3CC
      5600AEB8AFD33712561858965ED96B9DC310B80FDA
      F7,
q'= 0x3FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
      FFFFFFFFFFFFFFFFFFFFFFC98CDBA46506AB004C33
      A9FF5147502CC8EDA9E7A769A12694623CEF47F023
      ED,
```

and also both h and h' equal 4.

2.1 GOST

The system of Russian cryptographic standards (usually called *GOST algorithms*) started to develop in the 1980s after decades of *top secret cryptography*. The first Russian (or, rather, Soviet) relatively open cryptographic standard was published in 1989, describing symmetric cipher and MAC algorithms.

The first Russian standard for digital signatures was developed simultaneously with DSA and these two standards were published in 1994 with an interval of only four days. Like DSA, the Russian GOST R 34.10-94 was the ElGamal-style algorithm in Galois field of prime modulo, but the formula was slightly different: $s = (k \cdot \hat{m} + \alpha r) \bmod q$. The hash function to be used for calculating $\hat{m}$ was strictly defined and described in a separate standard, based on the GOST symmetric cipher.

In 2001 the new digital signature standard was adopted—the adaptation of the previous standard to elliptic curves over $GF(p)$, allowing only $\lg(p) = 256$. The hash function was not changed.

In 2012 the third Russian digital signature standard was adopted, almost word-to-word copy of the previous standard. The only changes were (i) the length of p can now be either 256 or 512; (ii) the standard prescribes to use a new (completely different) hash function. The official name of the current Russian digital signature standard is GOST R 34.10-2012 , and GOST R 34.11-2012 describes the hash function. The translation of these standards in English were published as RFC 7091 [17] and RFC 6986 [18] respectively.

GOST: digital signatures. Informally speaking and aligning with our previous notation, the Russian signature algorithm formula is

$$r = ([k]g)_x \bmod q, \quad s = k\hat{m} + r\alpha \bmod q \tag{7}$$

where α is the signer's secret key, k is a nonce chosen randomly and uniformly from $\{1, \ldots, q-1\}$, g is the base point of an elliptic curve, and q is the order of g. The GOST signature successfully verifies if $z_1 = s \cdot \hat{m}^{-1} \bmod q$ and $z_2 = -r \cdot \hat{m}^{-1} \bmod q$ satisfy

$$([z_1]g + [z_2]P)_x = r \bmod q. \tag{8}$$

In connection with these standards a number of sub-ordinary standards were adopted (the Russian standardization system has different levels of standards, but the difference is rather bureaucratic than practical). In parallel the corresponding RFCs were published, including several curves for use in GOST digital signature algorithms. The first three curves with $\lg(p) = 256$ were described in RFC 4357 [36] (peculiar that for many years it was the only normative reference to these curves—their first appearance in Russian standards was in 2019). All these curves have only the trivial cofactor $h = 1$, i.e. they are cyclic groups and all curve points can be a legal public key. After the adoption of the new digital signature standard, two curves with $\lg(p) = 512$ and $h = 1$ were standardized as well as two Twisted Edwards curves with $h = 4$: one with $\lg(p) = 256$ and the other with $\lg(p) = 512$, all described in RFC 7836 [40]. One important aspect is—at the standardization level—the Twisted Edwards curves are still specified as short curves for compatibility reasons.

GOST: key generation. Russian standards say nothing about the generation of secret keys: any random number α between 1 and $q - 1$ can be used as a secret key. Surprisingly despite the Russian regulation authority paying great attention to random number generation, there is no standard for this procedure, only some classified requirements. The public key for a given secret one is calculated as the result of multiplication of the curve base point by the secret key. In that sense, it does not differ from other standard definitions of ECC key generation.

GOST: key agreement (VKO). The VKO algorithm is defined in one of the sub-ordinary standards and described in RFC 7836 [40]. It consists of 2 steps: (i) a curve point K is calculated by the formula

$$K = [h \cdot (UKM \cdot x \bmod q)]Y$$

where x is the secret key of one side, Y is the public key of the other side, UKM is an optional non-secret parameter (*User Key Material*) known by both sides, q is the order of the base point and h is the cofactor of the used elliptic curve; (ii) the shared key is the hash of the affine coordinates of K. In this light, VKO shares similarities to ECC CDH [1], also featuring cofactor clearing but additionally utilizing UKM. But in contrast to NIST SP 800-108 [12] that accounts for (the equivalent of) UKM in the subsequent key derivation hash function, VKO encorporates UKM directly at the ECC level.

GOST: public key encryption. Incorrect phrase *encryption according to GOST R 34.10* is often used, but actually the asymmetric key encryption has never been used. Instead VKO calculates a shared key, then a symmetric encryption algorithm uses the key for data encryption. In this light, it is *hybrid encryption*.

2.2 The GOST OpenSSL Engine

The GOST Engine project was started during OpenSSL 1.0 development. Before OpenSSL 1.0 (released 2010) the engine mechanism allowed to provide own digests, ciphers, random number generators, RSA, DSA, and EC. Since OpenSSL 1.0 it became possible to use OpenSSL's engine mechanism [41] to provide custom asymmetric algorithms.

In short, `gost-engine` was created as a reference implementation of the Russian GOST cryptographic algorithms: symmetric cipher GOST 28147-89, hash algorithm GOST R 34.11-94, and asymmetric algorithms GOST R 34.10-94 (DSA-like, now deprecated and removed) and GOST R 34.10-2001 (ECDSA-like). In 2012, the support of new Russian hash algorithm GOST R 34.11-2012 Streebog (RFC 6986 [18]) and GOST R 34.10-2012 asymmetric algorithms (256 and 512 bits) was provided. After publishing RFC 7836 [40] and providing non-trivial cofactor support in OpenSSL, the support of the new parameters based on Twisted Edwards curves was added, though the implementation itself does not use Edwards representation and relies on OpenSSL's EC module for the curve arithmetic. It is worth mentioning that all the parameter sets (curves) specific for GOST R 34.10-2001 are allowed for use in GOST R 34.10-2012, though the hash algorithms are different.

Being OpenSSL-dependent software, `gost-engine` has been used many times as regression testing. Not only for general engine functionality, but also for lower level OpenSSL internals such as the EC module as discussed later in Section 3.

Deployments. Until OpenSSL 1.1.0 (released 2016), `gost-engine` was a part of OpenSSL and was distributed together. During 1.1.0 development, the engine code was moved to a separate GitHub repository[1]. Currently, the engine is available as separate package in RedHat-based Linux distributions, Debian-based distributions, and popular in Russia ALT Linux distribution. It is also widely used as an FOSS solution when there is no necessity to use the officially certified solutions. In these cases, `gost-engine` is often built from source instead of using the distribution-provided packages.

Asymmetric algorithms: architecture. Asymmetric algorithm architecture in OpenSSL requires providing two opaque callback structures per algorithm: (i) `EVP_PKEY_ASN1_METHOD` is a structure which holds a set of ASN.1 conversion, printing and information methods for a specific public key algorithm; (ii) `EVP_PKEY_METHOD` is a structure which holds a set of methods for a specific public key cryptographic algorithm—those methods are usually used to perform different jobs, such as generating a key, signing or verifying, encrypting or decrypting, etc. Unfortunately, because of 15-year history of the engine, the naming of the callbacks is not extremely consistent.

Asymmetric algorithms: operations. GOST asymmetric algorithm support the following operations: (i) key generation; (ii) digital signature and verification; (iii) key derivation; (iv) symmetric 32-bytes cipher key wrap/unwrap (named encrypting/decrypting).

The best starting point is the `register_pmeth_gost` function in the `gost_pmeth.c` file. This function provides the setting of all the necessary callbacks for various asymmetric algorithms. Most functions are very similar and just call a shared wrapper around OpenSSL's EC module for the elliptic curve arithmetic with different parameters such as hash function identifier or key length.

The following functions are especially worth studying. (i) `gost_ec_keygen` in `gost_ec_sign.c` is the common function for key generation, generating a random BIGNUM in the range corresponding to the order of the selected curve's base point and calculating the matching public key value. (ii) `gost_ec_sign` in `gost_ec_sign.c` is the common function for digital signature according to RFC 7091 [17]. (iii) `gost_ec_verify` in `gost_ec_sign.c` is the common function for digital signature verification according to RFC 7091 [17]. (iv) `pkey_gost_ec_derive` in `gost_ec_keyx.c` is the common function for shared key derivation. This function allows two mechanism for derivation. The one named VKO was originally specified in RFC 4357 [36], deriving 32-bytes shared key, is implemented in the `VKO_compute_key` function in the same file. RFC 7836 [40] defines the other one deriving 64-bytes key using `VKO_compute_key` as a step of key derivation. Currently, the choice of the expected result is done by the length of a protocol-defined UKM parameter. (v) `pkey_gost_encrypt` in `gost_ec_keyx.c` is the common function for symmetric key wrap using the shared key derived via `pkey_gost_ec_derive`. The key wrap for GOST 28147-89 symmetric cipher is done according to RFC 4357 [36]. The key wrap for GOST R 34.12-2015 ciphers (Kuznyechik, RFC 7801 [16] and Magma[2]) is done according to RFC 7836 [40]. (vi) `pkey_gost_decrypt` in `gost_ec_keyx.c` is the common function for symmetric key unwrap using the shared key derived via `pkey_gost_ec_derive`. It is a reverse function for the `pkey_gost_encrypt` function.

To summarize, regarding GOST-related ECC standards, `gost-engine` utilizes OpenSSL's engine framework to its fullest—supporting key generation, key agreement (derive in OpenSSL terminology), digital signatures and verification, and hybrid encryption/decryption. It supports all curves from the relevant RFCs—all the way from the test curve, to the $h = 1$ short curves, to the $h = 4$ short curves with Twisted Edwards equivalence. In total, eight distinct curves with several Object Identifier (OID) aliases at the standardization level.

3 ECC UNIT TESTING: ECCKAT

In this section, we present ECCKAT: a library-agnostic unit and regression testing framework for ECC implementations. The motivation for ECCKAT began with significant restructuring of OpenSSL's EC module introduced with major release 1.1.1 (released 2018). While the library featured simple positive testing of higher-level cryptosystems such as ECDH and ECDSA, this provides very little confidence in the underlying ECC implementation. To see why this is so, consider a scalar multiplication implementation that returns a constant: this will always pass ECDH functionality tests because the shared secret will be that constant, but is clearly broken. Similarly on the ECDSA side, consider a verification implementation that always returns true: this will always pass positive tests, but is clearly broken.

With that in mind, ECCKAT uses a data-driven testing (DDT) approach heavily relying on Known Answer Tests (KATs). The high level concept is as follows: (i) collect existing KATs from various sources such as standards, RFCs, and validation efforts; (ii) augment these with negative tests and potential corner cases, and extend to arbitrary curves using an Implementation Under Test (IUT) independent implementation; (iii) output these tests in a standardized format, easily consumable downstream for integration into library-specific test harnesses. Given the wide range of curves in scope, this should be as automated as possible. In the following sections, we expand on these aspects which make up our implementation of ECCKAT.

3.1 Collecting Tests

The purpose of this first step is to build a corpus of KATs that are already present in public documents. The goal is not only to utilize these tests but also understand their nature, limitations, and how they can be expanded.

[1]https://github.com/gost-engine/engine

[2]https://tools.ietf.org/html/draft-dolmatov-magma-06

Tests: ECC CDH. The NIST Cryptographic Algorithm Validation Program (CAVP)[3] provides test vectors for cofactor Diffie-Hellman on the following curves: `P-192`, `P-224`, `P-256`, `P-384`, `P-521`, `B-163`, `B-233`, `B-283`, `B-409`, `B-571`, as well as the Koblitz curve variant of each binary curve. The test vectors include the following fields: `dIUT`, the IUT's ephemeral private key; `QIUTx`, `QIUTy`, the IUT's ephemeral public key; `QCAVSx`, `QCAVSy`, the peer public key; `ZIUT`, the resulting shared key—in this case the x-coordinate of the ECC CDH computation. We added functionality to ECCKAT that parses these test vectors and makes them part of the unit test corpus.

Tests: ECDSA. CAVP also provides ECDSA test vectors for the aforementioned curves, that in fact aggregate many types of tests. Public key validation vectors give both negative and positive tests for `Qx`, `Qy` point public keys. Negative tests include coordinates out of range (i.e. must satisfy $[0, p)$ for prime curves or sufficiently small polynomial degree for binary curves) and invalid point (i.e. must satisfy the curve equation), anything else being a positive test. The negative tests are conceptually similar to the Project Wycheproof[4] ECC-related KATs. Key generation vectors include a private key `d` and the resulting `Qx`, `Qy` public key point, using the default generator point. Finally, on the ECDSA side the signing vectors include the long term private key (`d`), corresponding public key (`Qx`, `Qy`), the message to be signed (`Msg`), the ECDSA nonce (`k`), and the resulting signature (`R`, `S`). Each test is additionally parameterized by the particular hash function to apply to `Msg`. The ECDSA verification vectors are similar, but omit the private information `d` and `k`, also extending to both positive and negative tests (modifying one of `Msg`, `R`, `S`, or the public key). We added functionality to ECCKAT that parses these test vectors and makes them part of the unit test corpus.

Tests: Deterministic ECDSA. The CAVP ECDSA signing tests must parameterize by the nonce to counteract the non-determinism in stock ECDSA. In contrast, RFC 6979 [37] proposes a deterministic form of ECDSA that, at a high level, computes the nonce as a function of the private key and message to be signed. The document provides test vectors for the exact same set of curves used in the NIST CAVP, spanning both deterministic ECDSA signing as well as key generation. We added functionality to ECCKAT that parses these test vectors and makes them part of the unit test corpus. Deterministic ECDSA will likely feature in the upcoming renewed FIPS 186-5 [2].

3.2 Augmenting Tests

Based on the previously collected tests and our analysis of them, the next step is to expand these tests in several directions. First and foremost, the scope of ECCKAT is much wider: the handful of curves above is insufficient. We extended to general (legacy) curves over both prime and binary fields by utilizing the SageMath computer algebra system[5]. This gives us an IUT-independent ground truth during test generation. We built a large database of standardized curves with their specific curve parameters (semi-automated with the OpenSSL `ecparam` tool, listing over 80 standardized named curves), stored in JSON format that ECCKAT parses and uses the SageMath EC module to instantiate these curves given their parameters.

In terms of methodology, we deemed the previously collected ECDSA and deterministic ECDSA tests sufficient. In this case, ECCKAT simply extends coverage by allowing any legacy curve, computing the expected ECDSA output with SageMath arithmetic. We treat key generation tests similarly, again simply computing scalar multiplications with SageMath.

Methodology-wise, the most significant deficiency we discovered was the lack of negative tests for ECC CDH. The reason ECC CDH differs from classical Diffie-Hellman is to make sure the key agreement protocol fails for points of small order in adversarial settings. Yet surprisingly none of the existing tests actually check for this. For curves of prime order, the check is implicit because ECC CDH and classical ECDH are equivalent. But all binary curves (naturally including those in the original tests) have non-trivial cofactors by definition, as well as all legacy curve equivalents of Edwards curves, Twisted Edwards curves, and Montgomery curves require $h \geq 4$ (not in scope of the original tests). It is a rather peculiar dichotomy since binary curves have mostly fallen out of use, while current ECC trends for prime curves are strongly towards these modern forms (e.g. both X25519 [6] and X448 [21] are standardized in RFC 7748 [29] and widely deployed with e.g. codepoints in both TLS 1.2 RFC 8422 [34] and TLS 1.3 RFC 8446 [39]).

When applicable, i.e. curves with $h \neq 1$, ECCKAT generates negative tests for ECC CDH as follows. First, with SageMath find either a generator of the full elliptic curve group, i.e. an order-hq point if the group is cyclic, or with maximal order in the (in practice, rare) non-cyclic case. Scalar multiplication by q then yields a malicious generator of the largest small subgroup. This is precisely the peer point that should produce ECC CDH protocol failure, since the cofactor clearing (i.e. integer multiplication between the scalar and h) will cause the resulting scalar multiplication to yield $\mathcal{O}$: the peer point has either order h (cyclic case) or some divisor of h (non-cyclic case).

Lastly, we do note a slight deficiency in the original public key validation negative tests. They are only *partial* public key validation in that positive tests only ensure coordinates are in range and satisfy the curve equation. For prime-order curves, this is enough to guarantee order-q points and *full* public key validation is implicit. But this is not true for curves with $h \neq 1$. We claim this is only a minor issue because it is rare for real-world implementations to carry out explicit *full* public key validation (i.e. checking that scalar multiplication by q yields $\mathcal{O}$) at all, since it is costly and normally handled in other more efficient ways at the protocol level (e.g. with cofactor clearing).

We also added selective important corner cases for key generation. These include positive tests for extreme private keys (i.e. all keys in $[1, 2^b)$ and $[q - 2^b, q)$ for some reasonable bound $b > 1$) and negative tests for out of range keys (e.g. negative, zero, q or larger). These are important because underlying scalar multiplication implementations often make assumptions about scalar ranges that may or may not be ensured higher in the call stack.

[3]https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program
[4]https://github.com/google/wycheproof
[5]https://www.sagemath.org/

We feel that such augmentation is similar (in spirit) to the work of Mouha and Celi [33], that extended NIST CAVP tests to larger message lengths and led to CVE-2019-8741.

3.3 Integrating Tests

With the now expanded tests, the next step is applying these tests to specific libraries. The end goal is not a one-off evaluation, but rather the ability to apply these tests in a CI setting in an automated way and ease the integration of these unit tests into downstream projects. To that end, we now describe three backends ECCKAT currently supports.

Test Anything Protocol (TAP). Our most generic solution drives TAP[6] test harnesses. With roots in Perl going back to the 80s, TAP has evolved into a programming language-agnostic software testing framework made up of test producers and consumers. For this backend, ECCKAT generates shell-based tests using the Sharness[7] portable shell library, originally developed for Git CI. The advantage of this backend is its portability and flexibility. The disadvantage is, while the TAP tests themselves are library-agnostic, the test harnesses are indeed library-specific. This means downstream projects must either parse the TAP tests themselves and convert them to a format their internal testing framework understands (worst case), or write simple (again, library-specific) test harness applications that conform to the input and output expectations of the sample harnesses.

OpenSSL's testing framework. Following CVE-2014-0160 "HeartBleed", OpenSSL's testing framework was rapidly overhauled and continues to evolve daily. In the scale of OpenSSL testing (which is mostly TAP-based), the types of tests ECCKAT produces are very low level for OpenSSL, which is much more than a cryptography library. A significant change introduced in OpenSSL 1.1.0 (2016)—which, for the library, marked the switch from transparent to opaque structures—expanded the `evp_test` harness to generically support public key operations through OpenSSL's high level EVP ("envelope") interface. This is precisely the correct level to integrate ECCKAT tests.

Our OpenSSL backend for ECCKAT first encodes both private and public keys to the PEM standard format. It does this using the `asn1parse` OpenSSL CLI utility that, at a high level, directly injects ground truth ECCKAT values into an ASN1 structure, that can then be coerced to one of several portable formats—PEM in this case, but DER is equally feasible.

In terms of test types that `evp_test` understands, the format is a fairly simple text file containing PEMs for key material and then test parameters, either positive or negative; in the context of ECCKAT it supports testing key generation, key agreement (derivation in OpenSSL terminology), and digitally signing and verifying.

Our deterministic ECDSA KATs integrate smoothly into `evp_test` yet classical ECDSA does not. This issue is not OpenSSL-specific: it is normal for libraries to handle nonce selection internally and not expose this to application developers. There is no simple way to inject the chosen nonce into the ECDSA signing process. We feel this is strong motivation for libraries to migrate to deterministic ECDSA, where this KAT-style testing is very natural.

The motivation for the OpenSSL backend to exist at all and not simply use a generic TAP harness is the breadth of OpenSSL testing. A strong point of `evp_test` is it requires only a single application invocation, whereas ECCKAT's generic TAP backend uses a single test harness invocation per test. This would be prohibitively slow in the context of OpenSSL, which undergoes rapid development and already has a significant CI load, with frequent timeouts for GitHub PRs. In summary, test efficiency can be a practical issue, depending on the library.

GOST Engine's testing framework. Part of the existing `gost-engine` test framework is Perl TAP-driven, and this is a convenient place to integrate our ECCKAT tests. The engine already supports key generation through the OpenSSL CLI `genpkey` utility. As part of our work, as an FOSS contribution we extended `gost-engine` to support CLI key agreement through the OpenSSL `pkeyutl` utility. At a high level, our `gost-engine` backend is quite similar to the OpenSSL backend—similarly encoding ground truth key material from ECCKAT with the `asn1parse` utility. The differences are the test data being embedded directly into the Perl source as a hash structure instead of a standalone text file to match the current test framework, and the test logic calling the relevant OpenSSL CLI utilities to form the test harness itself. Our ECCKAT `gost-engine` backend does not support GOST digital signatures at this time. We are currently discussing porting the deterministic ECDSA concept to GOST-style signatures. In summary, our ECCKAT `gost-engine` backend provides positive and negative test coverage over all GOST curves for both key generation and key agreement.

3.4 ECCKAT: Results

While we have applied and deployed ECCKAT to ECCKiila (discussed later in Section 4) in a CI environment, here we summarizes our results of applying ECCKAT to other libraries. This demonstrates the flexibility and applicability of ECCKAT.

OpenSSL: ECC scalar multiplication failure. Applying ECCKAT to `gost-engine`, we identified cryptosystem failures for the `id_Gost-R3410_2001_CryptoPro_C_ParamSet` curve. Investigating the issue, OpenSSL returned failure when attempting to serialize the output point of scalar multiplication, which was incorrectly $\mathcal{O}$. Internal to the OpenSSL EC module, this was due to the chosen ladder projective formulae [23, Eq. 8] being undefined for a zero x-coordinate divisor—a restriction noted by neither the authors nor EFD[8]. This caused the entire scalar multiplication computation to degenerate and eventually return failure at the software level.

Broader than GOST, the $x = 0$ case can happen whenever prime curve coefficient b is a quadratic residue, and we integrated this test logic into ECCKAT for all curves; but the discovery was rather serendipitous. Most GOST curves choose the generator point as the smallest non-negative x-coordinate that yields a valid point—in this case, $x = 0$. Luckily we identified this issue during the development of OpenSSL 1.1.1, hence the issue did not affect any release version of OpenSSL. We developed the fix for OpenSSL (PR #7000, switching

[6]http://testanything.org/
[7]https://github.com/chriscool/sharness
[8]https://hyperelliptic.org/EFD/g1p/auto-shortw-xz.html#ladder-ladd-2002-it-3

to [23, Eq. 9]) as well as integrated our relevant tests into their testing framework.

OpenSSL: ECC CDH vulnerability. Applying ECCKAT to the development branch of OpenSSL 1.1.1 identified negative test failures in cofactor Diffie-Hellman. Investigating the issue revealed the cause to be mathematically incorrect side channel mitigations at the scalar multiplication level. As a timing attack countermeasure (ported from CVE-2011-1945 by [11]), the ladder code first padded the scalar by adding either q or $2q$ to fix the bit length and starting iteration of the ladder loop. But in key agreement scenarios, there is no guarantee the peer point is an order-q point—only a point with order dividing hq if it satisfies the curve equation, i.e. is an element of the elliptic curve group. This caused negative tests to fail for all curves with a non-trivial cofactor—for named curves in OpenSSL, this included all binary curves and the 112-bit `secp112r2` Certicom curve with $h = 4$.

Luckily the issue did not affect any release version of OpenSSL. We developed the fix for OpenSSL (PR #6535) as well as integrated our relevant tests into their testing framework.

GOST: VKO vulnerability. Applying ECCKAT to `gost-engine` identified negative test failures in VKO key agreement for the two curves with non-trivial cofactors, similar (in spirit) to the cofactor Diffie-Hellman failures above. Investigating the issue revealed `gost-engine` multiplied by the cofactor *before* modular reduction. Consulting the Russian standard and RFC 7836 [40], surprisingly this is in fact a valid interpretation of VKO at the standardization level.

Prior to the standard change and RFC errata resulting from our work, both the Russian standard and RFC specified VKO computation as

$$(m/q \cdot UKM \cdot x \bmod q) \cdot (y \cdot P)$$

where, recalling from Section 2.1, $m = hq$ is the curve cardinality, UKM is user key material, x is the private key, q is the order of the generator, and $y \cdot P$ is the peer public key (point). With this formulation, the cofactor clearing is ineffective: it is absorbed modulo q. For the two curves satisfying $h = 4$, in case of a malicious $y \cdot P$ such as an order-h point, the computation results in one of the four points in the order-h subgroup, i.e. a small subgroup confinement attack. This can reveal the private key value modulo h and, depending on the protocol, force session key reuse.

Subsequent to our work, the Russian standard and RFC 7836 [40] now specify the compatible (in the non-adversarial sense)

$$(m/q \cdot (UKM \cdot x \bmod q)) \cdot (y \cdot P)$$

where it is explicit the cofactor clearing is *after* the modular reduction. As part of our work, we implemented the `gost-engine` fix (PR #265) and integrated all the relevant ECCKAT positive and negative tests into the `gost-engine` testing framework. Luckily, packaged versions of `gost-engine` for popular distributions such as Debian, Ubuntu, and RedHat use older versions of the engine that only feature the $h = 1$ curves, not affected by this vulnerability.

4 GENERATING ECC LAYERS: ECCKIILA

This section focuses on the ECC layer generation and required library-specific rigging. Figure 1 summarizes our proposed full stack

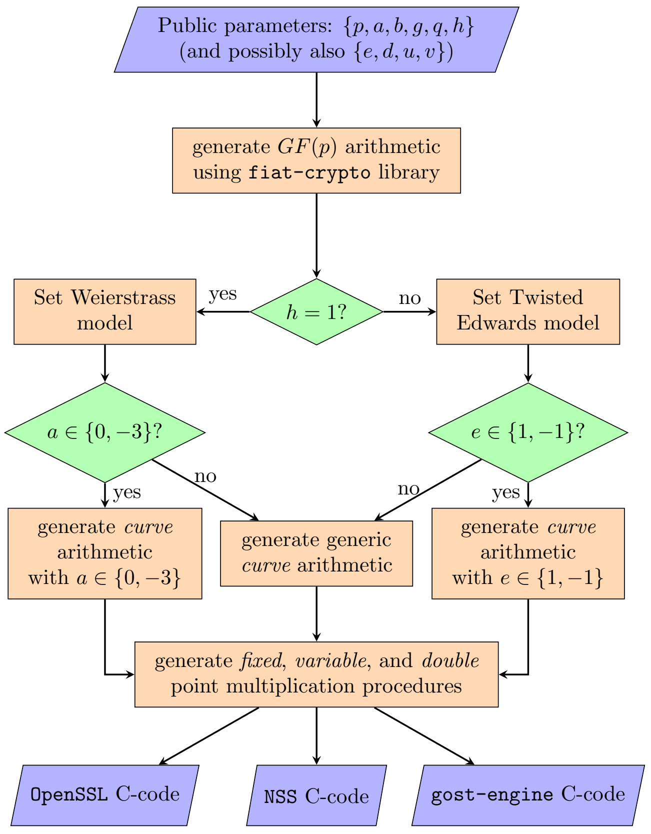
Public parameters: $\{p, a, b, g, q, h\}$
(and possibly also $\{e, d, u, v\}$)
generate $GF(p)$ arithmetic using `fiat-crypto` library
$h = 1$?
yes
no
Set Weierstrass model
Set Twisted Edwards model
$a \in \{0, -3\}$?
$e \in \{1, -1\}$?
yes
no
no
yes
generate *curve* arithmetic with $a \in \{0, -3\}$
generate generic *curve* arithmetic
generate *curve* arithmetic with $e \in \{1, -1\}$
generate *fixed*, *variable*, and *double* point multiplication procedures
`OpenSSL` C-code
`NSS` C-code
`gost-engine` C-code

Figure 1: General concept of ECCKiila. The public parameters determine a Weierstrass curve $E_w : y^2 = x^3 + ax + b$ such that $\#E_w = h \cdot q$ and g is an order-q *affine* point. The optional values determine the Twisted Edwards curve $E_t : eu^2 + v^2 = 1 + du^2v^2$ and the image point (u, v) of g on E_t.

implementation named *ECCKiila.* The name comes from the Finnish word *kiila* that means *wedge*, and it allows to dynamically create the C-code (supporting both 64-bit and 32-bit architectures, no alignment or endianness assumptions) regarding to the ECC layer as well as the rigging for seamless integration into `OpenSSL`, `NSS`, and `gost-engine`, all driven by Python Mako templating. Table 1 shows all the curves tested with ECCKiila.

Field arithmetic. In our proposal, we obtain the majority of the $GF(p)$ arithmetic by using the `fiat-crypto` project[9], that provides generation of field-specific formally verified constant time code [19]. `Fiat-crypto` has several strategies to generate the arithmetic but we have chosen the best per curve base on the form of p. In other words, the remainder of the section is centered on the EC layer that builds on top of the $GF(p)$ layer. It is important to note this is the formal verification boundary for ECCKiila—all other code on top of Fiat, while computer generated through templating and automatic formula generation for ECC arithmetic, has no formal verification guarantees. From now on, we assume all operations are performed in $GF(p)$.

[9]https://github.com/mit-plv/fiat-crypto

Short Weierstrass curves. All the legacy curves we consider in this work are prime order curves E_w, i.e. $\#E_w = q \approx p$ is a prime number and $h = 1$.

Twisted Edwards curves. Recall from Section 2.2 most of the legacy curves from `gost-engine` work on curves E_w of prime cardinality $q \approx p$ but two of them are centered on curves of cardinality $4q$ being $q \approx p$ a prime number. For those two special curves, `gost-engine` curves are represented in short Weiestrass form at the specification level (i.e. "on-the-wire" or when serialized) but internally we use the Twisted Edwards curve representation. Additionally, we implemented the mappings that connect E_w and E_t by writing (4) and (3) in their *projective* form to delay the costly inversion in $GF(p)$. We used the same strategy for `MDCurve20160`, the "Million Dollar Curve" [5] as a research-oriented example.

Point arithmetic. The way of adding points depends on the curve model being used, but we describe the three main point operations as follows: (i) the *mixed point addition* that takes as inputs a *projective* point and an *affine* point, and it returns a *projective* point; (ii) the *projective point addition* and (iii) the *projective point doubling*, which their inputs and outputs are *projective* points.

We use the *exception-free* formulas proposed by Renes et al. [38, Sec. 3] and Hisil et al. [22, Sec. 3.1, Sec. 3.3] for Weierstrass and Twisted Edwards models, respectively. In particular, all ECC arithmetic is machine generated, tied to the `op3` files[10] included in our software implementation. Our tooling is configurable in that sense, but also with high correctness confidence.

Now, recall in Weierstrass models $\mathcal{O}$ has no *affine* representation and thus the *mixed point addition* needs to catch whether the *affine* point describes $\mathcal{O}$. We solve this by asking if its affine Y-coordinate is zero and performing a conditional copy at the end of the *mixed point addition*, all in constant time. That is, we set $(0, 0)$ as the *affine* representation of $\mathcal{O}$, which does not satisfy the *affine* curve equation of E_w (no order-2 point exists on curves with prime cardinality, hence $y = 0$ is a contradiction). However, this is not the case for Twisted Edwards models, which allow fully *exception-free* formulas for point addition procedures.

Point multiplication. The heart of our ECC layer is point multiplication—*fixed* point g, *variable* point P, and also by the *double* point multiplication $[k]g + [\ell]P$. We implemented the *variable* point multiplication by representing scalars with the regular-NAF method [26, Sec. 3.2] and $\lg(q)/w$ digits with $\lg(q)$ doublings. The advantage of this method is we need only half the precomputed values compared to e.g. the base 2^w method. We support a variable window length w, and by default $w = 5$. We implemented the *fixed* point multiplication using the *comb* method (see [13, 9.3.3]) with interleaving and, similar to the variable point case, using the regular-NAF scalar representation. Our approach seeks full generality and word size independence on each architecture (32 or 64 bit), hence we automatically calculate the number of *comb*-teeth β and the distance between consecutive teeth $\lceil \lg(q)/\beta \rceil$, where the latter should be a multiple of w, considering the size of the L1 cache in this process. Therefore, the static LUTs span β tables requiring $\lceil lg(q)/\beta \rceil$ doublings. Both methods are constant time, performing exactly $\lg(q)/w$ point additions, and using linear passes to ensure key-independent memory access not only to LUTs but also in conditional point negations due to the signed scalar representation and conditional trailing subtraction to handle even scalars; the regular-NAF encoding itself is also constant time. We implemented *double* point multiplication using textbook wNAF [13, 9.1.4] combined with Shamir's trick [13, 9.1.5]. This shares the doublings, i.e. maximum $\lg(q)$ in number, but on average reduces the number of additions per scalar. This is because it is variable-time—only required in digital signature verification where all inputs are public.

[10] see e.g. https://www.hyperelliptic.org/EFD/g1p/auto-code/twisted/extended/doubling/dbl-2008-hwcd.op3

Rigging. At this point, the resulting C code yields functional arithmetic for the ECC layer. But we observe a gap between such code and real world projects. On one hand, researchers intimately familiar with ECC details lack the skill, motivation, and/or domain-specific knowledge to integrate the ECC stack into downstream large software projects. On the other hand, developers for those downstream projects lack the intimate knowledge of the upstream ECC layer to integrate properly—historical issues include assumptions on architecture, alignment, endianness, supported ranges to name a few. For example, one obscure issue encountered during OpenSSL ECCKiila integration was lack of OpenSSL unit testing for custom ECC group base points, which OpenSSL supports but ECCKiila cannot fully accelerate since the generated LUTs are static, but regardless must detect and handle the base point mismatch. Our integrations passed all OpenSSL unit tests, which is clearly not correct in this corner case.

To solve this issue, the last layer of ECCKiila is *rigging* that is essentially plumbing for downstream projects. Rigging is library-specific by nature, and ECCKiila currently supports three backends: OpenSSL, NSS, and `gost-engine`. For OpenSSL, ECCKiila generates an `EC_GROUP` structure, which is the OpenSSL internal representation of a curve with function pointers for various operations. We designed simple wrappers for three relevant function pointers, which are shallow and eventually (after sanity checking arguments and transforming the inputs to the expected format) call the corresponding scalar multiplication implementation in Figure 1. NSS is similar with an `ECGroup` structure. The `gost-engine` rigging (mostly) decouples it from OpenSSL's EC module, since it only needs to support GOST curves with explicit parameters.

Example: P-384 in OpenSSL and NSS. What follows is a walkthrough of our integration of `secp384r1` into OpenSSL and NSS. ECCKiila has a large database (JSON) of standard curves, then generates both 64-bit and 32-bit $GF(p)$ arithmetic using Fiat. It then takes the $h = 1$, $a = -3$ path in Figure 1 and generates the three relevant scalar multiplication functions that utilize exception-free formulas from [38] optimized for $a = -3$. Finally, ECCKiila emits the OpenSSL rigging for OpenSSL integration, and NSS rigging for NSS integration. Adding the code to OpenSSL is the only current manual step: one line to add the new code to the build system, one line to add the prototype of the new `EC_METHOD` structure in a header, and one line to point OpenSSL at this structure for the `secp384r1` definition. The NSS integration is very similar.

Example: GOST twisted 256-bit curve. What follows is a walkthrough of our integration of `id_tc26_gost_3410_2012_256_paramSetA` into `gost-engine`. ECCKiila takes the $h = 4$, $e = 1$ path in Figure 1

Curve	Library	External model (Standard)	Internal model (ECCKiila)	lg(p)
`secp192r1` / `P-192`	OpenSSL	Weierstrass with $a = -3$		192
`secp256r1` / `P-256`				256
`secp256k1`		Weierstrass with $a = 0$		256
`secp384r1`		Weierstrass with $a = -3$		384
`secp521r1` / `P-521`				521
`brainpool192t1`				192
`brainpool256t1`				256
`brainpool320t1`				320
`brainpool384t1`				384
`brainpool512t1`				512
SM2 (Chinese standard)				256
`X25519` / `ED25519` / `Wei25519`		Weierstrass with $a \neq 0$ and $a \neq -3$	Edwards with $e = -1$	255
`X448` / `ED448` / `Wei448`			Edwards with $e = 1$	448
`P-384`	NSS			384
`P-521`				521
`id_GostR3410_2001_CryptoPro_A_ParamSet`	GOST			
`id_GostR3410_2001_CryptoPro_B_ParamSet`				256
`id_GostR3410_2001_CryptoPro_C_ParamSet`				
`id_tc26_gost_3410_2012_512_paramSetA`				
`id_tc26_gost_3410_2012_512_paramSetB`				512
`id_tc26_gost_3410_2012_256_paramSetA`		Weierstrass with $a \neq 0$ and $a \neq -3$	Twisted Edwards with $e = 1$	256
`id_tc26_gost_3410_2012_512_paramSetC`				512
`MDCurve20160` (Million Dollar Curve)	—			256

Table 1: List of all the curves tested with ECCKiila

and generates the three relevant scalar multiplication functions that utilize exception-free formulas from [22] optimized for $e = 1$, and noting the E_w to E_t mappings (and back) are transparent to the caller (`gost-engine` rigging, in this case). Finally, ECCKiila emits the `gost-engine` rigging, and enabling this code in `gost-engine` is currently the only manual step: one line to add the code to the `gost-engine` build system, and three lines in a C switch statement to enable each of the relevant scalar multiplication routines.

Example: million dollar curve in OpenSSL. Not to limit ECCKiila to only formally standardized curves, here we showcase the research value of ECCKiila by applying it to `MDCurve20160` [5], which has undergone no formal standardization process. As such, we took the GOST approach that perhaps the E_w curve might be standardized, and the E_t curve utilized internally. We applaud this approach in GOST because, in practice, it eases downstream integration and lowers the effort bar during standardization—on the downside, it does reduce flexibility since it implies compliance with certain existing (legacy) standards.

The process for ECCKiila is a logical mix of the previous two examples: taking a path similar to the GOST example, but the generated rigging is OpenSSL. In this case, OpenSSL knows nothing about `MDCurve20160` so we obtained an unofficial OID for `MDCurve20160` and the rigging additionally emits the explicit curve parameters so OpenSSL knows how to construct its internal ECC group. The only manual steps are similar to the previous examples, yet additionally inserting these parameters.

Once OpenSSL knows about `MDCurve20160` from the automated rigging, it can drive operations with `MDCurve20160` like any other (legacy) curve: ECC key generation, ECDSA signing and verifying, and ECC CDH key agreement. This highlights the research value of ECCKiila, and gives a clear and simple path for researchers seeking dissemination and exploitation: obtain an official OID for standardization, provide curve parameters to ECCKiila, and submit a PR to downstream projects. In the case that more modern signature and key agreement schemes are desired, additional steps are needed at both the standardization, implementation, and integration levels.

4.1 ECCKiila: results

We now present the results of applying ECCKiila to the curves listed in Table 1. First, it is important to note our measurements are not on the ECCKiila code directly, rather on the application-level view of how developers and users of the corresponding libraries will transparently see the resulting performance difference. That is, we are measuring the *full integration*, not the ECCKiila code in isolation. So it includes e.g. all overheads from the rigging, any checks and serialization/deserialization the libraries perform, any memory allocation/deallocation and structure initialization, as well as any other required arithmetic not part of ECCKiila (e.g. $GF(q)$ arithmetic for ECDSA and GOST signatures).

To compare the performance of our approach, we measure the timing of unmodified OpenSSL 3.0 alpha, `gost-engine`, and NSS 3.53 (called *baseline*), and the same versions then modified with the ECCKiila output (*integration*). For each one of them, we measured the timings from the operations described in Section 2 such as key generation, key agreement (derivation), signing, and verification. For the sake of simplicity, we refer to them as keygen, derive, sign, and verify, respectively.

The hardware and software setup used to get the timings reported in this section are the following: Intel Xeon Silver 4116 2.10GHz, Ubuntu 16.04 LTS “Xenial”, GNU11 C, and `clang-10`. We used 64-bit builds, although the ECCKiila generated code selects the correct implementation using the compiler’s preprocessor at build time. For the clock cycle measurements, we used the `newspeed`[11] utility, unifying the OpenSSL and `gost-engine` measurements since it works through OpenSSL’s EVP interface and optionally supports engines. For the two NSS results, we modified their `ecperf` benchmarking utility[12] to report median clock cycles instead of wall clock time.

Table 2 reports timings for both approaches, showing the result of our proposal has good performance regarding the original versions. There are several nuances to clarify in the data. In particular, in the signature of our proposal where `id_GostR3410_-2001_CryptoPro_A_ParamSet` is quite faster than `secp256k1`: the reason is GOST signatures do not invert modulo q while ECDSA signatures do, and this is a costly operation. Also, we can see that `secp256r1` has excellent performance due to manual AVX assembler optimizations, while ECCKiila is portable C. Despite this, `id_tc26_gost_3410_2012_256_paramSetA` gives us similar performance, yet completely automated with limited formal verification guarantees and architecture independence. Last, there are some curve with extra slowdown in some operations such as `secp256r1`, `brainpoolP512t1`, and `secp521r1`. The reason for this varies. In the `secp256r1` and `secp521r1` cases, this is due to competition with curve-specific optimizations in libraries. For Brainpool curves, this is a combination of limited $GF(p)$ optimizations available both at the `fiat-crypto` and ECCKiila layers.

5 CONCLUSION

In this work, we presented two methodologies. ECCKAT allows carrying out a set of tests over an arbitrary ECC implementation, including (but not limited to) all standard curves from OpenSSL,

[11] https://github.com/romen/newspeed
[12] https://github.com/nss-dev/nss/tree/master/cmd/ecperf

Curve/Parameter	**bit**	**KeyGen**		**Derive**		**Sign**		**Verify**	
		Baseline	Integration	Baseline	Integration	Baseline	Integration	Baseline	Integration
`secp192r1`	192	587	77 (▲ 7.6x)	549	181 (▲ 3.0x)	574	75 (▲ 7.6x)	543	212 (▲ 2.6x)
`brainpoolP192t1`		574	92 (▲ 6.2x)	533	255 (▲ 2.1x)	560	90 (▲ 6.2x)	543	291 (▲ 1.9x)
`X25519` / `ED25519` / `Wei25519`	255	105	91 (▲ 1.2x)	104	173 (▽ 1.7x)	106	112 (▽ 1.1x)	284	211 (▲ 1.3x)
`secp256r1`	256	90	141 (▽ 1.6x)	139	465 (▽ 3.3x)	63	156 (▽ 2.5x)	183	524 (▽ 2.9x)
`P-256` (NSS)		310	116 (▲ 2.7x)	1628	916 (▲ 1.8x)	351	157 (▲ 2.2x)	1077	512 (▲ 2.1x)
`secp256k1`		1027	151 (▲ 6.8x)	989	400 (▲ 2.5x)	1037	165 (▲ 6.3x)	932	471 (▲ 2.0x)
`brainpoolP256t1`		939	175 (▲ 5.3x)	897	597 (▲ 1.5x)	953	187 (▲ 5.1x)	877	665 (▲ 1.3x)
`id_GostR3410_2001_CryptoPro_A_ParamSet`		1026	123 (▲ 8.3x)	1022	385 (▲ 2.7x)	993	91 (▲ 10.8x)	867	404 (▲ 2.1x)
`id_GostR3410_2001_CryptoPro_B_ParamSet`		982	129 (▲ 7.6x)	1007	449 (▲ 2.2x)	955	101 (▲ 9.4x)	845	461 (▲ 1.8x)
`id_GostR3410_2001_CryptoPro_C_ParamSet`		977	180 (▲ 5.4x)	986	639 (▲ 1.5x)	945	145 (▲ 6.5x)	848	662 (▲ 1.3x)
`id_tc26_gost_3410_2012_256_paramSetA`		960	101 (▲ 9.5x)	929	204 (▲ 4.5x)	926	69 (▲ 13.3x)	893	240 (▲ 3.7x)
`MDCurve201601`		—	155 (0.0x)	—	360 (0.0x)	—	171 (0.0x)	—	420 (0.0x)
`SM2`		1039	252 (▲ 4.1x)	889	549 (▲ 1.6x)	935	174 (▲ 5.4x)	874	612 (▲ 1.4x)
`brainpoolP320t1`	320	1470	314 (▲ 4.7x)	1430	1161 (▲ 1.2x)	1502	351 (▲ 4.3x)	1277	1271 (▲ 1.0x)
`secp384r1`	384	2156	417 (▲ 5.2x)	2117	1598 (▲ 1.3x)	2221	488 (▲ 4.5x)	1818	1823 (▽ 1.0x)
`P-384` (NSS)		2257	391 (▲ 5.8x)	4266	3225 (▲ 1.3x)	2310	454 (▲ 5.1x)	4013	1755 (▲ 2.3x)
`brainpoolP384t1`		2157	527 (▲ 4.1x)	2098	1978 (▲ 1.1x)	2206	599 (▲ 3.7x)	1828	2181 (▽ 1.2x)
`X448` / `ED448` / `Wei448`	448	309	306 (▲ 1.0x)	1046	760 (▲ 1.4x)	322	406 (▽ 1.3x)	1195	903 (▲ 1.3x)
`brainpoolP512t1`	512	3632	1325 (▲ 2.7x)	3590	4767 (▽ 1.3x)	3774	1451 (▲ 2.6x)	2959	5099 (▽ 1.7x)
`id_tc26_gost_3410_2012_512_paramSetA`		3716	664 (▲ 5.6x)	3634	2197 (▲ 1.7x)	3671	625 (▲ 5.9x)	2878	2405 (▲ 1.2x)
`id_tc26_gost_3410_2012_512_paramSetB`		3754	652 (▲ 5.8x)	3720	2359 (▲ 1.6x)	3715	619 (▲ 6.0x)	2798	2550 (▲ 1.1x)
`id_tc26_gost_3410_2012_512_paramSetC`		3663	515 (▲ 7.1x)	3653	1262 (▲ 2.9x)	3645	478 (▲ 7.6x)	3070	1387 (▲ 2.2x)
`secp521r1`	521	574	513 (▲ 1.1x)	941	1731 (▽ 1.8x)	753	742 (▲ 1.0x)	1492	2096 (▽ 1.4x)
`P-521` (NSS)		3239	480 (▲ 6.7x)	6063	3444 (▲ 1.8x)	3339	578 (▲ 5.8x)	5217	1840 (▲ 2.8x)

Table 2: Comparison of timings between the baseline and the integration from OpenSSL, `gost-engine`, and NSS. All timings are reported in clock cycles (thousands).

NSS, and `gost-engine` where it gave us excellent results because we uncovered several novel defects in OpenSSL such as a scalar multiplication failure and an ECC CDH vulnerability. Meanwhile, for GOST we detected a VKO vulnerability that can reveal sensitive information from the private key. Our second proposal ECCKiila is partially motivated by these vulnerabilities. With the use of ECCKiila, we can generate code dynamically for any curve, including all standard curves from OpenSSL, NSS, and GOST. This code is highly competitive in comparison with the original versions from OpenSSL, NSS, and `gost-engine` since we have a speedup factor up to 9.5x for key generation, 4.5x for key agreement, 13.3x for signing, and 3.7x for verifying as Table 2 shows. Furthermore, ECCKiila is flexible and robust since we can easily add new curves without increase the complexity of the development—upstream or downstream. Hence, we believe our methodologies are of interest for future work, both in research and application.

Quoting Davis [14]: *programmers need "turnkey" cryptography, not only cryptographic toolkits* and that is precisely what ECCKiila provides. The ease of integrating these stacks in downstream projects, coupled with formal verification guarantees on the Galois field arithmetic and simplicity of upper layers, and automated code generation, provides drop-in, zero-maintenance solutions for real-world, security-critical libraries. We release ECCKiila[13] as FOSS, furthermore in support of Open Science.

Acknowledgments. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 804476).

[13]https://gitlab.com/nisec/ecckiila

REFERENCES

[1] 2018. *Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography.* NIST Special Publication 800-56A Rev. 3. National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-56Ar3

[2] 2019. *Digital Signature Standard (DSS).* FIPS PUB 186-5. National Institute of Standards and Technology. https://doi.org/10.6028/NIST.FIPS.186-5-draft

[3] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port Contention for Fun and Profit. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019.* IEEE, 870–887. https://doi.org/10.1109/SP.2019.00066

[4] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. 2020. LadderLeak: Breaking ECDSA With Less Than One Bit Of Nonce Leakage. *IACR Cryptol. ePrint Arch.* 2020, 615 (2020). https://eprint.iacr.org/2020/615

[5] Thomas Baignères, Cécile Delerablée, Matthieu Finiasz, Louis Goubin, Tancrède Lepoint, and Matthieu Rivain. 2015. Trap Me If You Can - Million Dollar Curve. *IACR Cryptol. ePrint Arch.* 2015, 1249 (2015). http://eprint.iacr.org/2015/1249

[6] Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings (Lecture Notes in Computer Science)*, Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin (Eds.), Vol. 3958. Springer, 207–228. https://doi.org/10.1007/11745853_14

[7] Ingrid Biehl, Bernd Meyer, and Volker Müller. 2000. Differential Fault Attacks on Elliptic Curve Cryptosystems. In *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings (Lecture Notes in Computer Science)*, Mihir Bellare (Ed.), Vol. 1880. Springer, 131–146. https://doi.org/10.1007/3-540-44598-6_8

[8] Eli Biham, Yaniv Carmeli, and Adi Shamir. 2016. Bug Attacks. *J. Cryptology* 29, 4 (2016), 775–805. https://doi.org/10.1007/s00145-015-9209-1

[9] Billy Bob Brumley, Manuel Barbosa, Dan Page, and Frederik Vercauteren. 2012. Practical Realisation and Elimination of an ECC-Related Software Bug Attack. In *Topics in Cryptology - CT-RSA 2012 - The Cryptographers' Track at the RSA Conference 2012, San Francisco, CA, USA, February 27 - March 2, 2012. Proceedings (Lecture Notes in Computer Science)*, Orr Dunkelman (Ed.), Vol. 7178. Springer, 171–186. https://doi.org/10.1007/978-3-642-27954-6_11

[10] Billy Bob Brumley and Risto M. Hakala. 2009. Cache-Timing Template Attacks. In *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan,*

December 6-10, 2009. Proceedings (Lecture Notes in Computer Science), Mitsuru Matsui (Ed.), Vol. 5912. Springer, 667–684. https://doi.org/10.1007/978-3-642-10366-7_39

[11] Billy Bob Brumley and Nicola Tuveri. 2011. Remote Timing Attacks Are Still Practical. In *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings (Lecture Notes in Computer Science)*, Vijay Atluri and Claudia Díaz (Eds.), Vol. 6879. Springer, 355–371. https://doi.org/10.1007/978-3-642-23822-2_20

[12] Lily Chen. 2009. *Recommendation for Key Derivation Using Pseudorandom Functions.* NIST Special Publication 800-108. National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-108

[13] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren (Eds.). 2005. *Handbook of Elliptic and Hyperelliptic Curve Cryptography.* Chapman and Hall/CRC. https://doi.org/10.1201/9781420034981

[14] Don Davis. 2001. Defective Sign & Encrypt in S/MIME, PKCS#7, MOSS, PEM, PGP, and XML. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference, June 25-30, 2001, Boston, Massachusetts, USA*, Yoonho Park (Ed.). USENIX, 65–78. http://www.usenix.org/publications/library/proceedings/usenix01/davis.html

[15] Whitfield Diffie and Martin E. Hellman. 1976. New directions in cryptography. *IEEE Trans. Inf. Theory* 22, 6 (1976), 644–654. https://doi.org/10.1109/TIT.1976.1055638

[16] Vasily Dolmatov. 2016. *GOST R 34.12-2015: Block Cipher "Kuznyechik".* RFC 7801. RFC Editor. 1–14 pages. https://doi.org/10.17487/RFC7801

[17] Vasily Dolmatov and Alexey Degtyarev. 2013. *GOST R 34.10-2012: Digital Signature Algorithm.* RFC 7091. RFC Editor. 1–21 pages. https://doi.org/10.17487/RFC7091

[18] Vasily Dolmatov and Alexey Degtyarev. 2013. *GOST R 34.11-2012: Hash Function.* RFC 6986. RFC Editor. 1–40 pages. https://doi.org/10.17487/RFC6986

[19] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019.* IEEE, 1202–1219. https://doi.org/10.1109/SP.2019.00005

[20] Shay Gueron and Vlad Krasnov. 2015. Fast prime field elliptic-curve cryptography with 256-bit primes. *J. Cryptographic Engineering* 5, 2 (2015), 141–151. https://doi.org/10.1007/s13389-014-0090-x

[21] Mike Hamburg. 2015. Ed448-Goldilocks, a new elliptic curve. *IACR Cryptol. ePrint Arch.* 2015, 625 (2015). http://eprint.iacr.org/2015/625

[22] Hüseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. 2008. Twisted Edwards Curves Revisited. In *Advances in Cryptology - ASIACRYPT 2008, 14th International Conference on the Theory and Application of Cryptology and Information Security, Melbourne, Australia, December 7-11, 2008. Proceedings (Lecture Notes in Computer Science)*, Josef Pieprzyk (Ed.), Vol. 5350. Springer, 326–343. https://doi.org/10.1007/978-3-540-89255-7_20

[23] Tetsuya Izu and Tsuyoshi Takagi. 2002. A Fast Parallel Elliptic Curve Multiplication Resistant against Side Channel Attacks. In *Public Key Cryptography, 5th International Workshop on Practice and Theory in Public Key Cryptosystems, PKC 2002, Paris, France, February 12-14, 2002, Proceedings (Lecture Notes in Computer Science)*, David Naccache and Pascal Paillier (Eds.), Vol. 2274. Springer, 280–296. https://doi.org/10.1007/3-540-45664-3_20

[24] Tetsuya Izu and Tsuyoshi Takagi. 2003. Exceptional Procedure Attack on Elliptic Curve Cryptosystems. In *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings (Lecture Notes in Computer Science)*, Yvo Desmedt (Ed.), Vol. 2567. Springer, 224–239. https://doi.org/10.1007/3-540-36288-6_17

[25] Don Johnson, Alfred Menezes, and Scott A. Vanstone. 2001. The Elliptic Curve Digital Signature Algorithm (ECDSA). *Int. J. Inf. Sec.* 1, 1 (2001), 36–63. https://doi.org/10.1007/s102070100002

[26] Marc Joye and Michael Tunstall. 2009. Exponent Recoding and Regular Exponentiation Algorithms. In *Progress in Cryptology - AFRICACRYPT 2009, Second International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21-25, 2009. Proceedings (Lecture Notes in Computer Science)*, Bart Preneel (Ed.), Vol. 5580. Springer, 334–349. https://doi.org/10.1007/978-3-642-02384-2_21

[27] Emilia Käsper. 2011. Fast Elliptic Curve Cryptography in OpenSSL. In *Financial Cryptography and Data Security - FC 2011 Workshops, RLCPS and WECSR 2011, Rodney Bay, St. Lucia, February 28 - March 4, 2011, Revised Selected Papers (Lecture Notes in Computer Science)*, George Danezis, Sven Dietrich, and Kazue Sako (Eds.), Vol. 7126. Springer, 27–39. https://doi.org/10.1007/978-3-642-29889-9_4

[28] Neal Koblitz. 1990. Constructing Elliptic Curve Cryptosystems in Characteristic 2. In *Advances in Cryptology - CRYPTO '90, 10th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1990, Proceedings (Lecture Notes in Computer Science)*, Alfred Menezes and Scott A. Vanstone (Eds.), Vol. 537. Springer, 156–167. https://doi.org/10.1007/3-540-38424-3_11

[29] Adam Langley, Mike Hamburg, and Sean Turner. 2016. *Elliptic Curves for Security.* RFC 7748. RFC Editor. 1–22 pages. https://doi.org/10.17487/RFC7748

[30] Chae Hoon Lim and Pil Joong Lee. 1997. A Key Recovery Attack on Discrete Log-based Schemes Using a Prime Order Subgroup. In *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings (Lecture Notes in Computer Science)*, Burton S. Kaliski Jr. (Ed.), Vol. 1294. Springer, 249–263. https://doi.org/10.1007/BFb0052240

[31] Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2019. Verifying Arithmetic in Cryptographic C Programs. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019.* IEEE, 552–564. https://doi.org/10.1109/ASE.2019.00058

[32] Victor S. Miller. 1985. Use of Elliptic Curves in Cryptography. In *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings (Lecture Notes in Computer Science)*, Hugh C. Williams (Ed.), Vol. 218. Springer, 417–426. https://doi.org/10.1007/3-540-39799-X_31

[33] Nicky Mouha and Christopher Celi. 2020. Extending NIST's CAVP Testing of Cryptographic Hash Function Implementations. In *Topics in Cryptology - CT-RSA 2020 - The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24-28, 2020, Proceedings (Lecture Notes in Computer Science)*, Stanislaw Jarecki (Ed.), Vol. 12006. Springer, 129–145. https://doi.org/10.1007/978-3-030-40186-3_7

[34] Yoav Nir, Simon Josefsson, and Manuel Pégourié-Gonnard. 2018. *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier.* RFC 8422. RFC Editor. 1–34 pages. https://doi.org/10.17487/RFC8422

[35] Andy Polyakov, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2018. Verifying Arithmetic Assembly Programs in Cryptographic Primitives (Invited Talk). In *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China (LIPIcs)*, Sven Schewe and Lijun Zhang (Eds.), Vol. 118. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:16. https://doi.org/10.4230/LIPIcs.CONCUR.2018.4

[36] Vladimir Popov, Serguei Leontiev, and Igor Kurepkin. 2006. *Additional Cryptographic Algorithms for Use with GOST 28147-89, GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms.* RFC 4357. RFC Editor. 1–51 pages. https://doi.org/10.17487/RFC4357

[37] Thomas Pornin. 2013. *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA).* RFC 6979. RFC Editor. 1–79 pages. https://doi.org/10.17487/RFC6979

[38] Joost Renes, Craig Costello, and Lejla Batina. 2016. Complete Addition Formulas for Prime Order Elliptic Curves. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I (Lecture Notes in Computer Science)*, Marc Fischlin and Jean-Sébastien Coron (Eds.), Vol. 9665. Springer, 403–428. https://doi.org/10.1007/978-3-662-49890-3_16

[39] Eric Rescorla. 2018. *The Transport Layer Security (TLS) Protocol Version 1.3.* RFC 8446. RFC Editor. 1–160 pages. https://doi.org/10.17487/RFC8446

[40] Stanislav V. Smyshlyaev, Evgeny Alekseev, Igor Oshkin, Vladimir Popov, Serguei Leontiev, Vladimir Podobaev, and Dmitry Belyavsky. 2016. *Guidelines on the Cryptographic Algorithms to Accompany the Usage of Standards GOST R 34.10-2012 and GOST R 34.11-2012.* RFC 7836. RFC Editor. 1–32 pages. https://doi.org/10.17487/RFC7836

[41] Nicola Tuveri and Billy Bob Brumley. 2019. Start Your ENGINEs: Dynamically Loadable Contemporary Crypto. In *2019 IEEE Cybersecurity Development, SecDev 2019, Tysons Corner, VA, USA, September 23-25, 2019.* IEEE, 4–19. https://doi.org/10.1109/SecDev.2019.00014

[42] Nicola Tuveri, Sohaib ul Hassan, Cesar Pereida García, and Billy Bob Brumley. 2018. Side-Channel Analysis of SM2: A Late-Stage Featurization Case Study. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018.* ACM, 147–160. https://doi.org/10.1145/3274694.3274725

[43] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1789–1806. https://doi.org/10.1145/3133956.3134043