An extended abstract of this article appears in the proceedings of CCS 2021. This is the full version and available in the IACR eprint archive.

On the (in)security of ElGamal in OpenPGP

Luca De Feo*
IBM Research Europe – Zurich
Rüschlikon, Switzerland

Bertram Poettering*
IBM Research Europe – Zurich
Rüschlikon, Switzerland

Alessandro Sorniotti*
IBM Research Europe – Zurich
Rüschlikon, Switzerland

ABSTRACT

Roughly four decades ago, Taher ElGamal put forward what is today one of the most widely known and best understood public key encryption schemes. ElGamal encryption has been used in many different contexts, chiefly among them by the OpenPGP standard. Despite its simplicity, or perhaps because of it, in reality there is a large degree of ambiguity on several key aspects of the cipher. Each library in the OpenPGP ecosystem seems to have implemented a slightly different "flavour" of ElGamal encryption. While –taken in isolation– each implementation may be secure, we reveal that in the interoperable world of OpenPGP, unforeseen *cross-configuration attacks* become possible. Concretely, we propose different such attacks and show their practical efficacy by recovering plaintexts and even secret keys.

CCS CONCEPTS

• **Security and privacy → Public key encryption**.

KEYWORDS

OpenPGP, ElGamal encryption, side-channel attacks, key-recovery attacks, modular exponentiation

1 INTRODUCTION

The ElGamal cryptosystem [14] is one of the oldest and best-known public key encryption schemes. In the 80's and 90's it earned wide adoption for being simultaneously efficient and patent-free. Its most prominent use is arguably as part of OpenPGP [12], a standard aimed to promote consumable, interoperable email security, where it has been the default and most popular encryption option for decades [30]. While the change in patent status of RSA encryption slightly reduced its popularity, at the time of writing, still at least 1 in 6 registered OpenPGP keys have an ElGamal subkey [3], with about a 1,000 new registrations per year.

The ElGamal scheme builds on elegant mathematical structures and can be defined very compactly. This simplicity, together with the opportunity to mature for roughly four decades now, suggests that a crisp specification with clear parameter choices, rules, and algorithms would be present in international standards, in particular in OpenPGP. Surprisingly, this turns out *not* to be the case: our research reveals that OpenPGP's understanding of ElGamal encryption is open to interpretation, with several choices subject to the discretion of the implementer.

In this article we consider *cross-configuration attacks* on OpenPGP. Such attacks emerge when different interpretations ('configurations') of the same standard interact insecurely with each other. Towards identifying such conditions for ElGamal encryption, we need to first understand the universe of OpenPGP interpretations that are used in practice. We approach this challenge from various angles: we carefully study RFC4880 [12] which defines OpenPGP, we inspect the source code of three relevant OpenPGP-implementing software libraries (the `Go` standard library, `Crypto++`, and `gcrypt`), and we conduct a large-scale examination of millions of keys registered on OpenPGP key servers.

Our results reveal an insecure posture. For instance, we develop and prototype a plaintext recovery attack that can be mounted on ciphertexts produced by the ubiquitous GNU Privacy Guard (and other implementations, e.g. `Crypto++`) against keys generated following the original ElGamal specification [14]. The attack is effective against 2048 bit keys, which are considered secure at the time of writing. Our OpenPGP key server analysis reveals that more than 2,000 OpenPGP users are currently exposed.[1] We further illustrate how cross-configuration attacks can be combined with known side-channel exploitation techniques like FLUSH+RELOAD [38] or PRIME+PROBE [33]. One of our targets is the ElGamal implementation of `gcrypt`, the cryptographic library used by the GNU Privacy Guard. Interestingly, `gcrypt` has already been fixed twice after seminal work [21, 38] on side-channel attacks identified weaknesses. Concretely, by conducting an end-to-end attack we show that if a 2048 bit ElGamal key generated by `Crypto++` is used by `gcrypt` to decrypt a ciphertext, then an attacker that is OS- or VM-colocated with the decrypter can fully recover the decryption key.

Given that interoperability is the explicit and almost exclusive goal of any standardization effort, and commonplace in the OpenPGP world, we conclude that our attack conditions are as realistic as the attack results awakening. Our research is timely since a new version of the OpenPGP standard is currently being discussed [18]; we hope that our findings will influence that discussion.

This manuscript is organised as follows: In Section 2 we survey (a) the meaningful options available when implementing ElGamal encryption, (b) the options adopted by the `Go`, `Crypto++`, and `gcrypt` libraries, and (c) the options picked by over 800,000 users in practice (as far as reflected on key server databases); we also report on further interesting findings from our key server crawl. In Section 3 we recall various standard algorithms for solving discrete logarithms. In Section 4 we describe "vanilla" cross-configuration attacks, and in Section 5 we describe those combined with side-channel attacks. In Section 6 we conduct end-to-end exploits and describe how we bring in the required side-channel information. We conclude in Section 7.

1.1 Related Work

Since ElGamal encryption was first proposed [14], research efforts were both steered towards formally confirming its security (e.g. via reductions to the DDH problem [34]) and to shed light on its insecurities (e.g. when used in its textbook form [10]). CVE-2018-6829 [2]

*{feo, poe, aso}@zurich.ibm.com

[1] We found that at most a small fraction of ElGamal keys is formed according to the original specification of [14]; otherwise, more OpenPGP users would be affected.

and CVE-2018-6594 [1] highlight corresponding issues in one of the libraries that we investigate (`gcrypt`). However, OpenPGP employs ElGamal encryption exclusively for key transport to achieve hybrid encryption, and in this case the attacks do not seem to apply. To the best of our knowledge, our work is the first to challenge the security of ElGamal encryption in the specific way OpenPGP builds and relies on it. (Early versions of OpenPGP also relied on the known to be insecure ElGamal *signatures* [9]; the weaknesses of the latter do not suggest weaknesses of ElGamal encryption.)

Several works on OpenPGP focus on Web of Trust aspects [7, 13, 35], on OpenPGP's use of other cryptographic primitives such as modes of operation of symmetric ciphers [25, 27], or on the security of passphrases protecting keyrings [24]. An analysis of keys collected from OpenPGP servers was performed in [30]. In contrast to our work (see Section 2), [30] does not distinguish between different flavours of ElGamal parameters and keys.

Modular exponentiation has been the subject of extensive research to protect it from side-channel attacks [4, 15, 17, 23]. Specifically cache-based attacks have been explored extensively, using various techniques such as EVICT+TIME [8], PRIME+PROBE [33] or FLUSH+RELOAD [6, 38]. These techniques attempt to observe the micro-architectural traces left by the execution of sensitive cryptographic operation to obtain knowledge about the secret inputs thereto. For example, if a branch instruction is conditional to a secret bit, that bit may be learned if the attacker observes whether the instructions at the target or the fallthrough are present in the cache. Presence or absence from the cache is observed through measurables such as execution time, which is faster if the instructions are cached or slower otherwise. Similar considerations can be made for memory areas that are accessed at secret-dependent offsets: in this case, an attacker may learn whether a specific location was accessed by measuring access time to cache elements whose addresses are congruent to the target location. Yarom and Falkner [38] in particular target the `gcrypt` implementation of modular exponentiation which at the time was using square-and-multiply. Prompted by that work, the implementation was significantly revised. The changes are however not sufficient to protect against the attack we present in Section 5. Most cache-based side-channel attacks target L1 or L2. However, Liu *et al.* [21] show that exploitation based on last-level cache is possible for attacks targeting both data and instructions. The work also targets the `gcrypt` implementation of modular exponentiation and the library has also been fixed to avoid leaking table accesses. Once again however, this is not sufficient to prevent the attack described in Section 5. Other works present alternative cache side-channels (FLUSH+FLUSH [16], S$A [5]) which may be used to perform a practical exploitation of the attacks.

2 ELGAMAL ENCRYPTION

One of the earliest proposals to construct public key encryption is by Taher ElGamal [14]. As a first approximation, the construction is as follows.

☛ Generic ElGamal encryption. Let G be a (multiplicatively written) group and $g \in G$ a generator. To create a key pair (sk, pk), pick a random integer x, compute the group element $X = g^x$, and output $(sk, pk) := (x, X)$. Given pk, to encrypt a message M, pick an ephemeral random integer y, compute the group elements $Y = g^y$ and $Z = X^y = g^{xy}$, and output $C = (C_1, C_2) := (Y, M \cdot Z)$ as the ciphertext. Given sk, to decrypt C, first recover group element Z from C_1 by computing $Z = Y^x = g^{yx}$ and then use C_2, Z to recover $M = C_2/Z$.

The ElGamal encryption scheme, as described above, is not yet fully specified. To complete the specification, the following details have to be fixed: Which group G shall be used? How is generator g chosen, and shall it generate the full group G or just a subgroup? From which sets are exponents x, y picked, and according to which distributions? Multiple configurations for these parameters are possible and promise to lead to correct and secure public key encryption instances.

In the following we describe four such configurations that have in common that G is a 'prime field group', that is, the multiplicative group $\mathbb{Z}_p^\times = (\mathbb{Z}/p\mathbb{Z})^\times$ of a field $\mathbb{Z}/p\mathbb{Z}$ where p is a large prime number (also referred to as the modulus). In such cases the order of G is given by $\mathrm{ord}(G) = p - 1$ and the order of any subgroup $G' \subseteq G$ is an integer divisor of $p - 1$.

☛ Configuration A: "The original". In the original proposal of ElGamal [14], generator g is chosen such that it generates the full group, i.e., cyclically generates a total of $p-1$ elements. Further, the exponents x, y are picked uniformly at random from the interval $[1\,..\,p-1]$. The only condition on p that is formulated in [14] is that $p-1$ have at least one large prime factor. This is to sufficiently weaken the impact of the Pohlig–Hellman algorithm [28] so that it remains infeasible to compute discrete logarithms in G. See Appendix A for the recommendations in terms of size of p and of its large prime factor.

Note that if keys are generated according to Configuration A, then any element in G could become a public key. However, some of these public keys would have a considerably lower multiplicative order than others and thus promise less security. To see this, consider that picking the secret key $x = (p-1)/2$, i.e., the public key $X = g^{(p-1)/2} = -1$, leads to $Z \in \{+1, -1\}$ for any y, meaning that encrypted messages are easy to recover from their ciphertexts. In general, if the indications of [14] are followed verbatim, many more similarly weak low-order public keys exist. This can be prevented by restricting the ElGamal group operations to a subgroup $G' \subsetneq G$ such that G' has prime order. Indeed, if $G' = \langle g \rangle$ has prime order, then all elements generated by g necessarily have the same order as g (with the one exception of the neutral element which is easily avoided or tested for).

Let $p - 1 = q_0 \cdots q_n$ be the prime factor decomposition of $\mathrm{ord}(G)$. As p is a large prime number and hence odd, we know that one of these prime factors is 2, and we hence w.l.o.g. write $p-1 = 2q_1 \cdots q_n$. Now, for any prime $q = q_i$ in this list of factors there exists a subgroup $G' \subsetneq G$ of order q. The idea of using such groups G' in cryptography goes back to Schnorr [31].

☛ Configuration B: "ElGamal over Schnorr groups". Pick a large prime group order q and a prime modulus p such that $q \mid (p-1)$, choose a generator γ with $\langle \gamma \rangle = G$ and let $g = \gamma^{(p-1)/q}$ and $G' = \langle g \rangle$. Note that this implies $\mathrm{ord}(G') = q$. Pick exponents x, y in the interval $[1\,..\,q-1]$. Note that the condition on p (at least one large prime factor) is already satisfied by the choice of p and q.

Note that Configuration B not only removes the described issues related to small subgroups, but it also allows for more efficient

implementations than Configuration A. This is so because the exponents x, y are now picked from the interval $[1\,..\,q-1]$ instead of the typically much larger interval $[1\,..\,p-1]$, which in general leads to a significant efficiency gain for exponentiation operations.

A further advantage achieved by Configuration B is that the prime-order subgroup setting is considerably easier tractable in the framework of provable security. Indeed, virtually all textbook formalizations of the ElGamal-related DDH problem assume prime-order groups and would not be sufficient to make a formal statement about the security of Configuration A.

The small subgroup issues removed by Configuration B are relevant when ElGamal is operated according to specification, i.e., in attack settings that involve only passive adversaries. If we also admit active adversaries, then additional issues arise:[2] Consider a communication flow between two honest parties and assume an active adversary that intercepts a ciphertext $C = (C_1, C_2) = (Y, M \cdot Z)$ and re-injects it as $C' = (C'_1, C'_2) \coloneqq (-C_1, C_2)$. When decrypting C', the receiver will compute value $Z' = (-Y)^x = (-1)^x Y^x = (-1)^x Z$ and use it to recover message $M' = C'_2/Z' = (M \cdot Z)/(-1)^x/Z$. Note that if x is an even number then the decryption succeeds with the correct message $M' = M$, while if x is an odd number then the decryption results in a wrong message $M' \neq M$. As parties will react differently to correct (meaningful) and incorrect (effectively random) messages, more often than not it is observable from the outside whether a decryption is successful or not, and in such cases the ciphertext manipulation leaks one bit of the secret key x to the adversary.

The following generalization of the attack can be used to leak more than a single bit of the secret key. It is based on the observation that for any t with $t \mid (p-1)$ there exists a group element α of order t, i.e., with $\alpha^t = 1$ and $1 \notin \{\alpha, \alpha^2, \ldots, \alpha^{t-1}\}$. Fix such a pair (t, α), let $C = (C_1, C_2)$ be an honestly generated ciphertext as above, and assume an adversary that intercepts C and re-injects it as $C' = (C'_1, C'_2) \coloneqq (\alpha C_1, \alpha^h C_2)$, where $h \in [1\,..\,t]$. When processing C', the decrypter will obtain the message $M' = C'_2/Z' = (\alpha^h C_2)/(\alpha^x C_1^x) = \alpha^{h-x} M$. Note that $M' = M$ iff $\alpha^{h-x} = 1$ iff $h \equiv x \pmod t$, while otherwise $M' \neq M$. That is, by testing whether the injected ciphertext C' properly decrypts, the adversary can confirm or refute a guess on $x \pmod t$. Recovering $x \pmod{t_1}$, $x \pmod{t_2}$, ... for carefully picked values $t_1, t_2, \ldots$ allows recovering the full exponent x via the Chinese Remainder Theorem. Note that for each t-value the adversary has to request up to t decryptions, so that small t-values are a precondition for the attack to be effective.

To prevent the attack, two countermeasures are immediate: one is to ensure that the number of distinct integer divisors t of $p-1$ is very small, while the other is to arrange that all admissible t-values are very large. Configurations C and D formalize these ideas; the techniques can be traced back to Pollard [29] and Lim and Lee [20], respectively.[3]

☛ Configuration C: "ElGamal over safe primes". This is a refinement of Configuration B where the number of prime factors of $p-1$ is reduced to the minimum and thus prime numbers p, q have almost the same size: Pick a large prime group order q and a prime modulus p such that $p - 1 = 2q$, choose a generator γ with $\langle\gamma\rangle = G$ and let $g = \gamma^{(p-1)/q} = \gamma^2$ and $G' = \langle g\rangle$. Choose exponents x, y in the interval $[1\,..\,q-1]$.

Note that, in this configuration, choosing $g = 4 = (\pm 2)^2$ is always feasible, leading to a reduced public key size when compared to Configuration B. On the other hand, exponents x, y became large again, which negatively impacts exponentiation performance. In practice, some implementations hence work with 'short exponents' despite the group order q being considerably larger.

☛ Configuration D: "ElGamal over Lim–Lee primes". This is a refinement of Configuration B where all prime factors of $(p-1)/2$ are chosen to be of roughly the same size: Pick a prime modulus p such that if $p - 1 = 2q_1 \cdots q_n$ is a prime factorization and we let $q \coloneqq q_1$ then for all $q_i \in \{q_2, \ldots, q_n\}$ we have $q_i \approx q$; in particular, computing discrete logarithms shall be hard modulo all factors of $(p-1)/2$. Given such a setting, choose a generator γ with $\langle\gamma\rangle = G$ and let $g = \gamma^{(p-1)/q}$ and $G' = \langle g\rangle$. Choose exponents x, y in the interval $[1\,..\,q-1]$.

Note that both Configurations C and D effectively protect against small subgroup attacks in active attack settings, but while Configuration C allows for smaller public keys, Configuration D promises more efficient exponentiations. When building (or standardizing) a secure system from scratch, it seems advisable to use one of these two options. Unfortunately, as we will see next, the OpenPGP standard does not give the same suggestion.

2.1 ElGamal in OpenPGP

A widely-deployed cryptography standard that suggests using, and mandates implementing, ElGamal encryption is OpenPGP. While its first version was put forward in 1998 as RFC2440, the latest official version appeared in 2007 and is formalized in RFC4880 [12]. Given the unclarity over what is actually meant by ElGamal encryption, we analyse the RFC4880 document with respect to the precise understanding of ElGamal encryption that it assumes or conveys, paying special attention to the details connected to parameter and key generation, and to the encryption and decryption operations.

In [12, Sect. 9.1], two references to where to find specifications of the ElGamal encryption algorithms are given. The one is to ElGamal's original paper [14], and the other is to the Handbook of Applied Cryptography [22]. We note that the latter has two specifications of ElGamal encryption: [22, Sect. 8.4.1] describes the original scheme from [14], and [22, Sect. 8.4.2] describes a fully generic version that can be instantiated over any cyclic group. We conclude that the setting that Sect. 9.1 of RFC4880 describes is precisely what we refer to as "Configuration A".

Sects. 5.1 and 5.5.2 and 5.5.3 of [12] define how ElGamal ciphertexts, public keys, and secret keys, respectively, must be represented as binary strings, but add nothing to the picture that RFC4880 conveys of ElGamal encryption. As these are the only technical statements that RFC4880 makes on ElGamal encryption, we conclude that, while the standard is precise with the formatting of keys and ciphertexts, it does not go beyond a bare interpretation of [14] when it comes to *how* keys and randomnesses are meant to be picked and *how* key generation, encryption, and decryption are supposed to be conducted.

[2] It is well-known that ElGamal encryption is malleable and thus not IND-CCA secure. However, our analysis here is not concerned with indistinguishability but key recovery.

[3] A third countermeasure is to explicitly ascertain the order of ciphertext component C'_1, i.e., to refuse decrypting C' if $(C'_1)^q \neq 1$. This however would require an additional exponentiation.

ElGamal in OpenPGP libraries. This motivates us to look at the way OpenPGP libraries interpret the standard. Here are our findings on three of the most popular ones:

Go Go does not provide code to generate ElGamal keys; it only provides algorithms to encrypt and decrypt. The ephemeral exponent y used in encryption is chosen from $[0\,..\,p-1]$. This is fully conform with RFC4880.

Crypto++ The Crypto++ library follows Configuration C by generating a random safe prime and choosing for the generator the smallest quadratic residue. It however deviates from Configuration C by picking both exponents x and y from a *short interval* $\left[1\,..\,2^{2\left\lfloor 2.4\sqrt[3]{n\ln(n)^2}-5\right\rfloor}\right]$, where $n = \lceil \log_2(p) \rceil$. For convenience, we tabulate the upper bound in Appendix A. As the group generator is not primitive, this setting is in conflict with RFC4880.

gcrypt The gcrypt library generates a Lim–Lee modulus like in Configuration D; the minimum size of the prime factors $q_i | (p-1)$ is determined by a hard-coded table with entries very close to the upper bound in the Crypto++ case above. Unlike in Configuration D, the generator is chosen to be the smallest integer generating the full group $\mathbb{Z}_p^\times$. Both exponents x and y are sampled from *short intervals* of size roughly $q^{3/2}$. We give the exact sizes for q_i, x and y in Appendix A. Due to the short exponents, this setting is in conflict with RFC4880.

ElGamal parameters in the wild. To get a complete picture of the use of ElGamal in OpenPGP, it is not sufficient to look at the prominent open-source libraries. Indeed a large portion of the user base relies on proprietary or exotic implementations, which are impossible to track. To address these, we look at the OpenPGP keys registered on public key servers. We analyse an OpenPGP server dump [3] produced on Jan 15, 2021 containing 2,721,869 keys, out of which 835,144 contain ElGamal subkeys.

An OpenPGP ElGamal public key consists of the triplet (p, g, X). This information alone is not sufficient to ascribe the key to one configuration with certainty, however partial information can be deduced by attempting to factor $p-1$. For example, safe primes are easily recognized by running a primality test on $(p-1)/2$, then a quadratic residuosity test reveals whether g generates the prime order subgroup or the full group. For random primes of the sizes we look at, it is in general infeasible to obtain a full factorization of $p-1$, however partial factorizations and residuosity tests let us at least formulate credible hypotheses on the key generation process.

To classify public keys, we conducted trial division on $p-1$ with primes up to 2^{25}, then repeatedly applied the elliptic curve factorisation method (ECM) [19], until we felt guilty for the carbon emissions. Then we applied n-th residuosity tests to g for the factors we found. We did not attempt to gain information on the exponent x that defines $X = g^x$. Our findings are as follows:

- 69.4% use safe primes: 12.8% match Configuration C, while 55.6% use a quadratic non-residue for g. Surprisingly, only 16 primes account for all but 237 of these keys, and only 4 account for all but 1,493, indicating a lion's share of "standard" safe primes.
- For 25.3% of the moduli we could prove that $(p-1)/2$ is not prime, but we could find no factor, likely pointing to Lim-Lee primes. For all but 47 keys g fails all residuosity tests, which would be consistent with gcrypt's key generation.
- There is a small share (5.0%) of "quasi-safe primes", i.e., primes of the form $p = Q \cdot q + 1$ where q is an unusually large prime (0.988 times the size of p on average) and $Q > 2$, which suggests that q was chosen before p. Only 21 of these keys use a generator of the group of order q and are thus consistent with Configuration B; the rest is either consistent with Configuration A, or with none.
- Finally there are only 2,158 moduli for which we found non-trivial factors, but we were not able to finish the factorization, indicating that they were either chosen at random, or like in Configuration B. Looking at the order of g, only 30 are consistent with Configuration B, the rest being either consistent with Configuration A, or none. These would almost be irrelevant if it wasn't for the attack we describe in Section 4.

3 COMPUTING DISCRETE LOGARITHMS

In the next sections we will need to solve discrete logarithms given partial knowledge of the exponent. We review here the necessary algorithms. In what follows we let g be a group generator of order N, and we let $X = g^x$ be the element of which we seek the discrete logarithm.

Pollard's Rho algorithm [29] is the most efficient generic algorithm to compute discrete logarithms. On average, it performs $\sqrt{\pi N/2}$ group operations, and uses a constant amount of memory. In [29], Pollard also introduced the lesser known Lambda method, which performs better when it is known that $x < B \ll N$, requiring only $\approx 2\sqrt{B}$ group operations and $O(\log(B))$ memory [37, §5.1].

When $N = LM$, we can compute $x \bmod L$ by solving a discrete logarithm in the group of order L generated by g^M. The Pohlig–Hellman algorithm [28] applies this to all prime factors of N, and then recovers x via the Chinese Remainder Theorem (CRT).

We will combine all of these techniques in the case where $N = q_0 \cdots q_n$, and $x < B$. Assume $q_0 < \cdots < q_n$, and let $Q = q_0 \cdots q_{n-1}$. We first compute $x \bmod q_i$ for $0 \le i < n$ using Pollard's Rho, then use the CRT to compute $w := x \bmod Q$. Thus $g^x = g^{zQ+w}$ for some unknown $z < \lceil B/Q \rceil$. Finally, we recover z as the discrete logarithm of g^y / g^w to base g^Q, using Pollard's Lambda. The total cost is $O\big(\sqrt{q_{n-1}} + \sqrt{B/Q}\big)$ time and $O(\log(B/Q))$ storage. We stress that this is only better than Pohlig–Hellman when $B \ll N$.

This strategy was already used by van Oorschot and Wiener to reduce the security of variants of Configuration A that use short exponents for public keys [36]. We will use it, instead, to recover ephemeral secrets in cross-configuration scenarios described in Section 4.

In Section 5.4, we will need to solve discrete logarithm instances where some non-adjacent bits of x are known. Neither the Rho nor the Lambda method can take advantage of this information, but the simpler baby-step/giant-step (BSGS) method [32] can. If x has n unknown bits, BSGS performs $1.5 \cdot 2^{n/2}$ group operations on average, and stores $2^{n/2}$ hash table entries. A linear time/memory trade-off is possible in BSGS.

However, as n becomes larger, BSGS has two important drawbacks: it uses unrealistically large amounts of memory, and it parallelizes poorly. A better alternative is van Oorschot and Wiener's (vOW) parallel collision search applied to meet-in-the-middle algorithms [37, §5.3], which is much more memory efficient, and promises a linear parallel speed-up. Based on their analysis, vOW is expected to require $7 \cdot 2^{3n/4-m/2-1}n$ group operations, where 2^m is the amount of storage available, counted as a number of hash table entries, and subject to the constraint $m \leq n/2$.

4 CROSS-CONFIGURATION ATTACKS

The disagreements on the interpretation of the OpenPGP standard may raise doubts on the interoperability between the libraries. For instance, in an imaginary setting where the `Crypto++` code is used to generate a key pair, the `Go` code is used to encrypt a message to the public key, and the `gcrypt` code is used to decrypt the ciphertext, it has to be asked whether confidentiality is maintained. While in a basic scenario these three libraries can, to the best of our knowledge, in fact interoperate securely, we shall now see that some choices made by `Crypto++` and `gcrypt` prove to be fatal in a broader context.

van Oorschot and Wiener [36] had already observed that, if: (1) $p-1$ contains enough small factors, and (2) g generates the full group (e.g., in Configuration A), then using short exponents in the public key $X = g^x$ can lead to key recovery, as described in Section 3. As we have seen, `Crypto++` uses safe primes, so it is not at risk. Although `gcrypt` generates $p-1$ with many distinct factors, none of these is small, and it is thus also safe.

However, both `Crypto++` and `gcrypt` also use short exponents in the ephemeral key $Y = \hat{g}^y$. But the generator $\hat{g}$ is part of a public key that may have been generated by a different library, possibly one adhering to Configuration A. Our analysis of registered keys shows that such public keys, albeit rare, do exist. This immediately leads to a message recovery attack: given a ciphertext $(\hat{g}^y, M \cdot \hat{g}^{xy})$, the attacker finds y, uses the public key to compute $\hat{g}^{xy}$, and recovers the cleartext M.

To recap, whenever (1) the public key of the receiver defines a generator g of a group containing "enough" small-order subgroups, and (2) the sender uses short exponents in the ephemeral key, a message intercepting attacker can decrypt any communication from sender to receiver. The exact attack cost depends on the number and the size of the small subgroups, which we analyze next.

Practical exploitation. Like in Section 3, let $N = q_0 \cdots q_n =: Qq_n$ be the group order of a public key, with q_n not necessarily prime. Assume that ephemeral exponents for encryption are sampled from a short interval $[0, B[$, with $B \ll N$. As we saw, the cost of recovering the exponent depends on two factors: the largest divisor q_{n-1} of Q, and B/Q.

Note that, while Q and the q_i's depend on the receiver's public key, the bound B is set by the sender. To compare the relative risk for several public keys, without taking a specific sender library into account, it is best to look at the sizes of q_{n-1} and $1/Q$: the first number measures the maximum effort done by the Rho algorithm, the latter measures an amount of effort "spared" by the Lambda algorithm.

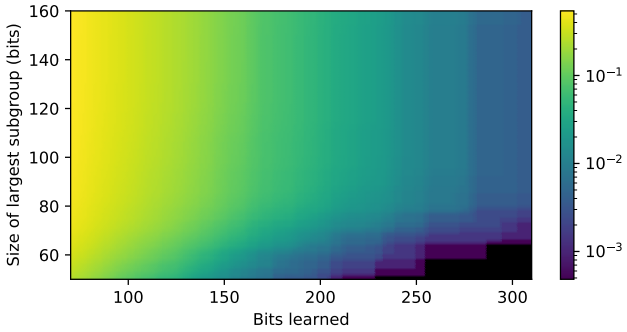

Figure 1: Normalized cumulative distribution function $\Phi(\lambda, \rho)$ of group orders N that contain a 2^ρ-smooth factor Q of at least λ bits.

To quantify the threat, we look at these two quantities for those groups in the OpenPGP key dump that contain small factors. None of the moduli we called "quasi-safe" are seriously at risk, since Q is too small. From the remaining 2,158 keys, we narrow down the selection to those that had 2048 bits modulus, were not created before 2016, and were not expired nor revoked, leaving us with 2,071 keys. Because we could not complete the factorization of any of these, we can only state an upper bound on the cost of attacking them, but it is entirely possible that some keys are much more vulnerable than what our findings suggest.

In Figure 1 we plot the (normalized) cumulative counting function

$$\Phi(\lambda, \rho) = \#\left\{N = q_0 \cdots q_n = Qq_n \;\middle|\; q_0, \ldots, q_{n-1} \leq 2^\rho, Q \geq 2^\lambda\right\},$$

which counts, among the 2,071 keys, how many are exposed to message recovery with a Rho effort at most $2^{\rho/2}\sqrt{\pi/2}$ and a Lambda effort at most $\sqrt{B}/2^{\lambda/2-1}$. Note that the factors of a single group order can arranged in more than one way, and thus appear in more than one bucket.

From $\Phi(\lambda, \rho)$ we obtain the number of keys for which message recovery is possible with a given effort with respect to a given encrypting library. For example, we know that for moduli of 2048 bits `gcrypt` has $y < 2^{344}$ (see Appendix A): limiting both the Lambda and the Rho efforts to roughly 2^{40} modular multiplications, a computation well within reach of a casual attacker, we find that there are at least $\Phi(344-80, 80) = 15$ vulnerable keys. Even more concerning is the case of `Crypto++`, which samples exponents in $[1 \,..\, 2^{226}]$: in this case the number of weak keys goes up to $\Phi(226-80, 80) \geq 231$. It is plausible that a state-level attacker could target as much as $\Phi(344-140, 140) \geq 83$ (`gcrypt`) and $\Phi(226-140, 140) \geq 895$ (`Crypto++`) keys.

To confirm the vulnerability we used GPG to encrypt a message to the only key contributing to $\Phi(344-56, 59)$, expecting to complete the attack using roughly 2^{30} modular multiplications. We were able to recover the plaintext in less than 2.5 hours on a single Intel E5-2640 core clocked at 2.40GHz.

5 ATTACKS ENABLED BY SIDE-CHANNELS

The attack conditions exploited in Section 4 emerged from different implementers coming up with different conceptual interpretations of ElGamal encryption. In particular, the attacks did not depend on implementation details. The current section specifically considers issues connected with implementation weaknesses. While a couple of our findings can quickly be recognized and comprehended as negatively affecting security, for others it remains unclear, at first, how they can be practically exploited. This is where, again, the different interpretations of ElGamal encryption come into play.

A crucial component of an ElGamal implementation is the modular exponentiation routine. In this section we study the details of the exponentiation routines of `Go`, `Crypto++`, and `gcrypt`, and we find all three vulnerable to side-channel attacks. The attack conditions in `Go` and `Crypto++` are immediately identified when inspecting (our pseudo-code versions of) the algorithms, and they can be exploited with standard side-channel exploitation techniques like cache timing analysis. The existence of the attack in `Go` is not surprising, as comments in the code indicate that the authors are aware of the weaknesses.

Attacking the `gcrypt` implementation is significantly more involved as the authors of that code took explicit measures to prevent side-channel leakage. By exploiting a condition that the authors allegedly overlooked we present a key recovery attack that significantly reduces the security margin of `gcrypt`. The attack is only practically exploitable against `gcrypt`-generated keys with moduli up to 1024 bits. However we show that the vulnerability is more worrisome in an interoperability context where `gcrypt` decryption is used in combination with a `Crypto++` key: in this case we experimentally confirm a practical key recovery attack against a non-negligible fraction of `Crypto++` keys with 2048 bits modulus.

5.1 Exponentiation Algorithms

The task of an exponentiation algorithm (EA) is to compute $R = B^x$ given B and x, where base B is an element of a cyclic group and exponent x is a non-negative integer. As any two EAs implement precisely the same function (just possibly in a different way), the specific exponentiation method relied upon in a cryptographic library can be freely chosen by the implementer. Many different EAs are known and common, where fixed-window, comb-based, and sliding-window exponentiation represent some classic options. These can be seen as derived from the basic Square-and-Multiply algorithm with the goal of improving its efficiency by intelligently grouping together the bits of the exponent, precomputing small sets of frequently used values, or similar. Some but not all details of the three algorithms are relevant for our attacks. Thus, while referring to [22, Sect. 14.6–7] for a systematic treatment, in this section we reproduce the algorithms' inner mechanics only up to a suitable level of abstraction. We start with formalizing two notions related to exponent recoding. Both are defined with respect to a parameter w (for 'width' or 'window length') that is typically instantiated such that $2 \le w \le 5$.

An *odd-digit representation* (ODR) of a non-negative integer x with respect to a parameter w is a sequence $\Gamma = (\gamma_l, \ldots, \gamma_0)$ such that

$$x = \sum_{j=0}^{l} 2^j \gamma_j \quad \text{and} \quad \forall j\colon \gamma_j \in \{0\} \cup \{1, 3, \ldots, 2^w - 1\} \ .$$

Independently of ODRs, a *radix representation* (RR) of a non-negative integer x with respect to a parameter w is a sequence $\Gamma = (\gamma_l, \ldots, \gamma_0)$ such that

$$x = \sum_{j=0}^{l} 2^{wj} \gamma_j \quad \text{and} \quad \forall j\colon \gamma_j \in \{0, \ldots, 2^w - 1\} \ .$$

The fixed-window, comb-based, and sliding-window methods have in common that they consist of three phases: (1) an initialization phase in which some data that does not depend on the exponent is (pre-)computed and tabulated; (2) an exponent encoding phase in which an ODR or RR of the exponent is encoded into some data structure;[4] and (3) a computation-intensive online phase in which the results of (1) and (2) are combined. Exploiting that phases (1) and (2) are independent of each other and thus concurrent, practical implementations in fact do not cleanly separate the phases but interleave them for efficiency. The reproductions provided in this article, however, separate the phases for readability.[5]

The three libraries analyzed and attacked in this article—`Go`, `Crypto++`, and `gcrypt`—implement AE via the fixed-window, comb-based, and sliding-window method, respectively. We dedicate the Sections 5.2, 5.3, and 5.4 to individual treatments of their security.

5.2 Fixed-window Exponentiation

Exponentiations $R \leftarrow B^x$ in `Go`[6] are computed, given a parameter w, using a fixed-window algorithm by (1) precomputing and tabulating a set of values that depend solely on w and base element B;[7] (2) encoding an RR $\Gamma = (\gamma_l, \ldots, \gamma_0)$ of the exponent x into a sequence $(e_1, \ldots, e_L)$ (the latter merely consists of relabelling the coefficients); and (3) combining the results of (1) and (2) in an online phase.

We provide a pseudo-code version of the crucial parts of `Go`'s EA in Figure 2 (left), where lines 1–2 implement phase (1), lines 3–4 implement phase (2), and lines 5–14 implement phase (3). Note that we outsource the derivation of the exponent encoding $(e_1, \ldots, e_L)$, i.e., the conversion of the exponent to the RR and the encoding of the latter, to the function f_w invoked in line 4. The only properties of this function that are relevant for our description and attack are that the exponent x can be uniquely recovered from $(e_1, \ldots, e_L)$ and that for all e_i we have $0 \le e_i < 2^w$. (For reference, we reproduce the details of function f_w in lines 45–51 of Figure 7 (top left), in Appendix B.)

Given the above description, the computation steps indicated in Figure 2 (left) should be self-explanatory. Note that lines 8–10

[4] If an ODR encoding is used, the values tabulated in Phase 1 are usually $B^1, B^3, \ldots, B^{2^w-1}$; if an RR encoding is used, the values tabulated in Phase 1 are usually $B^0, \ldots, B^{2^w-1}$.

[5] At a high level, our attacks work independently of whether the phases are separated or interleaved. Looking a bit closer reveals that the attacks actually work better if the phases are interleaved, as it is the case in all practical implementations we are aware of, as side-channel sensors can then be placed with higher precision.

[6] The code is available at https://github.com/golang/go/blob/e491c6ee/src/math/big/nat.go#L1386-L1477

[7] In the `Go` implementation, the width parameter is hard-coded as $w = 4$.

Go

Exponentiation $R \leftarrow B^x$
Parameterized by: w
Registers: $T[0], \ldots, T[2^w - 1], W$

```
1  // Initialize T[...]
2  For 0 ≤ i < 2^w: T[i] ← B^i
3  // Encode the exponent
4  (e_1, ..., e_L) ← f_w(x)
5  // Main loop
6  R ← 1
7  For i ← 1 to L:
8     // Square w times
9     If i ≠ 1, for j ← 1 to w:
10       R ← R · R
11    // Multiply with T[e_i]
12    W ← T[e_i]
13    R ← R · W
14 Return R
```

Crypto++

Exponentiation $R \leftarrow B^x$
Parameterized by: w
Registers: $T[1], T[3], \ldots, T[2^w - 1], H$

```
15 // Initialize T[...]
16 For odd 0 ≤ i < 2^w: T[i] ← 1
17 // Encode the exponent
18 (e_1, ..., e_L) ← f_w(x)
19 // Main loop
20 H ← B
21 For i ← 1 to L:
22    If e_i ≠ 0:
23       T[e_i] ← T[e_i] · H
24    H ← H · H
25 // Accumulate results
26 R ← 1
27 For odd 0 ≤ i < 2^w:
28    R ← R · T[i]^i
29 Return R
```

gcrypt

Exponentiation $R \leftarrow B^x$
Parameterized by: w
Registers: $T[1], T[3], \ldots, T[2^w - 1], W$

```
30 // Initialize T[...]
31 For odd 0 ≤ i < 2^w: T[i] ← B^i
32 // Encode the exponent
33 (e_0, c_1, e_1, ..., c_L, e_L, c_{L+1}) ← f_w(x)
34 // Main loop
35 R ← B // implicitly process e_0 = 1
36 For i ← 1 to L:
37    // Square c_i times, then multiply with T[e_i]
38    For j ← 1 to c_i + 1:
39       Securely W ← [ j ≤ c_i ? R : T[e_i] ]
40       R ← R · W
41 // Square c_{L+1} times
42 For j ← 1 to c_{L+1}:
43    R ← R · R
44 Return R
```

Figure 2: Exponentiation routines of `Go` and `Crypto++` and `gcrypt`. For the corresponding exponent encoders $f_w(\cdot)$, see Figure 7. For pointers to the original source code, see Footnotes 6 and 9 and 11.

compute $R \leftarrow R^{2^w}$, with the $i \neq 1$ condition in line 9 arranging for a little speed-up in the very first iteration (in which $R = 1$ by line 6).

The attack condition in `Go`'s EA is clearly visible in the pseudo-code: If we assume that in each iteration of line 12 the table index e_i leaks to the adversary, then the latter can readily recover the exponent x. In Section 6.3 we show how to do this in practice. The attack condition can be removed by implementing line 12 specifically such that index e_i is not leaked.[8] Notationally, in a secure implementation we would specify line 12 as 'Securely $W \leftarrow T[e_i]$'.

5.3 Comb-based Exponentiation

Exponentiations $R \leftarrow B^x$ in `Crypto++`[9] are computed, given a parameter w, using a comb-based algorithm by (1) tabulating a set of initial values that depend solely on w; (2) encoding an ODR $\Gamma = (\gamma_l, \ldots, \gamma_0)$ of the exponent x into a sequence $(e_1, \ldots, e_L)$ (the latter merely consists of relabelling the coefficients); and (3) combining the results of (1) and (2) in an online phase.

We provide a pseudo-code version of the crucial parts of `Crypto++`'s EA in Figure 2 (center), where lines 15–16 implement phase (1), lines 17–18 implement phase (2), and lines 19–29 implement phase (3). Note that we outsource the derivation of the exponent encoding $(e_1, \ldots, e_L)$, i.e., the conversion of the exponent to the ODR and the encoding of the latter, to the function f_w invoked in line 18. The only properties of this function that are relevant for our description and attack are that the exponent x can be uniquely recovered from $(e_1, \ldots, e_L)$ and that for all e_i we have $e_i \in \{0\} \cup \{1, 3, \ldots, 2^w - 1\}$. (For reference, we reproduce the details of function f_w in lines 52–61 of Figure 7 (top right), in Appendix B.)

Given the above description, the computation steps indicated in Figure 2 (center) should be self-explanatory. (For further details on the mechanics of comb-based exponentiation, see [22, Sect. 14].)

The attack condition in `Crypto++`'s EA is clearly visible in the pseudo-code: If we assume that each execution of line 22 leaks to the adversary whether the condition is fulfilled, and further each execution of line 23 leaks the table index e_i to the adversary, then the latter can readily recover the exponent x. In Section 6 we expose how to do this in practice.

Note that eliminating the attack condition is less immediate than in Section 5.2. One promising option would be to introduce an auxiliary (dummy) register $T[0]$ (initialized in line 16 together with the other registers), and to replace lines 22,23 by the single instruction 'Securely $T[e_i] \leftarrow T[e_i] \cdot H$' that implements the table accesses (read and write) and the multiplication without leaking the index e_i.[10]

5.4 Sliding Window Exponentiation

Exponentiations $R \leftarrow B^x$ in `gcrypt`[11] are computed, given a parameter w, using a sliding-window algorithm by (1) precomputing and tabulating a set of values that depend solely on w and base element B; (2) encoding an ODR $\Gamma = (\gamma_l, \ldots, \gamma_0)$ of the exponent x into

[8] If just this countermeasure is applied, the *length* of the encoding, which might convey non-trivial information about the exponent, would still leak.
[9] The code is available at https://github.com/weidai11/cryptopp/blob/45de5c6c/algebra.cpp#L255-L314

[10] If just this countermeasure is applied, the *length* of the encoding, which might convey non-trivial information about the exponent, would still leak.
[11] The code is available at https://github.com/gpg/libgcrypt/blob/ccfa9f2c/mpi/mpi-pow.c#L393-L772

a sequence $(e_0, c_1, e_1, c_2, e_2, \ldots, c_L, e_L, c_{L+1})$; and (3) combining the results of (1) and (2) in an online phase. The encoding in step (2) requires that $\gamma_l = 1$ be the leading coefficient of Γ[12] and is such that the sub-sequence $(e_0, e_1, \ldots, e_L)$ coincides with the support $(\gamma_j)_{j:\gamma_j \neq 0}$ of Γ (i.e., the set of non-zero elements; note here we have $e_0 = \gamma_l = 1$, and for all e_i we have $e_i \in \{1, 3, \ldots, 2^w - 1\}$), and the values $c_1, c_2, \ldots, c_{L+1}$ correspond with the lengths of the vanishing subsequences of Γ (i.e., the contiguous runs of zero elements). Precisely, the encoding is such that

$$x = \sum_{i=0}^{L} 2^{\bar{c}_i} e_i \quad \text{where} \quad \forall i\colon \bar{c}_i = c_{i+1} + \ldots + c_{L+1} \ . \tag{1}$$

We provide a pseudo-code version of the crucial parts of `gcrypt`'s EA in Figure 2 (right), where lines 30–31 implement phase (1), lines 32–33 implement phase (2), and lines 34–44 implement phase (3). The function f_w in line 33 computes the exponent encoding $(e_0, c_1, e_1, c_2, e_2, \ldots, c_L, e_L, c_{L+1})$ as follows: it sets $e_0 = 1$ then loops between: (a) advancing through the bit sequence of x (from most to least significant bit) until it encounters a bit set to 1, and (b) greedily outputting the largest possible pair (c_i, e_i) subject to $e_i \in \{1, 3, \ldots, 2^w - 1\}$. We reproduce the pseudocode of function f_w in lines 62–74 of Figure 7 (bottom), in Appendix B.

Given the above description, the code in Figure 2 (right) should be mostly self-explanatory, with the exception of lines 37–40 which require further explanation: The code is functionally equivalent with 'For $j \leftarrow 1$ to c_i: $R \leftarrow R \cdot R$' (similarly to lines 41–43) followed by '$R \leftarrow R \cdot T[e_i]$', but the implementation reduces side-channel leakage by hiding for each multiplication whether the factor W is R or $T[e_i]$, and, in the latter case, which index e_i is used for the table look-up.[13] That is, in contrast with the cases of `Go` and `Crypto++`, the EA of `gcrypt` is specifically designed to offer side-channel resilience.

Despite its built-in protection measures, we identify a side-channel condition in `gcrypt`'s EA that can lead to full exponent recovery. (In Section 6.2 we demonstrate that the attack is indeed practically exploitable.) The root of the problem is that the algorithm implements the sliding-window exponentiation method. Intuitively, this method covers the digits of the bit-representation of the exponent x with a sequence of w-wide non-overlapping windows such that every 1-bit of the exponent is covered by a window and conditioned on this the number of windows is minimized. In Equation (1), value L corresponds with the number of windows used for this, any coefficient $\bar{c}_i$ corresponds with the position of the i-th window, and any coefficient e_i encodes the w exponent bits that the i-th window covers. In the EA implementation, processing a window corresponds with a multiplication $R \leftarrow R \cdot T[e_i]$ while bridging a gap between two windows corresponds with a sequence of squaring operations $R \leftarrow R \cdot R$. These operations are jointly implemented in lines 39,40, and each iteration of the loop of line 36 processes one gap-window pair.

The approach of our side-channel attack is to closely monitor the execution of line 36: The number of iterations of the loop during one exponentiation immediately reveals the encoding length L, and the time taken for the i-th iteration is, by line 38, linear in $c_i + 1$ and thus leaks, one by one, the coefficients $c_1, \ldots, c_L$.[14] Finally, we can recover coefficient c_{L+1} by similarly monitoring the loop of line 42. That is, if $(e_0, c_1, e_1, \ldots, c_L, e_L, c_{L+1}) = f_w(x)$ is the encoding of exponent x then our measurements reveal all the c_i components while, for now, the e_i components remain hidden.

Before moving on to assessing the options for amplifying the extracted information towards recovering the full exponent, let us make a final observation on `gcrypt`'s EA. Recall that line 39 tries to hide not only the table index e_i but also whether the multiplication in line 40 is with $T[e_i]$ or R. However, as our method recovers the coefficients c_i straightaway, and exclusively the last iteration of the loop of line 38 will perform a multiplication with $W = T[e_i]$ (rather than with $W = R$), line 39 *de facto* just hides the e_i coefficients. Thus, replacing lines 38–40 by the instructions 'For $j \leftarrow 1$ to c_i: $R \leftarrow R \cdot R$' followed by 'Securely $R \leftarrow R \cdot T[e_i]$' would result in code that is equivalent from a security perspective yet simpler and more efficient.[15] We thus conclude that the authors of `gcrypt` were likely not aware of the fact that the coefficients $c_1, \ldots, c_{L+1}$ leak so easily.

Now that we retrieve copies of the coefficients $c_1, \ldots, c_{L+1}$, the next subsections are dedicated to assessing the value of this information towards an attack that recovers the full exponent.

5.4.1 Modelling the leakage. As a warm up, observe that when $w = 1$ the coefficients c_i leak the whole exponent, indeed all e_i are necessarily equal to 1 in this case. For larger values of w, however, a search over the possible values of e_i is necessary. Our goal here is to quantify this partial leakage.

Letting $n + 1 = \lceil \log_2(x + 1) \rceil$ be the bit length of x, we start by observing that $n = \sum c_i$ and is thus fully known. The sequence of c_i's indicates the position of each e_i in the binary writing of x, and from the description of f_w we know that e_i is odd and $e_i < \min(2^w, 2^{c_i})$, leaving us with $\min(w, c_i) - 1$ unknown bits in e_i. Note that there is no e_{L+1} associated to c_{L+1}, which indeed leaks the number of trailing zeros of x. Thus, the total number of unknown bits of x, which we shall call the *entropy* of x, is exactly

$$H_w(x) = \sum_{i=1}^{L} (\min(c_i, w) - 1). \tag{2}$$

Given h, h^x and the leakage for x, it the computational effort for finding x grows exponentially with $H_w(x)$. We will analyze this cost precisely in Section 5.4.2. The value $H_w(x)$ depends on x, on the window size w, and on the algorithm f_w used for the encoding. We are thus interested in estimating statistics on $H_w(x)$ for secrets x of n bits. To this end, we model the encoding function f_w as a Markov process, and use standard theorems on Markov chains to deduce statistics on $H_w(x)$ as $n \to \infty$.

Model x as a continuous stream of independent and uniformly distributed bits. For any fixed w, the leakage of the encoding f_w is described by a finite state machine outputting the per-coefficient

[12] Every integer $x \geq 1$ has at least one, but often multiple, ODRs with leading coefficient 1. The `gcrypt` implementation cannot handle the $x = 0$ case.

[13] Precisely, line 39 implements the instruction "If $j \leq c_i$ then $W \leftarrow R$; else, if $j = c_i + 1$, then $W \leftarrow T[e_i]$" but ensures that it neither leaks which if-branch is taken nor what the value of e_i is.

[14] This argument assumes that multiplications and squarings require uniform time. This is the case for the `gcrypt` routines.

[15] The performance gain comes from the possibility of removing some of the decoy code required for securely implementing line 39. For further details, see the source code.

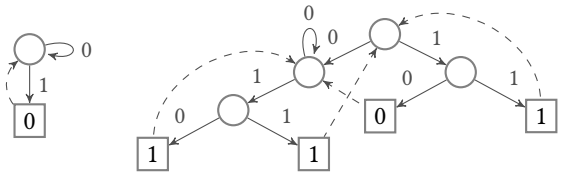

Figure 3: Finite state machines describing f_w for $w = 1$ (left) and $w = 2$ (right).

entropy $\min(w, c_i) - 1$ whenever it passes through a set of distinguished states. Figure 3 shows the machines for $w = 1$ and $w = 2$. Circles represent ordinary states and squares represent distinguished ones. Solid arrows represent a transition associated to reading a bit of x, with the read bit indicated next to the arrow. Dashed arrows represent transitions which happen without reading any bit from x, so called ϵ-transitions. The starting node is the one on top, although that is irrelevant for the modeling. The state machine moves from one state to another as it progresses through the bits of x; when it encounters a distinguished state it outputs the entropy value represented in the node, modeling the leakage of a value c_i. Both state machines in Figure 3 could be simplified, but we prefer this larger presentation as it lends itself more easily to generalization to any window size: a generalization of these state machines to a value w has $w2^w$ states.

Before converting the state machine to a Markov chain, we first need to get rid of the ϵ-transitions, by replacing them with the corresponding 0/1-transitions. This way, the number of states traversed by the machine corresponds exactly to the bit length of the input. We do not draw this replacement in Figure 3, as it would be hardly readable.

Then, the state machine is converted to a Markov chain by forgetting the bit values and assigning probability 1/2 to each transition. For $w = 2$, the transition matrix is

$$\frac{1}{2}\begin{pmatrix} 0&1&1&0&0&0&0&0\\ 0&1&0&0&0&1&0&0\\ 0&0&0&1&1&0&0&0\\ 0&1&0&0&0&1&0&0\\ 0&1&1&0&0&0&0&0\\ 0&0&0&0&0&0&1&1\\ 0&1&0&0&0&1&0&0\\ 0&1&1&0&0&0&0&0 \end{pmatrix},$$

which has stationary distribution

$$\begin{pmatrix} 0 & 1/3 & 1/12 & 1/4 & 1/24 & 1/24 & 1/8 & 1/8 \end{pmatrix}.$$

Standard theorems on Markov chains [11] let us interpret the stationary distribution as the probability that the finite state machine is found in each state at any given time, as $n \to \infty$. To compute the average entropy $H_w(x)$, it is thus sufficient to compute the inner product of the stationary distribution and the vector H_w of the entropy values output by the state machine, in this example

$$H_2 = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix},$$

leading to a "per bit" average entropy of $7/24$, i.e., $H_2(x) = 7n/24$ as $n \to \infty$, meaning that, given a uniformly sampled $n + 1$-bits exponent and its leakage $c_1, \ldots, c_{L+1}$, on average $7n/24$ bits remain to be found.

We can get even more precise information on the distribution of $H_w(x)$ using the Markov chain central limit theorem, which states that given a sequence of random variables $X_1, X_2, \ldots$ representing the states of a Markov chain, assuming the distribution of X_1, and thus of any X_i, is the stationary distribution, and given a function H assigning values to the states, the variable

$$\hat{H}_n = \frac{1}{n}\sum_{i=1}^{n} H(X_i)$$

converges in distribution to a normal variate of mean $\mu = H(X_1)$ and variance σ^2/n, where

$$\sigma^2 = \operatorname{var}(H(X_1)) + 2\sum_{i=1}^{\infty} \operatorname{cov}(H(X_1), H(X_{1+i})),$$

as $n \to \infty$. This theorem thus approximates the distribution of $H(x)/n$; put differently, it proves that as $n \to \infty$ the entropy $H(x)$ tends to a normal variate of mean $n\mu$ and variance $n\sigma^2$.

window width	states	μ	σ^2
2	8	7/24	0.058449
3	24	7/16	0.098632
4	64	341/640	0.126731
5	160	925/1536	0.145411

Table 1: Per bit statistics of the entropy $H_w(x)$ as $n \to \infty$ for different values of the window width w.

The per bit variance cannot be computed exactly, but it is easily estimated numerically: in the case $w = 2$ we get $\sigma^2 = 0.058449$. Table 1 lists the values of μ and σ^2 for all weights between 2 and 5. Of course, these asymptotic approximations only give a rough estimate of the actual probability distribution of $H_w(x)$ for a given bit length n; however they prove to be quite accurate in practice even for small values of n, as the experiments in Figure 8 in appendix show. Using these approximations, we will give rather precise estimates of the computational effort needed to recover random secret keys generated by `gcrypt` and other libraries.

5.4.2 Key recovery. To assess the practical impact of the leakage of sliding window exponentiation exposed so far we need to look into the exact parameters used by the various libraries. Recall that `gcrypt` uses short secret exponents, as detailed in Appendix A. The window size used for exponentiation goes from 1 to 5, depending on the bit size of the exponent.[16] Table 2 summarizes the parameters adopted by `gcrypt` for some common modulus sizes.

Based on `gcrypt`'s parameters, Table 2 also reports on the mean leakage entropy $H_w(x)$ as analyzed in Section 5.4.1. We can thus compare the expected difficulty of recovering the secret exponent via the leakage against the target difficulty of solving discrete logarithms directly. `gcrypt`'s moduli are parameterized so that the best algorithm for computing discrete logarithms is, indifferently, index calculus in $\mathbb{Z}_p$, or Pollard's Rho in the subgroup of largest prime order q. For simplicity, we use the cost of Pollard's Rho, namely $\sqrt{\pi q/2}$, as our reference difficulty.[17]

To exploit the leakage, we resort to vOW parallel collision search (see Section 3), which costs $7 \cdot 2^{3H_w(x)/4 - m/2 - 1} H_w(x)$ modular multiplications, where 2^m is the amount of RAM available, counted

[16] See https://github.com/gpg/libgcrypt/blob/1a83df98/mpi/mpi-pow.c#L442-L451.
[17] Since in practice q is unknown to an attacker, this ignores the cost of factoring $p - 1$ first, but accounting for it would only make the side channel attack look better.

$\|p\|$	$\|q\|$	$\|x\|$	w	$E(H_w(x))$	Pollard	vOW
1024	166	250	3	108.94	82.65	63.04
2048	226	340	4	180.62	113.15	114.77
3072	270	406	4	215.79	134.65	141.40
4096	306	460	4	244.56	152.65	163.16

Table 2: Group ($|q|$) and secret exponent ($|x|$) bit sizes, and window size (w) used by `gcrypt` for each modulus size ($|p|$). $E(H_w(x))$ is the mean leakage entropy estimated using Table 1. "Pollard" indicates the (base 2 log of the) expected running time of Pollard's Rho algorithm in a group of size q, as a number of modular multiplications. "vOW" indicates the expected running time of van Oorschot and Wiener's algorithm using a table of 2^{60} entries.

as a number of hash table entries (e.g., for $|p| = 2048$ an entry occupies slightly more than 256 bytes). Both Pollard's Rho and vOW can be parallelized with a linear speed-up [37, §5.1–3], justifying the comparison.

The last two columns of Table 2 report the base 2 logarithm of the expected efforts of Pollard's Rho, and of vOW assuming a hash table with at most 2^{60} entries (note that even this "little" memory is quite unrealistic). Comparing the columns, the leakage of windowed exponentiation does not appear to be a serious threat for the average `gcrypt` secret key. However this assessment is deceiving on two accounts: (a) the standard deviation of $H_w(x)$ is rather large, and thus some keys will be much easier to break than others; and (b) `gcrypt` is assumed to be secure even when it operates on secret keys generated by a different library, as long as these are valid OpenPGP keys. Some libraries, such as `Crypto++`, generate secret exponents with as few as $|q|$ bits, their keys are thus easier targets.

Taking for reference the most commonly used modulus size $|p| = 2048$, `Crypto++` exponents are between 1 and 226 bits long, and the window size used by `gcrypt` is at most $w = 3$, leading to an average entropy of 98.0 bits. Worse still, more than one in $10,000$ keys is expected to have at most 80 bits of leakage entropy: for such a small search space, vOW is expected to only require 2^{50} modular multiplications (roughly 2^{35} Gcycles in software) for a hash table of 2^{35} entries (roughly 4 TiB). We thus conclude that attacking a non-negligible proportion of `Crypto++`-generated public keys within `gcrypt`'s decryption routine is well within reach of a moderately resourceful adversary.[18]

Proof of concept. To confirm the reality of the threat, without going as far spending several thousands of dollars, we implement a simple parallelized BSGS using GMP 6.1.2, and test it on a node equipped with 20 Xeon E5-2640 cores clocked at 2.40GHz and 64GiB of RAM. Due to memory contention the parallel speed-up is sublinear, nevertheless we observe a speed-up slightly above 10× using all 20 cores, thus justifying our preference for BSGS over the more complex vOW algorithm which is marred by larger constants. Our implementation uses hash table entries of 16 bytes, independently of the modulus size; the node can thus accommodate for tables

[18] At the time of writing, 8 Amazon EC2 `r6g.metal` instances with 0.5 GiB of RAM and 64 virtual CPUs each cost about 12,000$ per month.

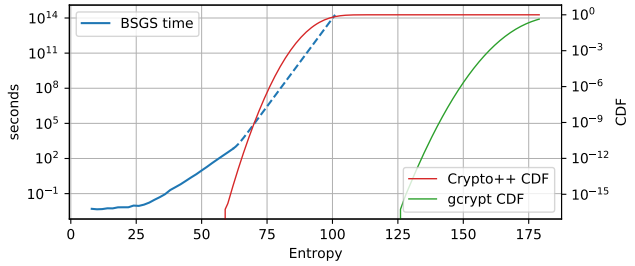

Figure 4: Running time of BSGS for increasing values of the entropy $H_w(x)$, on our node equipped with 20 cores and 64GiB of RAM. The dashed line indicates extrapolated running times. Also represented the cumulative distribution functions of the normal distributions of parameters $(225 \cdot 7/16, 225 \cdot 0.098632)$ and $(339 \cdot 341/640, 339 \cdot 0.126731)$, roughly corresponding to the entropy distributions of `Crypto++` and `gcrypt` for a modulus of 2048 bits.

with as many as 2^{32} entries, ensuring the running time scales like the square root of the entropy, up to 64 bits of entropy. Figure 4 plots the running time of BSGS for increasing values of $H_w(x)$, and superimposes the cumulative distribution function of $H_w(x)$ for `Crypto++` and `gcrypt`, indicating what percentage of keys is broken in a given time. Since BSGS is deterministic, we use for the benchmarks the exponent that is tested by BSGS last, thus the expected running time for an average exponent of given entropy is half the time reported in the figure.

As a final confirmation, we instrument the `Crypto++` code to generate random secret exponents in a tight loop, and output them along with their leakage entropy. Over 2^{28} samples, we obtained a 226 bits exponent with 62 bits of leakage entropy. Our probing strategy described in Section 6.2 produces the expected leakage pattern, from which our BSGS code recovers the exponent in about 30 minutes using 20 cores.

6 PRACTICAL EXPLOITATION

In this section we prototype the attacks described in Section 5 to show that practical exploitation is possible. The attacks are mounted on a Core i7-8650U CPU with Ubuntu 18.04.5. We demonstrate how FLUSH+RELOAD may be employed to extract leakage from the sliding window modular exponentiation implementation of `gcrypt` and use that to obtain full key recovery, as detailed in Section 5.4. The exploitation of `gcrypt` presents interesting aspects for two reasons: i) the library was patched to protect its modular exponentiation routine against cache-based side channels attacks highlighted by Yarom and Falkner [38] and by Liu *et al.* [21]; and ii) the attack can be conducted on commodity hardware specifically owing to the ElGamal interoperability issues highlighted in Section 2. We also show how a PRIME+PROBE attack on data cache is able to reveal the indices of the table accesses performed by the fixed window modular exponentiation implementation of `Go` described in Section 5.2. While the developers acknowledge that the implementation is not safe, practical exploitation is still of interest as it reveals a few caveats that we describe. Finally, we briefly discuss here the case of

comb-based exponentiation seen in `Crypto++` (see Section 5.3). The algorithm presents a combination of the weaknesses that we exploit for `gcrypt` and `Go`: with reference to Figure 2, secret-dependent control flow at line 22 and secret-dependent table access at line 23. Exploitation relies on the same techniques that we describe below.

6.1 Threat Model

We consider a victim process that uses a private key to decrypt ElGamal ciphertexts. We assume that the attacker has knowledge of the victim program and of the specific decryption algorithm that is used. We also assume that the attacker may cause the victim process to execute, for example because the victim uses a PGP-enabled mail client which automatically decrypts emails upon reception. Consequently, we assume that the attacker is able to synchronise side channel measurements with the victim execution. No physical access is required to perform the attacks. Given that we target cache side channels, the attacker may either be local, or it may be running on a different, co-located, VM. The former may target any cache level, the latter may only target LLC (unless the VM is hyperthread-colocated). The attack in Section 6.3 targets a `Go` binary, where ASLR is disabled by default. As to the attack in Section 6.2, we make use of FLUSH+RELOAD on shared (physical) data pages and so ASLR has no impact. Wlog we conduct the exploitation against the L1 cache: as shown by Liu *et al.* [21], the same exploitation may be achieved against higher cache levels.

6.2 Sliding window: the case of `gcrypt`

Environment. We target the function `_gcry_mpi_powm` of the latest version `gcrypt` shipped by Ubuntu 18.04.5 LTS. The function implements the approach described in Figure 2. The implementation takes a few precautions against the attacker described in Section 6.1 to avoid leaking sensitive data and instruction cache accesses. In particular, it hides i) whether the operation at line 40 is a modular multiplication or squaring; and ii) whether, at line 39, W is initialised with R or with an entry from table T (and in that case, it also hides which position e_i of the table is loaded). This is accomplished by way of a conditional `memcpy`-like operations (`mpi_set_cond`) that – depending on whether a flag argument is one or zero – copies a source operand into a destination operand, or leaves the destination operand unchanged. Masking is employed to ensure that an attacker cannot learn the nature of the operation by observing the control flow or changes to data or instruction cache. Furthermore, the flag is set using branchless operations.

Despite these precautions, considerable leakage can still be obtained by an attacker who can observe control flow dependent cache perturbations: in particular, the full set of values $c_1, \ldots, c_L$ can be recovered by counting the number of iterations of the `for` loop at line 38 for each of the iterations of the loop at line 36. The attacker may obtain this information by monitoring the state of the instruction cache line that contains code that is executed as part of an iteration of the outer loop *and not* of the inner loop. In practice, this can be achieved with the FLUSH+RELOAD side channel applied on a line of the instruction cache that is executed once per outer loop before the start of the inner one. The probe reveals when such a cache line is executed, and the time between two probes would depend on the execution time of the inner `for` loop. Crucially, the latter depends on the set of values $c_1, \ldots, c_L$, which would thus be leaked. This strategy is applicable to the `_gcry_mpi_powm` function: the implementation inlines the logic to determine the c_i values between the start of the first and the start of the second loop and so it fills more than one cache line; the invocation of the multiplication in the inner loop makes one iteration sufficiently long and constant-time so that inter-probe timings measurably encode the number of iterations of the inner loop.

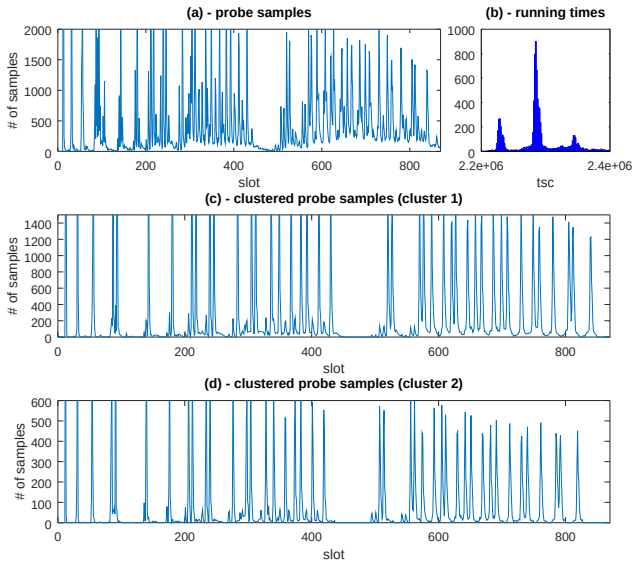

Figure 5: FLUSH+RELOAD attack using 10,000 decryption operations with `gcrypt`'s modular exponentiation function. a): number of samples with an icache hit on the target cache line across all runs. b): histogram of the running times (tsc count) for the operation. c) as a) but only considering samples whose total running time lies in the $[2.25 \cdot 10^6, 2.31 \cdot 10^6]$ interval. d) as a) but only considering samples whose total running time lies in the $[2.2 \cdot 10^6, 2.25 \cdot 10^6]$ interval.

Evaluation. To validate that the assumptions are correct, we prototype an end-to-end attack in the SMT-colocated threat model. The prototype uses two SMT-colocated processes, attacker and victim. The victim process may be triggered by the attacker to perform ElGamal decryption. We use `Crypto++` to generate a 2048-bit ElGamal instance as detailed in Section 5.4.2. The attacker process uses L1i FLUSH+RELOAD on the virtual address of the memory-mapped `libgcrypt.so` shared object that contains code that is executed by the outer loop before the inner loop begins. To collect side channel data, the attacker partitions time into N slots using the `rdtsc` instruction: at the beginning of each slot, the target line is loaded into the cache and the access time measured; after that, the cache line is flushed and finally the remaining (if any) time of the slot is spent in a busy loop. As we shall discuss later, we do not have to change any of the settings that might influence the running time of the operation (e.g. c/p states or Turbo Boost). The attacker thus collects N timing samples for load instructions from the target cache line.

We establish a threshold for the target system to determine whether the load is likely to have been served from cache (cache *hit*). We then collect 10,000 measurements by repeatedly triggering the victim. We keep track of each slot across all runs with a set of per-slot counters, all initialised to zero. For each run, whenever the sample of a slot indicates a hit based on the threshold, we increment the value of the corresponding counter.

Figure 5a plots the value of the counter of each slot for all runs. While patterns are discernible, the leakage cannot be obtained yet without further data processing. We employ a clustering strategy based on execution time, under the hypothesis that Figure 5a contains samples that are generated at different clock frequencies. Influencing factors are likely to include p-states, c-states, frequency scaling and the power state of certain execution units. Figure 5b confirms the hypothesis: it plots a histogram of the running times for the operation. The distribution of running times has a long tail, but most of the mass is concentrated in the three peaks. We use this information to cluster samples and show the results of the clustering in Figure 5c and Figure 5d, showing probes whose running time falls in the interval covered by the highest, and second highest peak, respectively. The peak intervals in the two latter figures accurately encode the c_i leakage for the key used in the exponentiation. The two figures are identical modulo a scaling factor which depends on the difference in running time. With the leakage thus obtained, we refer back to Section 5.4.2 for a detailed description of how the secret exponent is fully recovered.

6.3 Fixed window: the case of Go

Environment. We target the function `expNNMontgomery` in Go 1.15. The function implements the approach of Figure 2, with $w = 4$. The algorithm performs a secret-dependent table access: in particular, in line 13, the accumulator R is multiplied by the variable W, which is the results of a lookup into table T performed in line 12. The index e_i used for the lookup is the integer representation of the i-th four bits of the secret exponent. The table lookup leaves a cache-level side effect which leaks the value of all e_i.

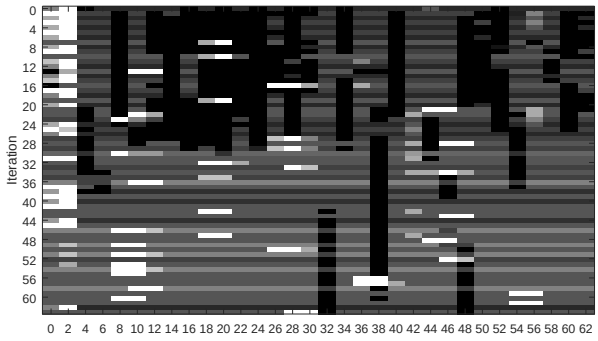

Figure 6: PRIME+PROBE attack using 10,000 decryption operations with Go's modular exponentiation function. Each row represents one iteration of the loop at line 7 of Figure 2; each column represents an L1 cache set. Lighter represents a higher probe time.

Evaluation. We prototype an end-to-end attack in the SMT colocated threat model against a 2048-bit modulus. Attacker and victim run on two separate, SMT-colocated processes. The victim process loads the private key, performs ElGamal decryption using the `golang.org/x/crypto/openpgp/elgamal` package and returns the resulting cleartext data. The code internally calls the `expNNMontgomery`. The attacker process uses PRIME+PROBE on L1d to observe the cache side-effects left by the table lookup and recover the secret index e_i. Ideally, this could be achieved by conducting the priming phase right after the start of each loop iteration (line 7 of Figure 2) and the probing phase right before the end of each loop iteration (line 13 of Figure 2). Achieving this in practice from a separate (though colocated) attacker process poses a set of challenges which we analyse here. The first challenge is how to synchronise the probes. Clearly it is impossible to practically achieve the ideal probing scenario described above, if anything because the victim cannot be made to wait until the priming and probing loops of the attacker complete. It is possible however to space them out – by busy wait and invocations of `rdtsc` – so that the priming begins as closely as possible to the start of the victim loop and the probing completes as closely as possible to the end of the victim loop. According to the assumptions of the threat model, the attacker can trigger the victim and so it is able to synchronise the first probe. The correct inter-probe delays to achieve the desired scheduling can be determined by profiling the execution of the victim. Running times in modern CPUs however are not only affected by scheduling but also by microarchitectural considerations such as temperature and load. Consequently, the attacker must build not one but a set of probing schedules, where each schedule has an associated victim running time. The attacker then chooses one to perform the attack and determines whether it is the appropriate one by comparing the running time of the victim with the running time associated with the chosen probing schedule. Whenever the schedule is not appropriate, the gathered sample has to be discarded, and the choice of probing schedule must be corrected. The first loop must be treated differently since the code is optimised to avoid the first w squarings. This means that the priming phase for the first operation must be conducted ahead of the start of the loop.

Another issue stems from the fact that the running time of the probe is roughly comparable to that of the multiplication operation of line 13. This causes a noisy signal from the side channel since not every probe is guaranteed to observe the instructions that load the table entry. This issue is addressed by probing only half of the cache sets, thus ensuring that load operations (which take place close to the beginning of the multiplication) are always observed. The set of excluded cache sets may be varied across runs to obtain data for the missing ones.

We then collect 10,000 sample sets by repeatedly triggering the victim. The attacker collects PRIME+PROBE data for the 64 iterations of the loop. Priming and probing phases make use of the pointer chasing and doubly-linked list to permit bi-directional traversal described in by Tromer *et al.* [33] to minimise the effect of prefetching. Figure 6 shows the results of the attack. The pixel array has one row per loop iteration and one column per monitored cache set (odd cache sets were excluded). Each cell displays the result of the probe during the row's iteration and the column's cache set. Lighter shades of grey mean a longer probing time for the cache set,

implying that it was accessed by the victim. The value of each row is obtained by subtracting, from each sample, the average timing of the cache set and then averaging the results. As we can see, patterns are visible and we were able to verify that they correspond to the indices e_i for the considered exponent. At this point, if the attacker knows the virtual addresses of the table entries, the value of the secret exponent is leaked in full. Otherwise, the attacker knows the pairwise equality of all four-bit sequences of the exponent (e.g. with reference to Figure 6, that the four-bit sequence at position 59 is equal to that at position 61). This knowledge restricts the search space for the exponent to the set of permutations of 16 symbols, which has size $16! \approx 2^{44}$ and can be fully explored by a trivial to parallelize exhaustive search easily done on commodity hardware. Note that this complexity is independent from the size of the modulus or the length of the secret exponent.

In our experiment we assume a short-lived victim process that terminates after decrypting. We verify that for short-lived processes, the absence of ASLR in `Go` coupled with the deterministic nature of the allocator ensures that virtual addresses (and hence cache sets) hosting the table T are constant across runs. If the victim application is long-lived however, the actual cache sets hosting table entries may vary across runs since the table is allocated in the course of each decryption. While this is clearly outside of the scope of this work, we highlight here two strategies for the attacker: the first one is to force the allocator into a specific state so that the table is allocated on the same cache sets. We verify that this is not hard to achieve since the attacker is afforded a high degree of control over garbage collector triggers (time and heap consumption) and allocation triggers caused by interacting with the victim. The other strategy is to collect all samples and cluster them according to the cache set layout of the table during the run.

7 CONCLUSIONS

We analyse one of the oldest, best understood and most historically used asymmetric encryption schemes, ElGamal [14]. We reveal that despite its popularity and longevity, when we speak about ElGamal we are referring to several different flavours of it, with key choices being left at the discretion of vendors and implementers. We show how some of these choices create interoperability challenges that lead to insecurity. We propose two "cross-configuration" attacks that are attributable to different, and – from a security standpoint – incompatible, configurations that operate together in the interconnected OpenPGP ecosystem. We believe our work can improve the security of the OpenPGP community and influence the new revision of the standard that is being drafted at the time of writing.

ACKNOWLEDGEMENTS

The authors would like to thank Werner Koch, Anil Kurmus, Yutaka Niibe and Filippo Valsorda for the discussions and feedback, and the CCS'21 reviewers for their comments that helped us improve the paper.

DISCLOSURE

We reached out to the OpenPGP users whose keys produce weak ciphertexts to encourage revoking the affected subkey. The identities of the vulnerable users (which are known since most users register OpenPGP keys associated to an email address) have been kept on a server to which only the authors have access. Physical access to that server is restricted. We have also informed the two vendors (`gcrypt` and `Go`) about the side-channel attack. As a consequence, `gcrypt` was patched (commit `632d80ef3`) to forbid the usage of short encryption randomness. `CVE-2021-33560` was also issued.

REFERENCES

[1] 2018. CVE-2018-6594. https://www.cvedetails.com/cve/CVE-2018-6594/.
[2] 2018. CVE-2018-6829. https://www.cvedetails.com/cve/CVE-2018-6829/.
[3] 2021. OpenPGP server key dump from 15/01/2021. https://pgp.key-server.io/dump/. [Online; accessed 15/01/2021].
[4] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. Predicting Secret Keys Via Branch Prediction. In *CT-RSA 2007 (LNCS, Vol. 4377)*, Masayuki Abe (Ed.). Springer, Heidelberg, 225–242. https://doi.org/10.1007/11967668_15
[5] Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. 2015. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing - and Its Application to AES. In *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 591–604. https://doi.org/10.1109/SP.2015.42
[6] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2014. Wait a minute! A fast, Cross-VM attack on AES. *IACR Cryptol. ePrint Arch.* 2014 (2014), 435. http://eprint.iacr.org/2014/435
[7] Alessandro Barenghi, Alessandro Di Federico, Gerardo Pelosi, and Stefano Sanfilippo. 2015. Challenging the Trustworthiness of PGP: Is the Web-of-Trust Tear-Proof?. In *ESORICS 2015, Part I (LNCS, Vol. 9326)*, Günther Pernul, Peter Y. A. Ryan, and Edgar R. Weippl (Eds.). Springer, Heidelberg, 429–446. https://doi.org/10.1007/978-3-319-24174-6_22
[8] Daniel J. Bernstein. 2005. *Cache-timing attacks on AES.* Technical Report.
[9] Daniel Bleichenbacher. 1996. Generating ElGamal Signatures Without Knowing the Secret Key. In *EUROCRYPT'96 (LNCS, Vol. 1070)*, Ueli M. Maurer (Ed.). Springer, Heidelberg, 10–18. https://doi.org/10.1007/3-540-68339-9_2
[10] Dan Boneh, Antoine Joux, and Phong Q. Nguyen. 2000. Why Textbook ElGamal and RSA Encryption Are Insecure. In *ASIACRYPT 2000 (LNCS, Vol. 1976)*, Tatsuaki Okamoto (Ed.). Springer, 30–43. https://doi.org/10.1007/3-540-44448-3_3
[11] Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng. 2011. *Handbook of markov chain monte carlo.* CRC press.
[12] Jon Callas, Lutz Donnerhacke, Hal Finney, David Shaw, and Rodney Thayer. 2007. OpenPGP Message Format. https://www.rfc-editor.org/rfc/rfc4880.html
[13] Srdjan Capkun, Levente Buttyán, and Jean-Pierre Hubaux. 2002. Small worlds in security systems: an analysis of the PGP certificate graph. In *Proceedings of the 2002 Workshop on New Security Paradigms, Virginia Beach, VA, USA, September 23-26, 2002*, Cristina Serban, Carla Marceau, and Simon N. Foley (Eds.). ACM, 28–35. https://doi.org/10.1145/844102.844108
[14] Taher ElGamal. 1984. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *CRYPTO'84 (LNCS, Vol. 196)*, G. R. Blakley and David Chaum (Eds.). Springer, Heidelberg, 10–18.
[15] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. 2015. Stealing Keys from PCs Using a Radio: Cheap Electromagnetic Attacks on Windowed Exponentiation. In *CHES 2015 (LNCS, Vol. 9293)*, Tim Güneysu and Helena Handschuh (Eds.). Springer, 207–228. https://doi.org/10.1007/978-3-662-48324-4_11
[16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, Spain, 2016 (LNCS, 9721)*, Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez. Springer, 279–299. https://doi.org/10.1007/978-3-319-40667-1_14
[17] Gaël Hachez and Jean-Jacques Quisquater. 2000. Montgomery Exponentiation with no Final Subtractions: Improved Results. In *CHES 2000 (LNCS, Vol. 1965)*, Çetin Kaya Koç and Christof Paar (Eds.). Springer, Heidelberg, 293–301. https://doi.org/10.1007/3-540-44499-8_23
[18] Werner Koch, brian m. carlson, Ronald Henry Tse, Derek Atkins, and Daniel Kahn Gillmor. 2020. *OpenPGP Message Format.* Internet-Draft draft-ietf-openpgp-rfc4880bis-10. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-openpgp-rfc4880bis-10 Work in Progress.
[19] Hendrik W. Lenstra. 1987. Factoring integers with elliptic curves. *Annals of Mathematics* 126 (1987), 649–673.
[20] Chae Hoon Lim and Pil Joong Lee. 1997. A Key Recovery Attack on Discrete Log-based Schemes Using a Prime Order Subgroup. In *CRYPTO'97 (LNCS, Vol. 1294)*, Burton S. Kaliski Jr. (Ed.). Springer, 249–263. https://doi.org/10.1007/BFb0052240
[21] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 605–622. https://doi.org/10.1109/SP.2015.43

[22] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. 1997. *Handbook of Applied Cryptography*. CRC Press.

[23] Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. 1999. Power Analysis Attacks of Modular Exponentiation in Smartcards. In *CHES'99 (LNCS, Vol. 1717)*, Çetin Kaya Koç and Christof Paar (Eds.). Springer, Heidelberg, 144–157. https://doi.org/10.1007/3-540-48059-5_14

[24] Fabrizio Milo, Massimo Bernaschi, and Mauro Bisson. 2011. A fast, GPU based, dictionary attack to OpenPGP secret keyrings. *J. Syst. Softw.* 84, 12 (2011), 2088–2096. https://doi.org/10.1016/j.jss.2011.05.027

[25] Serge Mister and Robert J. Zuccherato. 2005. An Attack on CFB Mode Encryption as Used by OpenPGP. In *Selected Areas in Cryptography, 12th International Workshop, SAC 2005, Kingston, ON, Canada, August 11-12, 2005 (LNCS, 3897)*, Bart Preneel and Stafford E. Tavares. Springer, 82–94. https://doi.org/10.1007/11693383_6

[26] Andrew M Odlyzko. 1995. The Future of Integer Factorization. *CryptoBytes (The technical newsletter of RSA Laboratories)* 1, 2 (1995), 5–12. http://www.dtc.umn.edu/~odlyzko/doc/future.of.factoring.pdf

[27] Damian Poddebniak, Christian Dresen, Jens Müller, Fabian Ising, Sebastian Schinzel, Simon Friedberger, Juraj Somorovsky, and Jörg Schwenk. 2018. Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, USA, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 549–566. https://www.usenix.org/conference/usenixsecurity18/presentation/poddebniak

[28] S. Pohlig and M. Hellman. 1978. An Improved Algorithm for Computing Logarithms over GF(p) and Its Cryptographic Significance. *IEEE Trans. Inf. Theor.* 24, 1 (Jan. 1978), 106–110. https://doi.org/10.1109/TIT.1978.1055817

[29] John M. Pollard. 1978. Monte Carlo methods for index computation mod p. *Math. Comp.* 32 (1978), 918–924.

[30] Birger Schacht and Peter Kieseberg. 2020. An Analysis of 5 Million OpenPGP Keys. *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.* 11, 3 (2020), 107–140. https://doi.org/10.22667/JOWUA.2020.09.30.107

[31] Claus-Peter Schnorr. 1990. Efficient Identification and Signatures for Smart Cards. In *CRYPTO'89 (LNCS, Vol. 435)*, Gilles Brassard (Ed.). Springer, Heidelberg, 239–252. https://doi.org/10.1007/0-387-34805-0_22

[32] Daniel Shanks. 1971. Class number, a theory of factorization, and genera. In *Proc. Symp. Pure Math.*, Vol. 20. AMS, 41–440.

[33] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptol.* 23, 1 (2010), 37–71. https://doi.org/10.1007/s00145-009-9049-y

[34] Yiannis Tsiounis and Moti Yung. 1998. On the Security of ElGamal Based Encryption. In *PKC'98 (LNCS, Vol. 1431)*, Hideki Imai and Yuliang Zheng (Eds.). Springer, Heidelberg, 117–134. https://doi.org/10.1007/BFb0054019

[35] Alexander Ulrich, Ralph Holz, Peter Hauck, and Georg Carle. 2011. Investigating the OpenPGP Web of Trust. In *ESORICS 2011 (LNCS, Vol. 6879)*, Vijay Atluri and Claudia Díaz. Springer, 489–507. https://doi.org/10.1007/978-3-642-23822-2_27

[36] Paul C. van Oorschot and Michael J. Wiener. 1996. On Diffie-Hellman Key Agreement with Short Exponents. In *EUROCRYPT'96 (LNCS, Vol. 1070)*, Ueli M. Maurer (Ed.). Springer, 332–343. https://doi.org/10.1007/3-540-68339-9_29

[37] Paul C. van Oorschot and Michael J. Wiener. 1999. Parallel Collision Search with Cryptanalytic Applications. *Journal of Cryptology* 12, 1 (Jan. 1999), 1–28. https://doi.org/10.1007/PL00003816

[38] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

A MAPPING FINITE FIELD SIZES TO PARAMETER SIZES

Matching an ElGamal key to a security level is more complex than just choosing a bit size for the modulus p. In configurations where p is not a safe prime, it is necessary to set a minimum size for one prime factor $q|(p-1)$, or all prime factors other than 2 in the case of Lim–Lee primes. Moreover, some libraries sample exponents x and y from short intervals, whose size must also be a function of the security level.

Table 3 indicates how different implementations and international standards map field sizes (a.k.a. modulus sizes) to the different parameters. The first group of columns describes `gcrypt`'s somewhat baroque parameters: a hard-coded table, which the source code attributes to Wiener[19], determines an integer l as a function of p. From l, `gcrypt` gets the next multiple of 2:

$$m = 2\lfloor (l+1)/2 \rfloor,$$

then generates a Lim–Lee modulus such that all the odd prime factors $q|(p-1)$ are larger than 2^m. Secret exponents x are uniformly sampled from the interval $[1, 2^{3m/2+1} - 1]$, the factor $3/2$ providing a "large safety margin".[20] Ephemeral exponents y are sampled from the similarly sized interval: first `gcrypt` increments $3l/2$ to the next multiple of 8, i.e.,

$$r = 8\lfloor (3l+14)/\rfloor,$$

then samples y from $[1, 2^r - 1]$, rejecting if $\gcd(y, p-1) \neq 1$.[21]

`Crypto++` uses safe primes in key generation, it does thus not need to apply a correspondence for the size of the prime order subgroup. It does however sample the exponents x and y for the short interval

$$\left[1 \,..\, 2^{2\left\lfloor 2.4\sqrt[3]{n\ln(n)^2}-5\right\rfloor}\right],$$

using a formula attributed to Odlyzko [26].[22] We tabulate the exponent of the upper bound in Table 3.

For reference, we also include in Table 3 the group size recommendations of standardization bodies NIST[23], RFC[24] and BSI[25].

Table 3: Bit-size of modulus and bit-size of exponents according to `gcrypt`, `Crypto++` and various standards.

	`gcrypt`			`Crypto++`	NIST	RFC	BSI
$\lvert p\rvert$	l	$\lvert x\rvert$	$\lvert y\rvert$	$\lvert x\rvert, \lvert y\rvert$	$\lvert q\rvert$	$\lvert q\rvert$	$\lvert q\rvert$
512	119	181	184	120			
768	145	220	224	144			
1024	165	250	248	164	160	135	140
1536	198	298	304	198		168	
2048	225	340	344	226	224	190	200
3072	269	406	408	268	256	224	256
4096	305	460	464	304		280	
7680	1160	1741	1744	398	384	380	384
15360	2120	3181	3184	530	512	480	512

B EXPONENT ENCODERS

In Section 5 and Figure 2 we presented a selection of exponentiation algorithms but didn't specify all their details. These details can be found in Figure 7. In the figure, we write int2str for the function that converts a non-negative integer into its Big-Endian binary representation, we write str2int for the function that converts a (Big-Endian) binary string into the non-negative integer that it represents. We denote string concatenation with ||, the length function for strings with len, the string prefix relation with $\preceq$, and the Hamming Weight function with HW.

[19] See comment "Michael Wiener's table" at https://github.com/gpg/libgcrypt/blob/1a83df98/cipher/elgamal.c#L105-L109

[20] See comment at https://github.com/gpg/libgcrypt/blob/1a83df98/cipher/elgamal.c#L312-L315.

[21] See https://github.com/gpg/libgcrypt/blob/1a83df98/cipher/elgamal.c/#L190-L196, the function `gen_k` is always called with `small_k = 1` in ElGamal.

[22] https://github.com/weidai11/cryptopp/blob/434e3189/nbtheory.cpp#L1045-L1050.

[23] Table 2 of https://doi.org/10.6028/NIST.SP.800-57pt1r5

[24] Sec. 5 of https://www.ietf.org/rfc/rfc3766.html

[25] Table 3.2 of https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf

Go

Exponent encoder $f_w(x)$

45 $t \leftarrow \text{int2str}(x)$
46 While $w \nmid \text{len}(t)$: $t \leftarrow \texttt{0} \mathbin{\|} t$
47 $L \leftarrow \text{len}(t)/w$
48 For $i \leftarrow 1$ to L:
49 Let $x_i \mathbin{\|} t \leftarrow t$ w/ $\text{len}(x_i) = w$
50 $e_i \leftarrow \text{str2int}(x_i)$
51 Return $(e_1, \ldots, e_L)$

Crypto++

Exponent encoder $f_w(x)$

52 $L \leftarrow 0$
53 While $x > 0$:
54 Find $0 \le D < 2^w$ s.t.
55 $x \equiv D \pmod{2^w}$
56 If $D \notin \{1, 3, .., 2^w - 1\}$:
57 $D \leftarrow 0$
58 $x \leftarrow (x - D)/2$
59 $L \leftarrow L + 1$
60 $e_L \leftarrow D$
61 Return $(e_1, \ldots, e_L)$

gcrypt

Exponent encoder $f_w(x)$

62 Assert $x > 0$
63 $t \leftarrow \text{int2str}(x)$
64 Let $1 \mathbin{\|} t \leftarrow t$ // Remove prefix 1 from t
65 $e_0 \leftarrow 1$; $L \leftarrow 0$
66 While $\text{HW}(t) > 0$:
67 Find longest prefix $y \preceq t$ s.t.
68 $y = \texttt{0} \ldots \texttt{0}\ y_{k-1} \ldots y_0$ with
69 $1 \le k \le w \wedge y_{k-1} = 1 \wedge y_0 = 1$
70 Let $y \mathbin{\|} t \leftarrow t$ // Remove prefix y from t
71 $L \leftarrow L + 1$
72 $c_L \leftarrow \text{len}(y)$; $e_L \leftarrow \text{str2int}(y)$
73 $c_{L+1} \leftarrow \text{len}(t)$
74 Return $(e_0, c_1, e_1, \ldots, c_L, e_L, c_{L+1})$

Figure 7: Exponent encoders of `Go` and `Crypto++` and `gcrypt`. For the corresponding exponentiation algorithms, see Figure 2. For an explanation of the symbols, see Appendix B. For pointers to the original source code, see Footnotes 6 and 9 and 11.

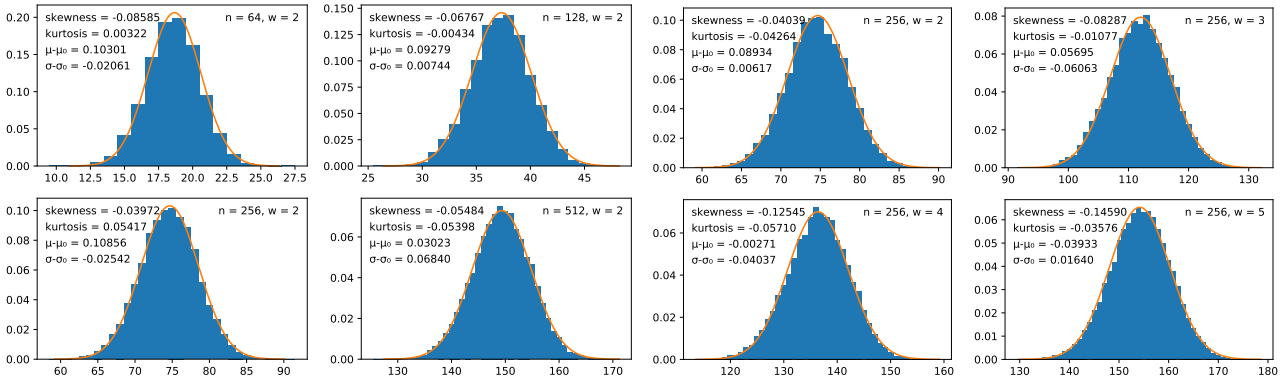

Figure 8: Predicted vs actual distribution of the `gcrypt` leakage entropy $H_w(x)$ for different window sizes w and bit lengths n. The plots show a histogram of measured entropies over 2^{14} random $(n+1)$-bits samples, and the gaussian of mean μ and variance σ^2 predicted in Table 1. The differences $\mu - \mu_0$ (and $\sigma - \sigma_0$) between predicted and observed mean (and standard deviation) are reported together with observed skewness and Fischer's kurtosis.