Anamorphic Cryptography using Baby-Step Giant-Step Recovery

William J. Buchanan[1], Jamie Gilchrist[1]

Blockpass ID Lab, Edinburgh Napier University, Edinburgh.

Abstract. In 2022, Persianom, Phan and Yung outlined the creation of Anamorphic Cryptography. With this, we can create a public key to encrypt data, and then have two secret keys. These secret keys are used to decrypt the cipher into different messages. So, one secret key is given to the Dictator (who must be able to decrypt all the messages), and the other is given to Alice. Alice can then decrypt the ciphertext to a secret message that the Dictator cannot see. This paper outlines the implementation of Anamorphic Cryptography using ECC (Elliptic Curve Cryptography), such as with the secp256k1 curve. This gives considerable performance improvements over discrete logarithm-based methods with regard to security for a particular bit length. Overall, it outlines how the secret message sent to Alice is hidden within the random nonce value, which is used within the encryption process, and which is cancelled out when the Dictator decrypts the ciphertext. It also shows that the BSGS (Baby-step Giant-step) variant significantly outperforms unoptimised elliptic curve methods.

1 Introduction

In cybersecurity, we can use anamorphic cryptography to change the viewpoint of a cipher [1]. With this, we assume that we have a dictator who will read all of our encrypted data, and will thus have a secret key of sk. The dictator (Mallory) will arrest anyone who sends secret messages that they cannot read. For this, Bob can construct a dual-decryption system, providing sk_0 to the Dictator and sk_1 to Alice (or Alice can supply Bob with one she chooses as sk_1 herself). As far as Mallory knows, he has the only key for the ciphertext, and outwardly the encrypted data appears standard (i.e has no difference in structure to that of a normal encrypted message). This paper outlines a new method of creating anamorphic cryptography using Baby-Step Giant-Step Recovery to provide a fast recovery method for Alice, and which improves on existing approaches.

2 Related work

An implementation of the ElGamal method for anamorphic cryptography is given in [2]. While discrete logarithm methods have been used to implement anamorphic cryptography [3], they tend to be slow in their operation. Dodis et al. [4] liken the approach of anamorphic cryptography to the addition of backdoors into semantically secure schemes, where entities might be forced to hand over their decryption keys.

David C. Wyld et al. (Eds): CRYPIS, CBIoT, CAIML, NLCA, NC, WiMo, ICAIT, ICDIPV, ITCSE – 2025
pp. 59-71, 2025. CS & IT - CSCP 2025 DOI: 10.5121/csit.2025.151704

This technique offers the ability to send covert data or to later claim plausible deniability for actions. Carnemolla et al. [5] have outlined that there are certain schemes which are *anamorphic resistant* and that, in some cases, anamorphic encryption is similar to substitution attacks. The methods that support anamorphic encryption include RSA-OAEP, Pailler, Goldwasser-Micali, ElGamal schemes, Cramer-Shoup, and Smooth Projective Hash-based systems.

Chu et al. [6] investigated cases where the dictator is involved in key generation, and where it is still possible to implement anamorphic communication. This involves the usage of threshold signature schemes, and the adversary is included within the signing group.

3 Method

This paper outlines the integration of ElGamal methods [7] with ECC for the implementation of anamorphic encryption. With anamorphic encryption, we can have a public key of pk and two private keys of $sk0$ and $sk1$. Bob can then have two messages of:

$$m_0 = \text{"I love the Dictator"} \tag{1}$$

$$m_1 = \text{"I hate the Dictator"} \tag{2}$$

Bob then encrypts the two messages with the public key(the PK_0 of the Dictator):

$$CT = Enc(pk, m_0, msg_1) \tag{3}$$

The Dictator will then decrypt with sk_0 and reveal the first message:

$$Dec(sk_0, CT) \rightarrow m_0 \tag{4}$$

Alice will decrypt with her key and reveal the second message:

$$Dec(sk_1, CT) \rightarrow m_1 \tag{5}$$

And, so, the Dictator thinks that they can decrypt the message, and gets, “I love the Dictator”. Alice, though, is able to decrypt the ciphertext to a different message of “I hate the Dictator”.

3.1 ElGamal encryption

With ElGamal encryption using elliptic curves [8], Alice generates a private key (x) and a public key of:

$$Y = x.G \tag{6}$$

and where G is the base point on the curve. She can share this public key (Y) with Bob. When Bob wants to encrypt something for Alice, he generates a random value (r) and the message value (M) and then computes:

$$C_1 = r.G \tag{7}$$
$$C_2 = r.Y + M \tag{8}$$

To decrypt, Alice takes her private key (x) and computes:

$$M = C_2 - x.C_1 \tag{9}$$

This works because:

$$M = C_2 - y.C_1 = r.x.G + M - x.r.G = M \tag{10}$$

Figure 1 outlines how Bob can encrypt data for Alice.

3.2 Anamorphic Encryption with elliptic curves

First, we pick a curve, such as secp256k1, which has a base point of G. Bob can then pick a secret key for the Dictator of $sk_{Dictator}$. The public key is then:

$$pk = sk_{Dictator}.G \tag{11}$$

Bob then generates a random scalar value of t and takes the secret message of cm, and produces:

$$r = cm + t \tag{12}$$

The value of t will be Alice's secret key. To encrypt the message of m, Bob uses:

$$rY = r.pk \tag{13}$$
$$rG = r.G \tag{14}$$
$$rYval = \text{Int}(rY) \tag{15}$$
$$c0 := rYval + M \tag{16}$$
$$c1 := rG \tag{17}$$
$$\tag{18}$$

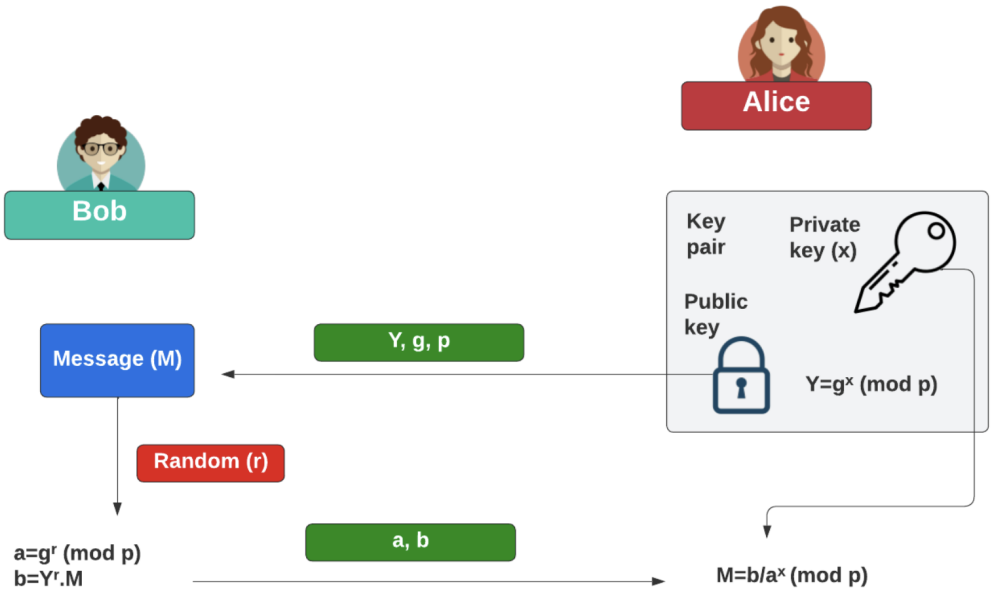

Fig. 1. ElGamal encryption with ECC

The cipher is the (c_0, c_1). To decrypt by the Dictator:

$$yC = sk_{Dictator}.c1 \tag{19}$$
$$yC_{val} = \text{Int}(yC) \tag{20}$$
$$res_{Dictator} = c_0 - yC_{val} \tag{21}$$
$$\tag{22}$$

Alice can then decrypt with her key (t):

$$t_c = t.G \tag{23}$$
$$res_{Alice} = c_1 - t_c \tag{24}$$

Alice will then search through the possible values of res_{Alice} to find the value of cm that matches the elliptic curve point. This works because:

$$res_{Alice} = c_1 - tc = r.G - t.G = r.G - (r - cm).G = cm.G \tag{25}$$

It is worth mentioning that both the t and tc for Alice must be treated as secret. This is because anyone who has access to the Public key and to the $c1$ component of the ciphertext can do:

Unlike traditional Public Key Cryptography, in the case of Alice, both the private key t and the public key (t_c) must be treated as secrets. This is in contrast to normal PKI, where a public key can be shared safely. The reason for this is that anyone with Alice's public key, can then also brute force the cm. They can do :

$$cm \cdot G = c_1 - tc \tag{26}$$

Given $cm \cdot G$, recovering cm reduces to solving a discrete logarithm problem:

$$cm = \log_G(cm \cdot G) \tag{27}$$

However, although the ECDLP is considered computationally difficult for large ranges, the system intentionally restricts cm to a low-bit space (e.g., 30 bits) so it can be recovered in a reasonable time on lower-performance hardware. This small search space then makes it feasible for anyone with t_c and c_1 to brute-force the remaining 30 bits of entropy. Some sample code and a test run are given in the appendix.

3.3 Baby-Step Giant-Step

The Baby-Step-Giant-Step solves for x in $h = g^x \pmod p$ [9], or will solve x for an elliptic curve point of $x.G$ Within normal logarithms, we define:

$$h = a^x \tag{28}$$

So if we want to find the value of x, we use:

$$x = log_a(h) \tag{29}$$

So 10^4 is 10,000, and the inverse log is $log_10(10,000)$ is 4.

Within discrete logarithms, we introduce a finite field with a prime number. For this, we have:

$$h = g^x \pmod p \tag{30}$$

and where p is the prime number. It is thus a difficult task to find the value of x which has been used, even if we know h, g and p. We utilise discrete logarithms in the Diffie-Hellman key exchange method and ElGamal encryption.

Let's start with an example [10]:

$$20 = 5^x \pmod{53} \tag{31}$$

In this case, we have $g = 5$, $h = 20$ and $p = 53$, and want to find x. We first determine the square root of $p - 1$, and we will round it up to the nearest integer. In this case, it will be:

$$N = Ceiling(\sqrt{p-1}) = Ceiling(\sqrt{52}) = 7 \tag{32}$$

Next, we will pre-compute from 1 to N with the baby table. These will store $g^i \pmod p$ and then store them in the form of:

$$g^i \pmod p, i \tag{33}$$

To give:

$$1:0, 3:7, 5:1, 51:5, 42:4, 43:6, 19:3, 25:2 \tag{34}$$

For example, if $i = 0$, we get $5^0 \pmod{53}$ gives 1 {1:0}. For $i = 1$ we get $5^1 \pmod{53}$ gives 5 {5:1}. For $i = 2$ we get $5^2 \pmod{53}$ gives 5 {25:2}. We now have a list of pairs from 0 to the square root of $p - 1$, and now compute g^{-N} to give c:

$$c = g^{N(p-2)} \pmod p \tag{35}$$

We then search through values of:

$$hc^x \pmod p \tag{36}$$

until we find a match in the table. We then take this value and multiply it by N and add the value we have found:

```
for j in range(N):
 y = (h * pow(c, j, p)) % p
 if y in t:
   return j * N + t[y]
```

3.4 Application to Anamorphic Cryptography

Anamorphic cryptography offers a compelling approach to covert communication and plausible deniability, but it does come with specific security considerations. In traditional public key cryptography, the public key can be treated as such, public. However, in this scheme, Alice's public and private keys (t and tc must be treated as secrets. These can effectively be seen as 'dual keys', where each must be safeguarded from disclosure to other parties. This is because an attacker who intercepts the ciphertext (c_0, c_1) and learns $t \cdot G$ (tc) can compute:

$$cm = \log_G(c_1 - t_c) \tag{37}$$

This works because cm is small (30 bits for example), and thus Alice is able to brute-force or Baby-Step Giant-Step cm in a reasonable time on commodity hardware.

For anyone else who does not know t or t_c, computing cm from c_1 is equivalent to solving the elliptic curve discrete logarithm problem(ECDLP). If t_c is secret, and the size of cm is small (even just two bits), without t or t_c, the attacker is left with only $c_1 = r \cdot G$, which is completely infeasible within modern computing.

In this setup, both t and $t_c = t \cdot G$ must be kept private. If tc leaks, anyone with access to c_1 can compute $c_1 - t_c = cm \cdot G$. Since cm is intentionally small (e.g., 30 bits), this makes brute-force recovery viable. In effect, t_c acts like a second private key — and its exposure would compromise the covert message. Thus, in the conventional sense, the keys(either t or t_c) Alice holds are both private keys.

3.5 Implementation

Now we will outline the implementation of our solution. Algorithm 1 outlines the method of applying anamorphic using ECC encryption, and Algorithm 2 defines the method for the Dictator to decrypt. Algorithm 3 provides the method for Alice to decrypt the message that is passed to her. With this, Alice will reveal the covert message, and which cannot be seen by the Dictator.

Algorithm 1 Anamorphic ECC Encryption

Require: Public key $pk = sk_0 \cdot G$, cover message m_0, covert message $cm \in \mathbb{Z}$, Alice's secret key $t \in \mathbb{Z}$

Ensure: Ciphertext (c_0, c_1)

1: $r \leftarrow cm + t$
2: $r_G \leftarrow r \cdot G$
3: $r_Y \leftarrow r \cdot pk$
4: $r_Y^{\text{val}} \leftarrow$ Integer encoding of r_Y
5: $c_0 \leftarrow r_Y^{\text{val}} + m_0$
6: $c_1 \leftarrow r_G$
7: **return** (c_0, c_1)

3.6 Key Derivation and Use as a Shared Secret

An alternative application can involve treating the result of Alice's decryption as a shared elliptic curve point:

Algorithm 2 Decryption by Dictator

Require: Secret key sk_0, ciphertext (c_0, c_1)
Ensure: Decrypted message m_0
1: $y_C \leftarrow sk_0 \cdot c_1$
2: $y_C^{\text{val}} \leftarrow$ Integer encoding of y_C
3: $m_0 \leftarrow c_0 - y_C^{\text{val}}$
4: **return** m_0

Algorithm 3 Covert Decryption by Alice

Require: Alice's secret key t, ciphertext component c_1, generator G
Ensure: Covert message cm
1: $tc \leftarrow t \cdot G$
2: $res \leftarrow c_1 - tc$ ▷ Now $res = cm \cdot G$
3: Search (e.g., brute-force or BSGS) for cm such that $cm \cdot G = res$
4: **return** cm

$$res_{Alice} = c_1 - t \cdot G = cm \cdot G \tag{38}$$

This point can act as a symmetric secret index or be hashed to derive encryption keys, enabling integration with secure storage systems or encrypted messaging layers.

3.7 Secrecy of Alice's Public Key

It is critical that Alice's scalar t and corresponding $t \cdot G$ remain confidential. If an adversary gains access to the latter, they can compute:

$$cm \cdot G = c_1 - t \cdot G \Rightarrow cm = \log_G(cm \cdot G) \tag{39}$$

Given that cm is intentionally small (e.g., 30-bit), this discrete log is trivially solvable, compromising the covert message. Therefore, $t \cdot G$ must be treated as secret in this scheme, contrary to standard public key conventions.

4 Evaluation

In order to benchmark the performance and practical efficiency of the anamorphic system, we have run a series of tests on a Windows PC with an Intel i7 3770, 3.4 GHz with 16 GB of DDR3 RAM, where we compare both the DLP and ECDLP variants, while also showing optimisations. We then solve for cm of varying sizes up to 34 bits. For the unoptimised version of both the DLP and ECDLP variants, tests were done up to 20 bits only due to the time taken. For optimised versions, we tested up to the full 34 bits in both cases.

Each script used in testing is included in the Appendix (Section 6).

– **Vanilla-DLP**: A non-optimised discrete logarithm (DLP) based implementation using naive scalar multiplication.
– **ECC-DLP-Vanilla**: A basic elliptic curve variant using a similar brute-force search for the scalar.
– **BSGS-DLP**: An optimised DLP implementation leveraging the Baby-Step Giant-Step (BSGS) algorithm.
– **ECCDLP-BSGS**: An optimised elliptic curve implementation using BSGS with curve precomputations.

Table 1: Alice Decryption Time vs. Message Index (cm) for Different Schemes

Scheme	**cm**	**Decryption Time (ms)**
Vanilla-DLP	9	6.9701
Vanilla-DLP	99	7.5207
Vanilla-DLP	999	6.1685
Vanilla-DLP	9,999	63.6082
Vanilla-DLP	99,999	1700.6238
Vanilla-DLP	999,999	27769.2745
ECC-DLP-Vanilla	9	6.5441
ECC-DLP-Vanilla	99	42.0362
ECC-DLP-Vanilla	999	429.0826
ECC-DLP-Vanilla	9,999	4,280.2626
ECC-DLP-Vanilla	99,999	42,637.5404
ECC-DLP-Vanilla	999,999	442,947.4792
BSGS-DLP	9	4.6448
BSGS-DLP	99	6.9718
BSGS-DLP	999	8.4206
BSGS-DLP	9,999	7.9787
BSGS-DLP	99,999	12.5426
BSGS-DLP	999,999	14.5983
BSGS-DLP	9,999,999	12.0785
BSGS-DLP	99,999,999	6.8217
BSGS-DLP	999,999,999	11.2848
BSGS-DLP	9,999,999,999	10.1387
ECCDLP-BSGS	9	1.5771
ECCDLP-BSGS	99	4.5974
ECCDLP-BSGS	999	10.1518
ECCDLP-BSGS	9,999	47.3780
ECCDLP-BSGS	99,999	145.2986
ECCDLP-BSGS	999,999	458.9628
ECCDLP-BSGS	9,999,999	1,487.1940
ECCDLP-BSGS	99,999,999	4,693.6531
ECCDLP-BSGS	999,999,999	14,142.6721
ECCDLP-BSGS	9,999,999,999	43893.8668

The ECCDLP-BSGS variant significantly outperforms its unoptimised elliptic curve counterpart (ECC-DLP-Vanilla), reducing decryption time by over three or-

ders of magnitude for large *cm* values. The DLP variants also benefit markedly from BSGS optimisation, with decryption times reduced from tens of seconds to milliseconds. Interestingly, the optimised **BSGS-DLP** implementation outperforms the **ECCDLP-BSGS** variant in raw decryption time across most *cm* values, owing to faster integer arithmetic compared to elliptic curve point operations.

5 Conclusions

While RSA-OAEP, Pailler, Goldwasser-Micali, ElGamal schemes, Cramer-Shoup, and Smooth Projective Hash-based systems all support anamorphic cryptography, the usage of elliptic curve methods provides an opportunity to enhance the overall performance of the methods implemented for the ElGamal technique. The results show that the ECCDLP-BSGS variant significantly outperforms unoptimised elliptic curve methods for anamorphic cryptography.

References

1. G. Persiano, D. H. Phan, and M. Yung, "Anamorphic encryption: private communication against a dictator," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2022, pp. 34–63.
2. F. Banfi, K. Gegier, M. Hirt, U. Maurer, and G. Rito, "Anamorphic encryption, revisited," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2024, pp. 3–32.
3. fbanfi90, "robust-anamorphic-encryption/elgamal.py at main · fbanfi90/robust-anamorphic-encryption — github.com," https://github.com/fbanfi90/robust-anamorphic-encryption/blob/main/elgamal.py, [Accessed 21-04-2025].
4. Y. Dodis and E. Goldin, "Anamorphic-resistant encryption; or why the encryption debate is still alive," *Cryptology ePrint Archive*, 2025.
5. D. Carnemolla, D. Catalano, E. Giunta, and F. Migliaro, "Anamorphic resistant encryption: the good, the bad and the ugly," *Cryptology ePrint Archive*, 2025.
6. H. Chu, K. Do, L. Hanzlik, and S. A. Thyagarajan, "When threshold meets anamorphic signatures: What is possible and what is not!" *Cryptology ePrint Archive*, 2025.
7. T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE transactions on information theory*, vol. 31, no. 4, pp. 469–472, 1985.
8. W. J. Buchanan, "Elgamal ecc encryption (using message string)," https://asecuritysite.com/elgamal/elgamal02_str, Asecuritysite.com, 2025, accessed: April 21, 2025. [Online]. Available: https://asecuritysite.com/elgamal/elgamal02_str
9. D. Shanks, "Class number, a theory of factorization, and genera," in *Proc. Symp. Math. Soc., 1971*, vol. 20, 1971, pp. 415–440.
10. W. J. Buchanan, "Baby-step-giant-step," https://asecuritysite.com/principles/baby, Asecuritysite.com, 2025, accessed: July 09, 2025. [Online]. Available: https://asecuritysite.com/principles/baby

6 Appendix

This appendix outlines the main algorithms used within the anamorphic ECC encryption system described in the paper. Full source code and command line tools

that support this implementation are available at:
`https://github.com/billbuchanan/babydictator`.

6.1 Source Code and Tools

The full implementation is provided in the companion GitHub repository. Tools include:

- `keygen.go` – Generates key pairs for both the Dictator and Alice
- `encrypt.go` – Encrypts both cover and covert messages
- `decrypt_dictator.go` – Decrypts the cover message using sk_0
- `decrypt_alice.go` – Recovers the covert message using Alice's key

The repository also includes benchmarking scripts and a chatbot interface to demonstrate how covert communication works in real-time using this proof of concept (POC).

6.2 Application Implementation

To support further experimentation, testing and real-time demo, we have implemented a full-stack application for Anamorphic Encryption. This consists of Go CLI tools, a Python API wrapper, and a Chatbot front-end. Collectively, these represent a messenger service, where covert data can be hidden in the ciphertext.

6.3 Command-Line Tools (CLI)

The core cryptographic logic is implemented in Go and exposed through standalone CLI binaries. These tools form the computational backend and include:

- `keygen.go` — Generates Dictator and Alice key pairs
- `encrypt.go` — Performs anamorphic encryption with standard and covert message support
- `decrypt_dictator.go` — Decrypts messages with the Dictator's key
- `decrypt_alice.go` — Recovers covert message cm using Alice's key and optional BSGS optimisation

Each tool supports flexible inputs (via flags), and all test cases used for benchmarking (see Section 1), other than the DLP-based tests, were produced using these scripts.

6.4 REST API Layer

A Python Flask-based web API wraps the Go CLI binaries, exposing endpoints for key generation, encryption, and both decryption flows. This API layer supports:

- Stateless operation with ephemeral or uploaded keys
- Endpoint support for JSON-based requests:
  - `POST /api/keygen`
  - `POST /api/encrypt`
  - `POST /api/decrypt-alice`
  - `POST /api/decrypt-dictator`
- Docker-friendly deployment and integration with the frontend

6.5 Repository Access and Scripts

The full source code, CLI tools, Python API server, React frontend, and schema definitions are hosted at: https://github.com/billbuchanan/babydictator

Scripts include:

- CLI: `keygen.go`, `encrypt.go`, `decrypt_alice.go`, `decrypt_dictator.go`
- API: `app.py` wrapping CLI binaries with RESTful endpoints
- Frontend: Chatbot UI for encryption testing and message interpretation
- Examples: Full usage walkthroughs, pre-built binaries, and LLM integration hooks

7 Authors

William (Bill) J Buchanan OBE FRSE is a Professor of Applied Cryptography in the School of Computing, Edinburgh and the Built Environment at Edinburgh Napier University. He is a Fellow of the BCS and a Principal Fellow of the HEA. Bill was appointed an Officer of the Order of the British Empire (OBE) in the 2017 Birthday Honours for services to cybersecurity, and, in 2024, he was appointed as a Fellow of the Royal Society of Edinburgh (FRSE). In 2023, he received the "Most Innovative Teacher of the Year" award at the Times Higher Education Awards 2023 (the "Oscars of Higher Education"), and was awarded "Cyber Evangelist of the Year" at the Scottish Cyber Awards in 2016 and 2025.

Jamie Gilchrist is a self-taught cryptography and cybersecurity enthusiast with over 20 years of professional experience in the IT industry. His research interests include applied cryptography, decentralised systems, identity solutions and the intersection of machine learning, privacy and data security.