May the Fourth Be With You: A Microarchitectural Side Channel Attack on Several Real-World Applications of Curve25519

Daniel Genkin
University of Pennsylvania and
University of Maryland
danielg3@cis.upenn.edu

Luke Valenta
University of Pennsylvania
lukev@cis.upenn.edu

Yuval Yarom
University of Adelaide and Data61
yval@cs.adelaide.edu.au

ABSTRACT

In recent years, applications increasingly adopt security primitives designed with better countermeasures against side channel attacks. A concrete example is Libgcrypt's implementation of ECDH encryption with Curve25519. The implementation employs the Montgomery ladder scalar-by-point multiplication, uses the unified, branchless Montgomery double-and-add formula and implements a constant-time argument swap within the ladder. However, Libgcrypt's field arithmetic operations are not implemented in a constant-time side-channel-resistant fashion.

Based on the secure design of Curve25519, users of the curve are advised that there is no need to perform validation of input points. In this work we demonstrate that when this recommendation is followed, the mathematical structure of Curve25519 facilitates the exploitation of side-channel weaknesses.

We demonstrate the effect of this vulnerability on three software applications—encrypted git, email and messaging—that use Libgcrypt. In each case, we show how to craft malicious OpenPGP files that use the Curve25519 point of order 4 as a chosen ciphertext to the ECDH encryption scheme. We find that the resulting interactions of the point at infinity, order-2, and order-4 elements in the Montgomery ladder scalar-by-point multiplication routine create side channel leakage that allows us to recover the private key in as few as 11 attempts to access such malicious files.

CCS CONCEPTS

• **Security and privacy** → **Cryptanalysis and other attacks**; Public key encryption;

KEYWORDS

Side Channel Attacks, Curve25519, Cache-Attacks, Flush+Reload, Order-4 Elements

1 INTRODUCTION

Since their introduction over a decade ago [13, 60, 61], microarchitectural attacks [32] have become a serious threat to cryptographic implementations. A particular threat arises from asynchronous attacks, where the attacker only has to execute a program concurrently with the victim's program (on the same physical CPU) in order to collect temporal information about the victim's behavior. With this temporal information at hand, the attacker can recover the internal workings of the victim.

Because microarchitectural attacks execute on the same processor as the victim, the attacker can only achieve limited temporal resolution. Typically, the attacker can only distinguish between event timings if the events are several hundreds or thousands of execution cycles apart. Consequently, past asynchronous attacks often target key-dependent variations in either the order of high-level operations or in their arguments. More specifically, such attacks usually target the square-and-multiply sequence of the modular exponentiation in RSA [61, 72], ElGamal [55, 75] and DSA [63], or the equivalent double and add sequence of scalar-by-point multiplication in ECDSA [6, 10, 65, 71]. A notable exception is the attack of Pereida García and Brumley [62], which targets the modular inversion used in ECDSA.

With the increased sophistication of microarchitectural attacks, many implementations of cryptographic algorithms have had their side channel robustness investigated, analyzed, and improved. Dealing away with obvious side channel vulnerabilities such as multiplication operations on every set key bit and key-dependent table access, existing implementations have been replaced with more regular algorithms, while newer schemes are designed with side channel resistance in mind from the start.

For elliptic curve cryptography, one approach for reducing the leakage from the scalar-by-point multiplication is to use the Montgomery powering ladder [57]. Performing one point addition operation and one point doubling operation per key bit, regardless of the value of the bit, makes the Montgomery ladder much more resilient to side channel attacks compared to other scalar-by-point multiplication algorithms [49, 59]. Side channel resistance can be further improved by using unified addition formulas, which eliminate operand-dependent branches [18] from point addition and point multiplication routines.

Since modern cryptographic algorithms and implementations have almost completely eliminated high-level key-dependent branches and memory accesses, our work studies the side channel implications of low-level branches typically performed deep inside basic integer arithmetic operations, such as modular reductions.

1.1 Our Contribution

In this paper, we present a new microarchitectural key extraction attack on a highly-regular real-world implementation of Curve25519 [14]. We show that the specific mathematical structure and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '17, October 30-November 3, 2017, Dallas, TX, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134029

recommendations of use for many recently suggested elliptic curves (including Curve25519) actually allow for an *easier* exploitation of low-level side channel weaknesses.

We empirically demonstrate our attack using three real-world applications of Curve25519: git-crypt [1], a git plugin for encrypting git repositories; Pidgin-OpenPGP [39], a plugin for the Pidgin chat client for encrypting chat messages; and Enigmail [66], a popular Thunderbird plugin for email encryption. All of these applications use Libgcrypt [2] as their underlying cryptographic library. Since Libgcrypt's implementation of Curve25519 uses the Montgomery ladder for scalar-by-point multiplication, branchless formulas for point doubling and addition, and built-in countermeasures specially designed to resist cache attacks, our attack cannot observe high-level key dependent behavior, such as key-dependent branches or memory accesses. Instead, we achieve key extraction by combining the specific mathematical structure of Curve25519 with low-level side channel vulnerabilities deep inside Libgcrypt's basic finite field arithmetic operations. By observing the cache access patterns during at most 11 scalar-by-point multiplications, our attack recovers the entire secret scalar within a few seconds. We note that the mathematical structure that enables our attacks in also present in other popular curves such as Curve41417 [15] and Curve448 [44] (Goldilocks curve) when represented in Montgomery form [57].

The Dangers of Order-4 Elements. To extract the secret key, our attack uses side channel leakage produced during decryption with low-order elements, which are present in many recently designed curves. While the side channel risks of order-2 elements are known [37, 73], attacking Montgomery ladder implementations using an order-2 element fails to produce key extraction (see Section 1.2). Instead, our attack takes advantage of the side channel leakage produced by decrypting with an order-4 element. The risks of such elements have been suggested in the past [29], however we are not aware of any demonstration of a practical attack on elliptic curve cryptography that exploits elements of order 4.

The Shortcomings of Existing Countermeasures. Many recently designed elliptic curves support scalar-by-point multiplication using single-coordinate ladders, which forces all received inputs x to be either on the curve or on the "twist" of the curve. Moreover, these curves are also twist-secure, meaning that the twist is also resistant to small subgroup attacks. While these properties mitigate many invalid-curve attacks [14], they also lead implementations to omit *all* input validation, causing them to perform secret-key operations on potentially adversarial inputs. Indeed, while the recommendation to avoid performing validation [12] makes sense in the original context of a carefully designed, constant-time implementation that does not contain side channel weaknesses and is not vulnerable to small subgroup attacks, we argue that this validation improves side-channel resistance for implementations that might not be as carefully designed and implemented for constant-time side-channel resistant execution. This is because the absence of input validation leaves the door open for exploiting other potential side-channel vulnerabilities, as we show in this paper.

Even when countermeasures against low order elements and small subgroup attacks exist, they often do *not* prevent all side-channel attacks. For example, RFC 7748 [53] recommends "ORing all the bytes (of the output) together and checking whether the result is zero, as this eliminates standard side-channels in software implementations." One reason that this countermeasure does not work against side-channel attacks that exploit low-order elements is that it is enacted *after* the scalar-by-point multiplication has been performed, when the leakage is already obtained by the adversary.

Thus, we suggest that implementations reject low-order elements *before* performing sensitive secret key operations, in addition to deploying other side channel countermeasures such as point blinding and exponent randomization. See Section 6 for details.

1.2 Attack Description

We target the ECDH public-key encryption algorithm, as specified in RFC 6638 [48] and NIST SP800-56A [8] and implemented in OpenPGP [22]. We demonstrate our attack on applications that use Libgcrypt, the underlying cryptographic library of GnuPG [2]. The ECDH decryption operation primarily consists of multiplying the secret key (a scalar) by a curve point. For the case of ECDH encryption using Curve25519, Libgcrypt performs the scalar-by-point multiplication using a Montgomery ladder implementation with a single branchless formula for simultaneously computing point addition and doubling. As a protection from cache attacks, Libgcrypt also contains carefully designed routines for performing the swap operations needed to implement the Montgomery ladder. Thus, for every bit of the secret scalar, Libgcrypt performs the same fixed sequence of operations that do not contain any high-level operand-dependent branches or memory accesses.

Unlike traditional Weierstrass and Koblitz curves (such as P192, P224, P256, P384, P521 and Secp256k1) many newly designed curves (such as Curve25519, Curve41417 and Curve448) can be represented in Montgomery form [57] to obtain additional performance speedups. We observe that for a curve to be representable in Montgomery form, it must have an order that is a multiple of four, implying that it contains low-order elements such as an order-2 element G_2 and in many cases an order-4 element G_4. While the existence of order-2 elements is a known side channel risk [29, 37, 73], this risk is slightly mitigated for Montgomery curves using a Montgomery ladder based implementation since the order-2 element is the point of origin ($x = 0, y = 0$). When this point is passed into many implementations of the Montgomery ladder, it causes the result and all computed intermediate values to be zero, irrespective of the secret key [29, 67].

Instead, we perform the ECDH decryption operation using an order-4 element G_4, and take advantage of its representation in projective coordinates. As our analysis in Section 3 shows, using a Montgomery ladder for decrypting G_4 results in curve points of particular mathematical structure appearing as intermediate values during the decryption process. Thus, while the operations performed by the Montgomery ladder scalar-by-point multiplication routine are fixed, our attack links the *operands* of these operations to the secret scalar. Exploiting a side channel weakness in Libgcrypt's modular reduction operation via a cache side channel, we can observe this link and recover the secret scalar.

1.3 Targeted Software and Current Status

In this paper, we focus on the ECDH decryption operation and the Montgomery ladder scalar-by-point multiplication routine as

implemented in Libgcrypt. We used Libgcrypt version 1.7.6 (which is the latest version of Libgcrypt at the time of writing) as supplied as part of the latest Ubuntu 17.04.

We have disclosed our findings to the GnuPG team and are working with them to implement countermeasures against our attack. The vulnerability has been assigned CVE-2017-0379.

1.4 Attack Scenarios

Libgcrypt is part of the GnuPG code base [2], and is used in particular by GnuPG 2.x, a popular implementation of the OpenPGP standard [22] for encrypting files and emails. While our attack requires the decryption of a specific adversarial input (an order-4 element), Libgcrypt is used as the cryptographic back-end for many applications, and as such is often supplied with externally controlled inputs. See [3] for a list of supported Libgcrypt front ends. In Section 4.3, we detail our attacks against the following three front-end applications:

Enigmail. For an attack on encrypted email we use the Thunderbird plugin Enigmail. As Genkin et al. [38] observe, Enigmail automatically decrypts incoming emails by passing them to GnuPG, which uses Libgcrypt as its cryptographic engine. To attack Enigmail, we inject an element of order 4 into Libgcrypt we send the victim a PGP/MIME-encoded e-mail [28], with the element of order-4 as the ciphertext.

Git-crypt. Git-crypt is a git plugin for encrypting files uploaded to git repositories. The aim is to allow for uploading content to a public repository and only authorize a select group of users to access the uploaded content. The user specifies the files to be encrypted, with encryption taking place automatically when pushing changes to the repository. The files are automatically decrypted when changes are pulled from the repository. Git-crypt uses a hybrid encryption scheme. Repository files are encrypted with a randomly-generated AES key. For each authorized user, git-crypt encrypts the AES key with the user's public key and stores the encrypted AES key in the repository. An attacker can thus create a malicious key file with an order-4 element as the ECDH public value in the ciphertext. Uploading this file as the victim's encrypted key file. When the victim pulls the repository, git-crypt automatically tries to decrypt the repository, resulting in an order-4 element being injected into Libgcrypt's scalar-by-point multiplication routine.

Pidgin-OpenPGP. Pidgin is a popular open-source chat application that supports communication across a variety of chat networks [25]. The Pidgin-OpenPGP plugin allows users to encrypt and sign their chat messages using GnuPG [39]. For the attack on Pidgin-OpenPGP, we generate an encrypted chat message and replace the ciphertext with the element of order 4. When the victim receives the message, Pidgin-OpenPGP automatically tries to decrypt it, triggering the attack.

1.5 Attack Feasibility and Limitations

The specific attack that we describe in this paper is realistic in settings where the attacker can share memory with the victim. In particular, we have tested the attack when the attacker process is running as a separate user within the same operating system as the victim process. The Flush+Reload technique we use has also been shown to be effective in PaaS cloud environments, where the attacker and the victim execute within two different containers [76] and in virtualized environments that use memory de-duplication [47, 72]. In these settings, monitoring cache activity during the decryption of only a few chat messages, emails, or git pulls will be sufficient to extract the victim's secret key.

When the attacker and the victim do not share memory, our specific attack does not work. However, we note that use the LLC Prime+Probe attack [55] does not require memory sharing and has been shown effective in cloud environments [46]. Hence, avoiding memory sharing does not guarantee protection.

1.6 Related Work

In this section, we review classes of side-channel attacks that built the foundations for our work.

Physical Side Channel Attacks on ECC Running on Small Devices. Since the first (simulated) attacks of Coron [26], there have been numerous physical side channel key extraction attacks on implementations of elliptic curve cryptography running on small devices. See the surveys [30, 31] and the references therein. However, most of these results either attack naive implementations which contain key-dependent branches (such as the double-and-add algorithm) or take advantage of subtle effects which are only visible at bandwidths exceeding the device's clock rate and are thus impossible to observe using low-bandwidth channels such as the cache side channel.

Two exceptions to the above approach are the Refined Power Analysis attack of Goubin [40] and the Zero-Value Point Attacks of Akishita and Takagi [5] which do seem to use low-bandwidth-observable effects. However both of these attacks require obtaining measurements during the decryption of hundreds of adaptively chosen ciphertexts in order to perform key extraction, making them easily detectable.

Physical Side Channel Attacks on ECC Running on Complex Devices. Key extraction attacks against elliptic curve cryptography implementations running on complex devices have also been demonstrated using both cache and physical side channels. More specifically, electromagnetic key extraction attacks were demonstrated by Genkin et al. [35] on GnuPG's ECDH encryption using a double-and-add 1NAF implementation running on PCs and by Genkin et al. [36] and Belgarric et al. [9] for ECDSA signing routine executed on smartphones.

Attacks on Curve25519. Kaufmann et al. [50] describe an attack on an implementation of Curve25519, which shows timing variations when compiled with the Microsoft Visual C compiler. The attack requires 25000 chosen ciphertexts per each key bit and takes about a month to recover the key.

Duong [27] describes a theoretical attack against Diffie-Hellman with Curve25519 which exploits the lack of public key validation. The attack assumes an adversary that can replace public keys with the element at infinity, in which case the shared secret will be known.

Software-based Side Channel Attacks on Cryptography Running on PCs. Attacks on PC implementations of cryptography have also been demonstrated using software channels such as the

timing channel [20, 21]. Starting with [13, 60, 61, 68, 69] cache attacks have been extensively used to break implementations of cryptographic primitives running on PCs. See Ge et al. [32] for a survey. Brumley and Hakala [19] perform a cache attack on an implementation of ECDSA. The Flush+Reload technique we use has been used for attacks on RSA [72], AES [43, 47], ECDSA [6, 10, 65, 71] and BLISS [41]. The attacks of [4, 71] are of special relevance as they are the only prior works to use microarchitectural attacks to break an implementation that uses the Montgomery ladder. Their attacks, however, exploited a high-level conditional statement that does not exist in the Libgcrypt implementation of the ladder.

Side Channel Attacks on GnuPG. Starting with [38, 58, 72], GnuPG has been targeted by various key extraction attacks. These include attacks on GnuPG's RSA and ElGamal implementations [33, 34, 37, 38, 55, 72] as well as attacks on GnuPG's ECDH encryption [35] and ECDSA signatures implementations [10, 65]. We note that the attacks of [10, 35, 65] are not applicable to the implementation of Montgomery ladder based ECDH encryption that we attack in this paper; after version Libgcrypt 1.6.5, GnuPG no longer uses the Double-and-Add 1NAF implementation attacked by [35], and the attacks of [10, 65] that mount a lattice attack on ECDSA using partially known nonces are not applicable for ECDH.

Attacks Using Low-Order Elements. The risk of performing public key cryptographic operations on elements of low order has been previously demonstrated on various types of public key encryption methods. Yen et al. [73] and Genkin et al. [37] achieve key extraction by using an order-2 element as a chosen ciphertext with implementations of RSA and ElGamal that are based on the square-and-always-multiply exponentiation algorithm. For Elliptic Curve Cryptography, low-order elements have been used for mounting invalid point attacks [17, 54] as well as for fault injection attacks [29]. More specifically, Fan et al. [29] present a theoretical fault injection attack against elliptic-curve Diffie-Hellman key exchange operating over NIST curves, which do not have low-order elements. The attack starts by performing a Diffie-Hellman key exchange using a valid curve point with a short Hamming distance to a point of low order on a twist of the curve. Next, the attacker can (theoretically) inject a carefully-timed fault in the hope of flipping bits in the point's coordinates thus causing the implementation to perform a scalar-by-point multiplication operation with a low-order element on the twist. While Fan et al. [29] do not empirically demonstrate their attack, they do argue, similar to our analysis in Section 3, that the leakage (via physical side channels) resulting from performing the scalar-by-point multiplication with a low order point (order-4 or order-2) should contain enough information to reveal the secret key.

2 PRELIMINARIES

2.1 Elliptic Curve Cryptography

Elliptic curve cryptography (ECC) is an approach to public-key cryptography using elliptic curves over finite fields. The underlying hardness assumption in ECC schemes is the Elliptic Curve Discrete Logarithm Problem (ECDLP): given an elliptic curve group $\mathbb{G}$, a generator G, and a point P it is assumed to be hard to find a scalar k satisfying $\mathsf{P} = [k]\mathsf{G}$. (Here and onward, we use additive group notation, and $[k]\mathsf{G}$ denotes scalar-by-point multiplication further described in Section 2.2 below.) The running time of the best known algorithm for solving ECDLP (without the presence of side channel leakage) is linear with the square root of the order of the subgroup generated by the elliptic curve's generator.

Curve Formulas. Elliptic curves can be expressed with several different representations. The traditional model for elliptic curves is the Weierstrass equation $y^2 = x^3 + ax + b$. Every elliptic curve over a finite field $\mathbb{F}_p$ of a prime order can be converted to this form. Some widely-used examples of curves expressed in this form are the NIST curves from FIPS 186-4 [51] and the Brainpool curves [56].

Alternative elliptic curve representations are often used for speed. Montgomery [57] introduced the eponymous Montgomery form elliptic curves, which are specified using the curve shape $By^2 = x^3 + Ax^2 + x$. A main advantage of curves of this form is that scalar-by-point multiplication can be implemented using only the x coordinate. The single-coordinate version of the Montgomery ladder algorithm for scalar-by-point multiplication requires fewer arithmetic operations than standard Weierstrass scalar-by-point multiplication methods while offering better side channel resistance [49, 59]. The most widely used curve of this form is Curve25519, which was introduced by Bernstein [14]. Other curves that can be specified in this form include Curve41417 [15] and Curve448 [44] (the Goldilocks curve).

Domain Parameters and Cofactors. An elliptic curve group is defined by a set of domain parameters which consists of the following values: p, a prime which defines the prime-order finite field $\mathbb{F}_p$ in which the curve operates; A and B, the coefficients of the curve equation; G, a generator of a subgroup of a prime order on the curve; n, the order of the subgroup that G generates; and h, the cofactor, which is equal to the number of curve points w divided by n. Elliptic curve groups are typically chosen to have small cofactors to limit the number of elements of small order on the curve and to limit the checks required to protect against small subgroup attacks [14]. NIST recommends a maximum cofactor for various curve sizes [51]. The NIST curves over prime order fields specified in FIPS 186-4 are in the Weierstrass form and have a cofactor 1, but curves in the Montgomery form always have a cofactor that is a multiple of 4 [57].

ECDH Encryption. We target the OpenPGP ECDH public-key encryption scheme, ECDH encryption, as specified in RFC 6637 [48] and defined as method `C(1e,1s,ECC CDH)` in NIST SP800-56A [8]. ECDH encryption is a hybrid scheme that combines elliptic curve Diffie-Hellman key exchange with a symmetric-key cipher such as AES. To generate a key pair given an elliptic curve group generator G, Alice first generates a random scalar k as her private key, and computes $[k]\mathsf{G}$ as her public key. To encrypt a message m to Alice, Bob chooses a random scalar k' and computes $[k']([k]\mathsf{G})$, where $[k]\mathsf{G}$ is Alice's public key. Bob uses the result to derive a symmetric encryption key x. The message m is then symmetrically encrypted using x to obtain $\mathsf{Enc}_x(m)$, and the ciphertext is set to $c = (\mathsf{Enc}_x(m), \mathsf{P})$, where $\mathsf{P} = [k']\mathsf{G}$ is the ephemeral public key, which also plays the role of a ciphertext in our chosen ciphertext attack. To decrypt c, Alice computes $[k](\mathsf{P}) = [k]([k']\mathsf{G})$. She then derives from it a symmetric key x'. This key can then be used to

symmetrically decrypt $\mathsf{Enc}_x(m)$ to get message m'. By the commutative property of elliptic curve scalar-by-point multiplication $[k]([k']\mathsf{G}) = [k']([k]\mathsf{G})$. Hence we have $x' = x$ and $m' = m$.

Point Representation. Elliptic curve points can be represented in many different forms. The canonical representation uses the *affine coordinates*, where a point on the curve is represented by a pair of integers (x, y) that satisfy the curve equation. However, this representation requires an expensive field inversion operation to add two elliptic curve points. Using *projective coordinates*, where a point (x, y) is represented by the triplet (X, Y, Z), where $(x, y) = (X/Z, Y/Z)$ for $Z \neq 0$, obviates the field inversion [23]. A special "point at infinity" is represented by $Z = 0$. Points can have many different representations depending on the value of Z, and this equivalence class is denoted $(X : Y : Z)$.

Optimization for Montgomery Coordinates. Elliptic curve points support arithmetic operations based on the elliptic curve's group addition law. For Montgomery curves, the group addition law which adds two projective points (X_0, Y_0, Z_0) and (X_1, Y_1, Z_1) to produce the sum (X_s, Y_s, Z_s) computes X_s and Z_s without using the y-coordinates at all. This allows us to represent a point $\mathsf{P} = (x, y)$ without the y-coordinate using the *projective Montgomery coordinates* $\mathsf{P} = (X, Z)$, where $x = X/Z$ for $Z \neq 0$. This form loses some information: there is no way to distinguish between the points (x, y) and $(x, -y)$ since they both have the representation (X, Z), but this is not an issue for the application of ECDH key exchange. These x-coordinate point operations on Montgomery curves are extremely fast, and they also allow points to be represented with only half as many bits, so that a public key can be represented with only $x = X/Z$ instead of (x, y).

Low-Order Elements. Every elliptic curve group has an order-1 element called the identity element, which we will denote G_1. G_1 is often called the "point at infinity". For every prime divisor p_i of the group order w, there exists an element on the curve with order p_i. Because Montgomery curves must have a cofactor that is a multiple of 4, such curves must contain an element G_2 of order 2. (That is because 2 is a prime that divides the group order). Next, since 4 divides the group order for Montgomery curves, there is also a subgroup of order 4. This does not imply that the curve has an order-4 element, but this is often the case. We denote order-4 elements as G_4 when they exist. In the Montgomery projective coordinates, the point at infinity is represented by $(X \neq 0 : Z = 0)$, the element of order 2 by $(X = 0 : Z \neq 0)$. The coordinates of the elements of order 4, when the exist, depend on the specific curve.

Curve25519. Introduced by Bernstein [14], Curve25519 is specified in the Montgomery form as $y^2 = x^3 + 486662x^2 + x$ over the field with prime modulus $p = 2^{255} - 19$. Curve25519 has a cofactor 8, meaning that the order of the curve is $8 \cdot n$, for a prime n. Curve25519 also has two order-4 elements with affine coordinates $(x = 1, y = \pm\sqrt{486664})$. Both these elements are represented in the Montgomery projective coordinates by $(X = \lambda : Z = \lambda)$, where $\lambda \neq 0$. The curve has no element with affine x-coordinate $x = -1$, however such elements, represented by $(X = \lambda : Z = -\lambda)$ exist on the twist of the curve, where they have an order 4. For the purposes of this work, the elements of order 4 on the curve and on the curve's twist behave in a similar manner and we refer to all of them as G_4.

When introduced, Curve25519 timings were more than twice as fast as previously reported times for elliptic curves of an equivalent security level, while also including "free key compression, free key validation, and state-of-the-art timing-attack protection" [14]. Implementations are not required to perform key validation, since by definition secret keys have the low-order bits set to zero, so there is no risk of leaking these bits in a small subgroup attack [14]. Moreover, the use of the Montgomery ladder scalar multiplication algorithm provides side-channel resistance [49, 59]. Curve25519 was standardized by RFC 7748 [53], and is implemented in a wide variety of protocols and software [45].

Public Key Validation for Curve25519. Part of the appeal of using Diffie-Hellman with Curve25519 is that implementations are not required to validate public keys, including the ephemeral public key in ECDH. Not only is validation not required, but the recommendation is to not validate public keys because "The Curve-25519 function was carefully designed to allow all 32-byte strings as Diffie-Hellman public keys" [11]. This recommendation is the subject of debate, where proponents claim that key validation is not required [64] whereas critics maintain that the recommendation is risky [7, 27].

In this work we identify another risk associated with this recommendation. The recommendation implicitly assumes that the implementations of the curve functions and of the underlying field arithmetic are constant-time. Our attack exploits the failure to reject low-order elements, combined with a non-constant-time implementation of the underlying field arithmetic.

2.2 Scalar-by-Point Multiplication

Scalar-by-point multiplication is one of the core operations in elliptic curve cryptography. Given a positive scalar k and an elliptic-curve point P, the scalar-by-point multiplication operation adds P to itself k times to produce the point $[k]\mathsf{P}$. There are several popular methods for implementing scalar-by-point multiplication in the literature.

Double-And-Add. The simplest method is the double-and-add method, which is similar to the square-and-multiply algorithm in modular exponentiation. For each bit of the scalar k, the algorithm performs one doubling operation. Additionally, in case the bit is set, the algorithm also performs an addition operation. However, the fact that the sequence of doubles and adds performed by this algorithm leaks the bits of k is a major side channel weakness [26].

Montgomery Ladder. Implementations that wish to protect against side channel attacks can use the Montgomery ladder algorithm [57] for scalar-by-point multiplication. This algorithm performs the same number of addition and double operations regardless of the value of the scalar k. As such, the algorithm can be implemented without any key-dependent branches, making it more side channel resistant [49, 59].

The Montgomery ladder is based on the observation that given $[\lfloor n/2 \rfloor]\mathsf{P}$ and $[\lfloor n/2 \rfloor + 1]\mathsf{P}$, we can easily calculate $[n]\mathsf{P}$ and $[n+1]\mathsf{P}$. More specifically, if we have $\mathsf{R}_0 = [\lfloor n/2 \rfloor]\mathsf{P}$ and $\mathsf{R}_1 = [\lfloor n/2 \rfloor + 1]\mathsf{P}$, for even n we calculate $\mathsf{R}_1 \longleftarrow \mathsf{R}_0 + \mathsf{R}_1, \mathsf{R}_0 \longleftarrow [2]\mathsf{R}_0$, and for odd n we use $\mathsf{R}_0 \longleftarrow \mathsf{R}_0 + \mathsf{R}_1, \mathsf{R}_1 \longleftarrow [2]\mathsf{R}_1$. We note that in both cases we perform one addition and one doubling operation and the only difference between the cases is the roles that the variables play.

Algorithm 1 Montgomery ladder scalar-by-point multiplication operation.

Input: A positive scalar k and an elliptic-curve point P, where $k = \sum_{i=0}^{n-1} 2^i \cdot k_i$ and $k_i \in \{0, 1\}$ for all $i = 0, \cdots, n-1$.
Output: $[k]\mathsf{P}$.

```
1:  procedure MONTGOMERY_LADDER(k, P)
2:      R0 ← G1          ▷ G1 represents the order-1 identity element
3:      R1 ← P
4:      dif_x ← P.x
5:      for i ← n − 1 to 0 do
6:          b ← k_i
7:          Q0, Q1 ← CONDITIONAL_SWAP(R0, R1, b)
                                  ▷ Constant time swap when b = 1
8:          S0, S1 ← MONTGOMERY_STEP(Q0, Q1, dif_x)
                                  ▷ S0 = [2]Q0, S1 = Q0 + Q1
9:          R0, R1 ← CONDITIONAL_SWAP(S0, S1, b)
                                  ▷ Constant time swap when b = 1
10:     return R0
```

Naive implementations of the Montgomery ladder scan the scalar from the most significant bit to the least significant. For each bit, they conditionally execute one of the computations specified above, based on the value of the bit. However, such implementations are known to be vulnerable to side channel attacks [4, 71]. A common mitigation, which Libgcrypt uses, is to conditionally swap the values of R_0 and R_1 before and after the computation. Algorithm 1 shows the pseudocode of such an implementation. The conditional swaps can be implemented using bit manipulations to avoid any branches or memory access operations that depend on secret-key bits. Such implementations are protected against timing and cache-based side channel attacks.

As mentioned earlier, one of the advantages of Montgomery curves is that the *Montgomery step*, which sums its two arguments and doubles one of them (Line 8 of Algorithm 1), can be calculated efficiently using only the x-coordinates in the projective Montgomery form. Algorithm 2 shows a pseudo code of an implementation of the Montgomery step. We note that the implementation does not contain any branches or memory accesses that depend on secret values.

2.3 Libgcrypt's Implementation

We now describe Libgcrypt's implementation of Montgomery curves and point operations. Libgcrypt stores points using projective Montgomery coordinates. Each point is represented as a pair (X, Z), where each element is a large integer stored using Libgcrypt's arithmetic library, MPI. MPI stores large integers as arrays of *limbs*, which are 64-bit words on the x86-64 architecture used in our tests. For Curve25519, field elements are calculated modulo $2^{255} - 19$ hence integers can have up to four limbs. Multiplication and squaring operations on field elements can be up to 510 bits long before modular reduction and may require 8 limbs for storage.

Libgcrypt's Scalar-by-Point Multiplication. Libgcrypt uses the Montgomery ladder (Algorithm 1) for scalar-by-point multiplication. In order to protect from side channel attacks, Libgcrypt's implementation uses a side-channel-resistant constant-time point swap function to set the inputs and outputs of the MONTGOMERY_STEP function based on the value of the secret key bit in each loop iteration.

Algorithm 2 Libgcrypt's Montgomery step operation (simplified).

Input: Two points $\mathsf{Q}_0 = (X_0, Z_0)$ and $\mathsf{Q}_1 = (X_1, Z_1)$ in projective coordinates on an elliptic-curve based group of order p, and `dif_x` which should be equal to the difference in x-coordinates of the input points.
Output: Two points $\mathtt{Dbl} = (X_d, Z_d)$ and $\mathtt{Sum} = (X_s, Z_s)$ in projective coordinates such that $\mathtt{Dbl} = [2]\mathsf{Q}_0$ and $\mathtt{Sum} = \mathsf{Q}_0 + \mathsf{Q}_1$.

```
1:  procedure MONTGOMERY_STEP(Q0, Q1, dif_x)
2:      l1 ← X1 + Z1 mod p
3:      l2 ← X1 − Z1 mod p
4:      l3 ← X0 + Z0 mod p
5:      l4 ← X0 − Z0 mod p
6:      l5 ← l4 l1 mod p
7:      l6 ← l3 l2 mod p
8:      l7 ← l3^2 mod p
9:      l8 ← l4^2 mod p
10:     l9 ← l5 + l6 mod p
11:     l10 ← l5 − l6 mod p
12:     Xd ← l7 l8 mod p
13:     l11 ← l7 − l8 mod p         ▷ l11 = 4 X0 Z0 (see Equation 5)
14:     Xs ← l9^2 mod p
15:     l12 ← l10^2 mod p
16:     l13 ← l11 · (A − 2)/4 mod p  ▷ A = 486662 for Curve25519
17:     Zs ← l12 · dif_x mod p
18:     l14 ← l7 + l13 mod p
19:     Zd ← l14 l11 mod p
20:     return ((Xd, Zd), (Xs, Zs))
```

Libgcrypt's Montgomery Step Implementation. The MONTGOMERY_STEP function receives inputs Q_0, Q_1, and `dif_x` which is the affine x-coordinates of the input point P. It returns $([2]\mathsf{Q}_0, \mathsf{Q}_0 + \mathsf{Q}_1)$. Doubling of Q_0, represented in the projected Montgomery coordinates as (X_0, Z_0), is computed by

$$X_d = (X_0 + Z_0)^2 (X_0 - Z_0)^2 \tag{1}$$

$$Z_d = (4X_0Z_0)((X_0 + Z_0)^2 + ((\mathsf{A} - 2)/4) * (4X_0Z_0)), \tag{2}$$

and the Montgomery addition operation for computing $\mathsf{Q}_0 + \mathsf{Q}_1$ performs

$$X_s = ((X_0 - Z_0)(X_1 + Z_1) + (X_0 + Z_0)(X_1 - Z_1))^2 \tag{3}$$

$$Z_s = \mathtt{dif_x}((X_0 - Z_0)(X_1 + Z_1) - (X_0 + Z_0)(X_1 - Z_1))^2, \tag{4}$$

where A is a curve parameter.

Algorithm 2 shows a simplified version of Libgcrypt's implementation of the MONTGOMERY_STEP algorithm for projective Montgomery coordinates. The actual Libgcrypt implementation re-uses the coordinates of the input variables for temporary storage during the computation and precomputes $(A - 2)/4$. For clarity, we replace these with local variables and explicit formulas.

We pay special attention to the multiplication on Line 19, which we target in Section 3. In particular we note that the value l_{11}

Algorithm 3 Libgcrypt's modular reduction operation (simplified).

Input: Two integers x and m, represented as a sequence of limbs $x_0 \ldots x_{l-1}$ and $m_0 \ldots m_{k-1}$.
Output: $x \bmod m$.

```
 1: procedure MODULAR_REDUCTION(x, m)
 2:     l ← SIZE_IN_LIMBS(x)
 3:     k ← SIZE_IN_LIMBS(m)
 4:     if l < k then
 5:         return x                    ▷ Early exit if x is smaller than m
 6:     for i ← l − 1 downto k − 1 do
 7:         q ← (x_i · 2^64 + x_{i−1})/m_{k−1}      ▷ Estimate quotient q
 8:         if q(m_{k−1} · 2^128 + m_{k−2}) > x_i · 2^128 + x_{i−1} · 2^64 + x_{i−2} then
 9:             q ← q − 1               ▷ If q is too large, adjust estimate
10:         x ← x − q · m · 2^{64(i−k)}          ▷ Subtract from x
11:     return x                        ▷ x holds the remainder
```

computed in Line 13 of Algorithm 2 is

$$\begin{aligned} l_{11} &= l_7 - l_8 = l_3^2 - l_4^2 \\ &= (X_0 + Z_0)^2 - (X_0 - Z_0)^2 \\ &= (X_0^2 + 2X_0Z_0 + Z_0^2) - (X_0^2 - 2X_0Z_0 + Z_0^2) \\ &= 4X_0Z_0. \end{aligned} \tag{5}$$

Libgcrypt's Modular Reduction Routine. After each arithmetic operation in MONTGOMERY_STEP (Algorithm 2), the result is reduced modulo p using Libgcrypt's modular reduction function. Algorithm 3 shows a simplified version of this function, which uses the classical long division algorithm formalized by Knuth [52]. The quotient q is estimated in each iteration of the loop and adjusted if the initial estimate was off by 1. Then, the appropriate multiple of q is subtracted from the input before execution returns to the top of the loop. Notice that code execution only reaches the body of the main `for` loop at Line 6 when the number of limbs of the number being reduced, is equal to or greater than the number of limbs of m, the modulus. Otherwise, when the input is shorter, and therefore guaranteed to be smaller, than m, the algorithm exits early without performing a modular reduction.

As we show in Section 3, detecting the early exit in Line 5 shows that the value $l_{14} \cdot l_{11}$, as computed in Line 19 of Algorithm 2, is smaller than the order of the group, p, allowing the attacker to determine the order of the group elements being multiplied. Using this information, the attacker can then extract the bits of the secret scalar k, resulting in a complete key extraction.

3 CRYPTANALYSIS

In this section we present our non-adaptive chosen ciphertext side-channel attack against Libgcrypt's ECDH implementation. Since the sequence of arithmetic field operations performed by the Montgomery ladder is not key-dependent, we wish to find some elliptic curve point P that, when multiplied by the secret key k, will cause an observable correlation between the intermediate values used as *operands* of these arithmetic operations and the bits of k. We then use a side-channel attack to obtain information about the values of the operands of these operations, achieving complete key recovery.

Chosen Ciphertext as Order-2 Element. Previous work [37, 73] used an order-2 element as a chosen ciphertext for attacks on RSA and ElGamal in order to create an observable correlation between the operands of the arithmetic operations performed by the exponentiation routine and the secret key. Unfortunately, this approach does not work in our case. The order 2 element is $\mathsf{G}_2 = (X = 0, Z \neq 0)$. If we use $\mathsf{P} = \mathsf{G}_2$, we have `dif_x` $= \mathsf{G}_2.x = 0$ in Line 4 of Algorithm 1. As Ransom [67] observes, this is an exceptional case that causes incorrect results for the Montgomery addition computed by MONTGOMERY_STEP. More specifically, because Z_s is set to 0 on Line 17 of Algorithm 2, the sum $(X_s, Z_s) = \mathsf{G}_1 + \mathsf{G}_2$ is computed as $(X = 0, Z = 0)$, which is illegal in the Montgomery projective representation. Subsequent iterations of the loop in Algorithm 1 treat this undefined point as G_1 instead of G_2. The consequence of this irregularity is that when we use $\mathsf{P} = \mathsf{G}_2$, all of the intermediate values in Algorithm 1 are the invalid point irrespective of the secret key bits. We stress that the irregularity in the implementation only happens when $\mathsf{P} = \mathsf{G}_2$. For every other value of P, the point addition will involve at least one value that is neither G_1 nor G_2 and the results of the algorithm are correct.

3.1 Long and Short Modular Reductions and Order-2 Elements

Our attack exploits the early exit in Line 5 of Algorithm 3. We say that the modular reduction in $l_{14} \cdot l_{11} \bmod p$ (Line 19 of Algorithm 2) is short when the number of limbs in $l_{14} \cdot l_{11}$ is smaller than the number of limbs in p, causing an early exit. Otherwise, we say that modular reduction in $l_{14} \cdot l_{11} \bmod p$ is long. We later show that by monitoring the cache, we can detect the early exit. We now proceed to describe when early exits occur and how we can recover the key based on them.

Order-1 and Order-2 Arguments Imply Short Modular Reductions. Consider the case where the first argument Q_0 to MONTGOMERY_STEP (Algorithm 2) is either the order-1 element G_1 or the order-2 element G_2. As mentioned in Section 2.1, for G_1 we have $(X_0 \neq 0, Z_0 = 0)$ and for G_2 we have $(X_0 = 0, Z_0 \neq 0)$. In both cases the value $l_{11} = 4X_0Z_0$ (see Equation 5) computed in Line 13 is equal to 0. Next, since l_{11} is zero we obtain that the value $l_{14} \cdot l_{11}$ computed in Line 19 is also equal to 0. Finally, since the representation of 0 consists of only one limb, the condition in line Line 4 of Algorithm 3 is true, causing an early exit on Line 5, and the modular reduction in $Z_d \leftarrow l_{14} \cdot l_{11} \bmod p$ is short.

Order-4 Arguments Typically Imply Long Modular Reductions. As we discuss in Section 2.1, an order-4 element G_4 has the form $(X = \lambda, Z = \pm\lambda)$, with $\lambda \in [1, \ldots, p-1]$. The fact that the affine point $x = 1$ can be expressed in this way with projective coordinates actually helps our attack. As above, consider passing the order-4 element $(X_0 = \lambda, Z_0 = \pm\lambda)$ as the Q_0 argument of MONTGOMERY_STEP. We now look at the values of l_{11} and l_{14} used in Line 19. From Equation 5 we have $l_{11} = 4X_0Z_0 = \pm 4\lambda^2$.

For l_{14} we have:

$$\begin{aligned} l_{14} = l_7 + l_{13} \bmod p &= l_3^2 + l_{11} \cdot (A-2)/4 \bmod p \\ &= (X_0 + Z_0)^2 + 4\lambda^2 \cdot (A-2)/4 \bmod p \\ &= \lambda^2 \cdot (A \pm 2) \bmod p \end{aligned}$$

where the ±2 depends on whether G_4 is on the curve or on its twist, i.e. whether $Z_0 = \lambda$ or $Z_0 = -\lambda$. Consequently, if $\lambda < (2^{192}/(A+2))^{1/4}$ or $p-\lambda < (2^{192}/(A+2))^{1/4}$, we have that $l_{14}l_{11} < 2^{192}$ and the reduction in Line 19 is short. Otherwise, we have that $l_{14}l_{11} > 2^{192}$ and the reduction is long, except with a negligible probability of $2^{192-510}$.

3.2 Order-4 Element as a Chosen Ciphertext

We now consider decryption when the adversary sends an element of order 4 G_4 as chosen ciphertext. Recall that there are two elements of order 4, an element on the curve, with affine x-coordinate of 1 and an element on the twist with x-coordinate of −1. However, for our purposes these elements behave the same so we refer to both as G_4. The relevant rules of point addition for order-4 elements are as follows:

$$[2]\mathsf{G}_4 = \mathsf{G}_2$$
$$\mathsf{G}_1 + \mathsf{G}_4 = \mathsf{G}_4$$
$$\mathsf{G}_2 + \mathsf{G}_4 = \mathsf{G}_4$$

Montgomery Ladder Invariant Revisited. Next, we recall that in the Montgomery ladder, the difference in affine coordinates of the tracked values R_0 and R_1 is P, the input point. Based on the addition rules above, when the input point is G_4, as is the case in our attack, one of R_0 and R_1 must be G_4 and the other must be either G_1 or G_2.

Determining Key Bits. We now show how, an attacker that knows the value of the i-th key bit, k_i can leverage the side channel leakage to learn the value of bit k_{i-1}. Repeating this argument for all of the bits of k results in a complete key extraction. Indeed, note that based on the invariant and the rules above, every time the MONTGOMERY_STEP function is executed in Algorithm 1, the output value $\mathsf{S}_1 = \mathsf{Q}_0 + \mathsf{Q}_1$ must be an order 4 element G_4. Next, since $\mathsf{S}_1 = \mathsf{G}_4$ the Montgomery ladder invariant implies that S_0 is either G_1 or G_2. The values held by S_0 and S_1 after processing bit k_i will propagate to the Montgomery step of bit k_{i-1} as the values held by Q_0 and Q_1, possibly getting swapped at two locations: Line 9 if bit k_i is set, and Line 7 in the next loop iteration in case bit k_{i-1} is set.

Thus, we consider the following two cases based on the values of the key bits k_i and k_{i-1}:

(1) $\boldsymbol{k_{i-1} = k_i}$. When propagating from S_0 and S_1 to Q_0 and Q_1, the values will either be swapped twice if $k_i = k_{i-1} = 1$, or not swapped at all, when $k_i = k_{i-1} = 0$. In both cases, $\mathsf{Q}_0 \in \{\mathsf{G}_1, \mathsf{G}_2\}$ and $\mathsf{Q}_1 = \mathsf{G}_4$. As stated in Section 3.1, having $\mathsf{Q}_0 \in \{\mathsf{G}_1, \mathsf{G}_2\}$ implies that the modular reduction in Line 19 of Algorithm 2 performed during the processing of k_{i-1} will be short.

(2) $\boldsymbol{k_{i-1} \neq k_i}$. When propagating from S_0 and S_1 to Q_0 and Q_1, the values will be swapped exactly once, since only one of k_i and k_{i-1} is set. In either case, $\mathsf{Q}_0 = \mathsf{G}_4$ and $\mathsf{Q}_1 \in \{\mathsf{G}_1, \mathsf{G}_2\}$. As stated in Section 3.1, having $\mathsf{Q}_0 = \mathsf{G}_4$ implies that the modular reduction in Line 19 of Algorithm 2 performed during the processing of k_{i-1} will be long.

Hence, when the attacker knows k_i, observing the length of the modular reduction will allow the attacker to determine the value of k_{i-1}. This culminates in an easy procedure for recovering bits directly from a sequence of short and long reductions: a short reduction means that the current bit is the same as the previous bit, and a long reduction means that the current bit is the complement of the previous bit.

Key Extraction. Confirming the above, in Figure 1 we show a sequence of modular reductions performed in Line 19 during 39 loop iterations of Montgomery ladder (Algorithm 1). As can be seen, some modular reductions are long while others are short, which clearly indicates the leakage of secret key material.

Assuming that the bit preceding the captured sequence was 0, we apply our easy rule: a long reduction implies that the value of the next bit (the first captured) is 1. The next modular reduction is long again, and we can conclude that the bit is 0. The third reduction is short, indicating that the value of the bit remains 0 and so forth.

Small values of $\boldsymbol{\lambda}$. A minor limitation of the above approach is that, as discussed above, when doubling G_4 with a small λ, the modular reduction will be short. Experimentally, we find that during most of the algorithm the probability of this happening is negligible. However, when Libgcrypt initializes R_1, it sets $\lambda = 1$. Nevertheless, the length of λ increases rapidly, reaching the full size of four limbs (255 bits) within four loop iterations. However, during these first four iterations the value of λ is small, hence our attack is unable to determine the first four key bits used during these iterations.

4 EXPERIMENTAL RESULTS

4.1 Attack Technique

For the side channel, we use the Flush+Reload attack [72] in conjunction with the amplification attack of Allan et al. [6]. Microarchitectural attacks such as Flush+Reload leak information on programs by monitoring the effects that executing a program has on the state of the components of the processor. See Ge et al. [32] for a survey of published microarchitectural attacks. In particular, the Flush+Reload attack leaks information by monitoring the presence of memory locations in the cache.

The Flush+Reload Attack. The Flush+Reload attack consists of two phases. In the *flush* phase, the attacker evicts the contents of one or more monitored memory addresses from the cache. This is typically achieved by using a dedicated instruction, such as the x86 `clflush`, but in the absence of such an instruction, the attacker can use other mechanisms to achieve eviction [42, 74]. After the flush phase is completed the attacker waits for a short while to allow the victim time to execute. Then, during the *reload* phase, the attacker reads the contents of the memory addresses, measuring the time it takes to perform the read.

In case the victim accesses one or more of the monitored memory addresses between the flush and the reload phases, the contents of these addresses will be cached again causing the attacker's reads to be fast. Conversely, in case the victim does not access a monitored memory address, the contents will not be cached, causing the attacker's read to take longer. Performing the attack repeatedly, the attacker can trace the victim's memory accesses to specific addresses over time. In case the monitored memory addresses are part of the victim's code, the attacker learns some information about the victim's execution patterns.

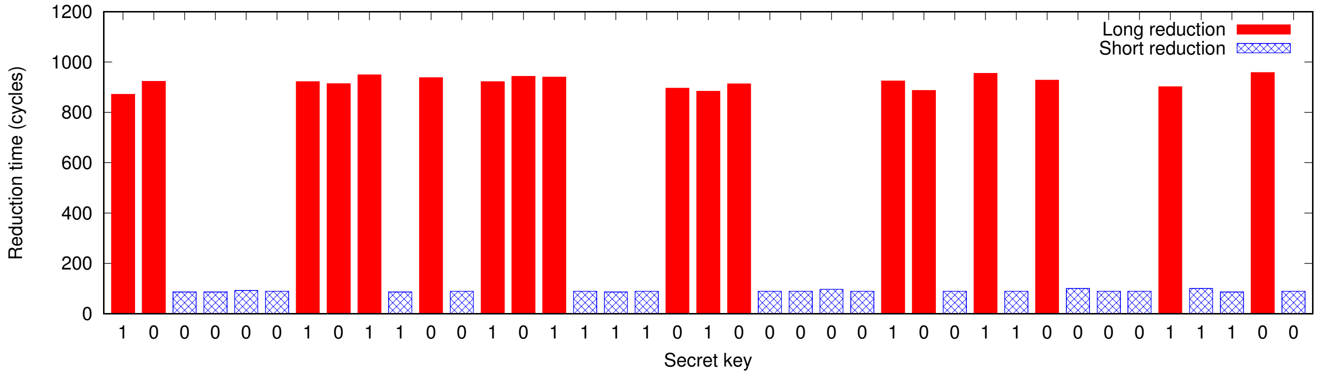

Figure 1: Trace (excluding four first bits) of scalar-by-point multiplication of a secret key with an element of order 4. We can learn the bits of the scalar (shown on the x-axis) from the sequence of long and short modular reduction operations: a short reduction implies that the current bit is the same as the previous bit, whereas a long reduction means that the current bit is the complement of the previous bit.

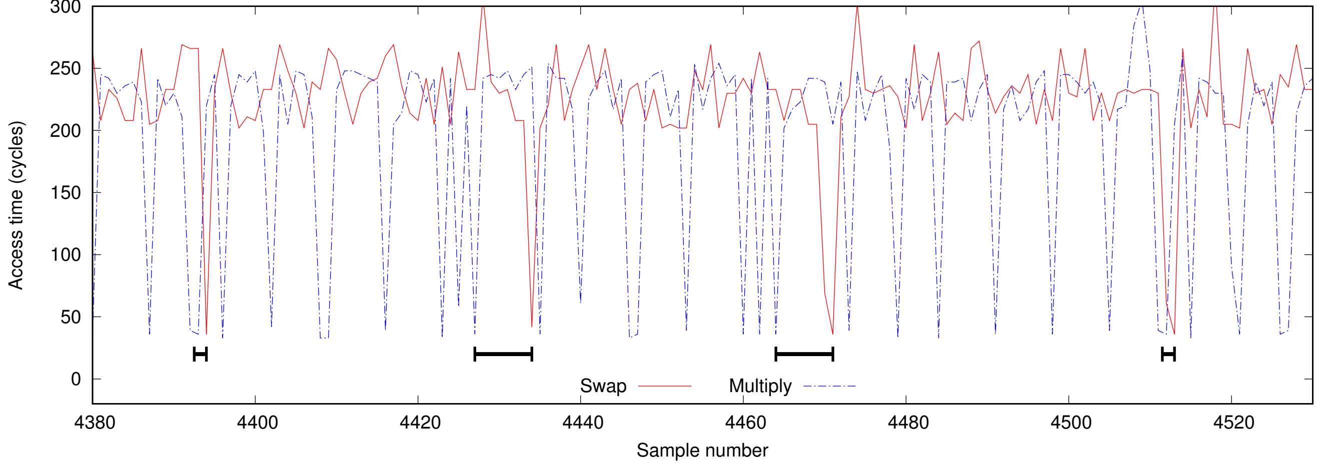

Figure 2: Memory access times of the Flush+Reload attack, with the lengths of the horizontal bars corresponding to the lengths of modular reductions. The results were obtained by flushing and reloading four memory locations, two within the constant-time swap code and two within the multiplication code. In each sample, we perform a flush followed by a reload for each of these four memory locations, measuring access times. We show the minimum of the access times for the two memory locations in the constant-time swap code in red, and the minimum of the access times for the two memory locations in the multiplication code in blue.

The Amplification Attack. Because the Flush+Reload attack executes concurrently with the victim, the Flush+Reload attack has a limited temporal resolution. To improve the attack resolution, Allan et al. [6] suggest slowing the victim down. At high level, this is done by identifying frequently accessed, or "hot", sections of the victim code and then repeatedly evicting these sections from the cache. Next, in order to execute code that has been evicted, the victim has to wait until the processor loads the code from the main memory. This, in turn, increases the time it takes the victim to execute each operation and provides a larger time window for the attacker to make accurate side-channel measurements. To evict the code from the cache, Allan et al. [6] use the `clflush` instruction, hence like the Flush+Reload attack, amplification only works when the victim and the attacker share memory.

4.2 Attacking the Scalar-by-Point Multiplication

Experimental Setup. We target Libgcrypt's implementation of the Montgomery ladder scalar-by-point multiplication routine. We first demonstrate the attack's feasibility by directly invoking Libgcrypt's scalar multiplication on an order-4 element. As described in Section 1.3, we target Libgcrypt 1.7.6, which is the latest version of Libgcrypt at the time of writing this paper, as supplied in the latest Ubuntu 17.04. Below, all experiments and cache attacks

were performed on a Dell Optiplex 9010 desktop, equipped with an i7-3770 3.4 GHz processor and 8GB of memory, running unmodified Ubuntu 17.04. To mount the Flush+Reload attack, we used the `FR-trace` utility of the Mastik toolkit [70]. `FR-trace` provides a command-line interface for performing the Flush+Reload attacks as well as support for the amplification attack of Allan et al. [6].

Applying the Flush+Reload Attack. To extract information about whether the modular reduction in Line 19 of Algorithm 2 was long or short during each iteration of the main loop of Algorithm 1, we set `FR-trace` to monitor four memory locations within the Libgcrypt library. Two of these locations are within the field multiplication code (which executes before the modular reduction operation) and the other two are within the CONDITIONAL_SWAP function (which executes after the modular reduction operation). As Allan et al. [6] observe, monitoring two memory locations with the same functionality reduces the probability that the attack will miss a memory access due to overlap between the victim's memory accesses during the attacker's reload phase. To improve our ability to detect the length of the modular reduction operation, we use the amplification attack of [6] to repeatedly evict the code of the operation. This increases the time to perform modular reduction by a multiplicative factor of 11.1.

Recall that our attack correlates the bits of the secret key and the time it takes to perform the modular reduction in Line 19 of Algorithm 2. Since this modular reduction operation is executed between our two measurement points, we expect that the temporal separation between the two measurements will reveal the length of the modular reduction, i.e. whether it is long or short.

Trace Analysis. Figure 2 shows a sample of a trace of a scalar multiplication. For each measured functionality (field multiplication code and the CONDITIONAL_SWAP function) we plot the shorter of reload times of the two measurement locations. Recall that the reload time of a monitored location is shorter following a victim's access to that location. In our test environment, we find that loads from memory take over 150 cycles, whereas loads from the cache take less than 100 cycles. Thus, whenever the reload takes below 100 cycles we can assume that the victim has accessed the monitored location.

Observing Swap Operations. Looking at Figure 2, we see a sequence of "dips" which indicate various victim accesses. Dips in the swap line (solid red) indicate that the victim performed the constant time swap operation. Due to the low temporal resolution of the Flush+Reload attack, we are unable to distinguish between the swap that occurs at the end of one loop iteration of Algorithm 1 and the swap at the start of the next one. Hence, the four dips visible in the solid red line show the times where processing of one scalar bit ends and processing of the following bit starts during the main loop of Algorithm 1.

Observing Multiplication Operations. Dips in the multiply line (dashed blue) indicate times when the victim performed the multiplication operations in Algorithm 2. Gaps between the dips correspond to all of the other operations that the algorithm performs. Due to the amplification attacks, the dominant component in the gaps is the time it takes to compute the modular reduction.

The amplification attack only amplifies the main loop of the modular reduction. Hence, when Algorithm 3 exits early, its timing is not affected by the attack. Due to the limited temporal resolution of the Flush+Reload attack, in the case of a short reduction, the attack is unable to distinguish between the timing of the multiplication in Line 19 of Algorithm 2 and the following swap operation.

Observing Long and Short Modular Reductions. We now turn our attention to the gap between the last *observed* multiplication operation and the following swap. These are marked with black horizontal bars. We note that in the case of a long reduction this gap is due to the modular reduction in Line 19 of Algorithm 2. However, as discussed above, in the case of a short reduction, Flush+Reload samples this multiplication in the same time as the swap operation. Hence, the gap is due to the preceding multiplication, in Line 16. Because one of the multiplicands in Line 16 is short, the multiplication result is short and the modular reduction in this case is faster than that of a long reduction.

As we can see, Figure 2 shows one short gap, followed by two long and another short gap. These correspond to long and short modular reductions. Hence, by measuring the length of the gap, the attacker can recover the information on the length of the last modular reduction, and from it recover the bits of the key.

Handling Measurement Errors. Side-channel attacks rarely produce error-free results. To measure the number of errors in our attack, we captured 1000 traces and compared with the ground truth. On average, there are 3.8 errors in a trace. See Figure 4 for the distribution of the number of errors in traces.

Overall Attack Performance. To correct the errors, we selected five arbitrary traces (see Figure 3), aligned them manually (about 10 minutes of wall-clock time) and used a simple majority rule to decide the length of each modular reduction operation. From this we were able to deduce for all but the leading four key bits whether the modular reduction in Line 19 of Algorithm 2 was long or short. Finally, applying the cryptanalysis from Section 3, we successfully recovered all but the first four bits of a randomly generated `Curve25519` scalar. The leading bits can then easily be found using exhaustive search.

4.3 Attacking Applications

We now turn our attention to attacking applications that use Libgcrypt. We attack three applications: git-crypt [1], Pidgin's OpenPGP plugin [25, 39], and Enigmail [66]. We first describe these applications with a focus on how they use encryption and the attack vector. We then describe the attack results.

4.3.1 Git-crypt

Git-crypt is a plugin for the git revision control system, used to selectively encrypt files in a repository. When initialized, git-crypt selects a random AES key, which is used for encrypting the files stored in the git repository. To publish the repository's AES key, git-crypt creates encrypted key files using the Gnu Privacy Guard (GnuPG) software. Each of the key files is encrypted with the public key of an authorized user and is stored in the repository. When git processes modifications to an encrypted file, it invokes git-crypt, which calls GnuPG to retrieve the repository's AES key. Git-crypt then encrypts or decrypts the modified file.

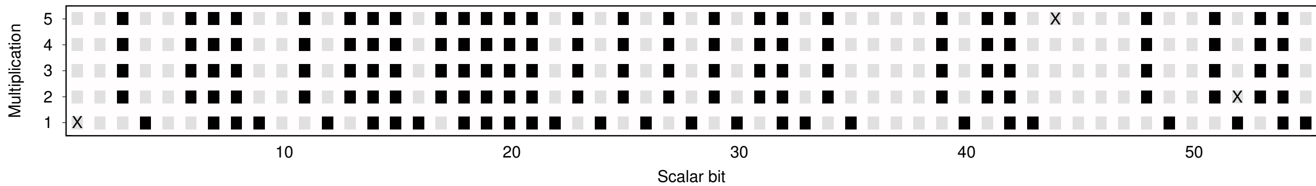

Figure 3: Five processed traces. Dark spots indicate an observed long reduction and light spots indicate an observed short reduction. Three errors in the observation are marked with X marks. Two of them observe the wrong reduction length and the third is a superfluous bit.

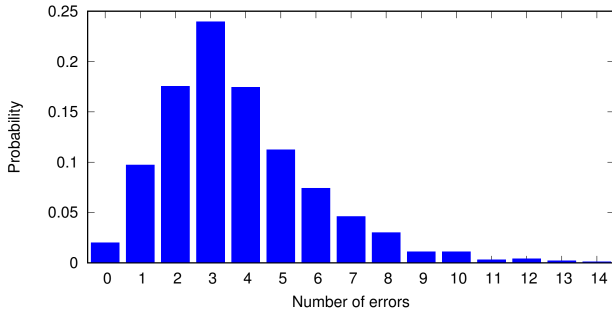

Figure 4: Distribution of the number of errors (excluding four first bits) in traces of the scalar multiplication.

Attack Scenario. We use the default install of git-crypt on Ubuntu 17.04. To attack, we modify the victim's encrypted key file by replacing the ECDH ephemeral public key with the element of order 4 and commit the change into the repository. Once the victim pulls the modified key file, any attempt to encrypt or decrypt files in the repository will send an element of order 4 into Libgcrypt's scalar multiplication routine, allowing the attacker to collect side channel information.

Attack Results. Running the attack on real-world software rather than on the scalar multiplication code only, presents two problems. The first is that GnuPG performs several public key operations when trying to match the public key used for encrypting the key file with the victim's key storage (called *keyring* in the GnuPG nomenclature). These operations access both the constant-time swap code and the multiplication code which our attack monitors. Consequently, the side channel attack collects much more information and we need to distinguish between the ECDH scalar multiplication operation and the other operations. To achieve that, we also use `FR-trace` to monitor the entry to the ECDH decryption code and ignore all accesses to monitored code that precede the entry.

The second problem we witness is that when running more software the system is more noisy, increasing the error rate. On average, we find that we have 14.9 errors in a trace and therefore we require 11 traces to recover the secret key.

4.3.2 Pidgin

Pidgin is a popular open-source chat application that supports communication across a variety of chat networks [25]. We target Pidgin's OpenPGP plugin [39], which allows a sender to encrypt messages with the recipient's public GnuPG key. When the recipient has the plugin enabled and receives a PGP-encrypted message, the message is automatically decrypted using GnuPG with no action required by the recipient.

Attack Scenario. We use the default APT distribution of Pidgin and the OpenPGP plugin for Ubuntu 17.04. To carry out the attack, we first enable PGP for the chat session and then send a chat message, replacing the ECDH ephemeral public key with an element of order 4. When the victim receives the message, Pidgin uses GnuPG to decrypt the ciphertext, calling the scalar multiplication function in Libgcrypt with the order-4 element and enabling the side-channel attack.

Attack Results. We sent 100 malicious Pidgin messages containing an order-4 element to the target machine, while monitoring its cache activity. This resulted in 100 traces containing an average of 7.6 errors with 3 of the traces containing unusable data. Overall we recovered the victim key using information from 7 traces.

4.3.3 Enigmail

Enigmail is an add-on for the Mozilla Thunderbird email client that enables the sender to encrypt emails using the recipient's public GnuPG key. When the recipient views a GnuPG-encrypted email, Enigmail passes the ciphertext to GnuPG to be decrypted.

Attack Scenario. For our attack, we assume that the victim is running Mozilla Thunderbird in Ubuntu 17.04 with the default version of Enigmail installed. The attacker sends a GnuPG-encrypted email with the ECDH public key replaced with an order-4 element. When the victim clicks on the encrypted email, Enigmail passes the ciphertext to GnuPG for decryption, enabling a side-channel attack similar to the above.

Attack Results. Similar to the Pidgin attack above, we used Enigmail to decrypt 100 encrypted email messages containing order-4 elements on the target machine while monitoring its cache activity. This resulted in 100 traces containing an average of 9.1 errors with 9 of the traces containing unusable data. Overall we recovered the victim key using information from 7 traces.

5 SOFTWARE COUNTERMEASURES

Our attack works by passing specially chosen ciphertexts (order-4 curve points) to the ECDH decryption routine to be multiplied by the secret scalar. Due to the mathematical structure of these inputs and the Montgomery ladder algorithm, they trigger key-dependent leakage patterns deep inside Libgcrypt's basic finite field arithmetic operations. Observing these patterns using the cache side

channel, we are able to recover the secret key. We now briefly review common countermeasures for preventing such chosen ciphertext attacks. See Fan et al. [30] and Fan and Verbauwhede [31] for more extended discussions.

Constant Time Arithmetic. Both the original publication of Curve25519 [14] and the NaCl library [16] use constant-time field arithmetic. Replacing Libgcrypt's code with any of these implementations would prevent our attack as well as any known microarchitectural side-channel attack. We repeat here the recommendation stated in RFC 7748 [53] as our attack uses a similar type of leakage from Libgcrypt's arithmetic library in order to achieve key extraction: "it is important that the arithmetic used not leak information about the integers modulo p, for example by having $b \cdot c$ be distinguishable from $c \cdot c$."

Rejecting Known Bad Points. To protect against small subgroup attacks against Curve25519 and related curves that have a small set of low-order elements, an implementation can simply check if the received public key is in the set. Bernstein [12] provides a full list of these points for Curve25519, but suggests that rejecting these points is only necessary for protocols that wish to ensure "contributory" behavior. Langley and Hamburg [53] have a similar suggestion. We argue that rejecting these points would also give better side-channel protection. While this protection may seem unnecessary when used with constant-time code, as Kaufmann et al. [50] demonstrate, constant-time code is fragile and may fail to provide adequate protection.

Point Blinding. To protect the scalar k that is multiplied by a potentially-malicious ciphertext P, one can generate a random point R, compute $[k](\mathsf{P} + \mathsf{R})$, and then subtract $[k](\mathsf{R})$ from the result [26]. This countermeasure completely protects against the chosen ciphertext attack we describe in this paper, since the attacker can no longer choose the point P to be multiplied with k. However, this countermeasure introduces an extra scalar-by-point multiplication for each decryption, so the negative performance effect of this countermeasure is significant.

Scalar Randomization. Many side-channel attacks rely on combining the leakage over several decryption operations in order to extract the key. A possible countermeasure to prevent such averaging is scalar randomization [26], which adds a random multiple of the group order to the scalar k before performing the scalar-by-point multiplication operation. This changes the sequence of elliptic curve operations performed for every decryption operation, hindering the averaging operation. A similar countermeasure splits the scalar k into n parts $k_1, \ldots k_n$ such that $k = \sum_{i=1}^{n} k_i$, performs the scalar-by-point multiplication operation separately on each k_i, and then combines the result [24]. This countermeasure is cheaper than point blinding, but not as effective.

According to Bernstein [14], the order of the base point of Curve-25519 is

$$2^{252} + 27742317777372353535851937790883648493.$$

We note that this number has a sequence of 128 consecutive zero bits. Ciet and Joye [24] note that scalar randomization with multipliers of this form still reveals a large number of bits. Thus, we do not recommend using this countermeasure.

Defense in Depth. The cache attack described in this paper will not work against an implementation that has truly constant-time code, since the attack relies on subtle timing differences deep within arithmetic functions. However, writing constant-time code is a non-trivial task; even the side-channel resistant Montgomery ladder algorithm still leaves room for error, as this paper demonstrates. Rather than providing the bare minimums for security, we argue that systems should be designed to have defense in depth, so that a single mistake on the part of the developer does not have disastrous consequences for security.

With regard to the attack described in this paper, the lack of input validation caused sensitive secret-key operations to be performed on adversarial inputs, which allowed us to transform an existing side-channel weakness into a full key-recovery attack. Thus, we recommend that in addition to writing side-channel resistant code, developers should also deploy the aforementioned countermeasures. This would have the effect of reducing the capability of an attacker to mount key-extraction attacks by exploiting side-channel weaknesses.

6 CONCLUSIONS

In this work, we demonstrate a side-channel attack against Libgcrypt's implementation of ECDH encryption with Curve25519, which uses the Montgomery ladder and branchless formulas for point addition and doubling. Instead of relying on easily observable behavior such as high-level key-dependent branches or memory accesses, our attack exploits a low-level side channel vulnerability deep inside Libgcrypt's basic finite field arithmetic operations. We find that by passing order-4 elements into the decryption routine, we can trigger specific key-dependent code execution paths that a cache side channel attack is able to detect. From these key-dependencies, we are able to recover the key within about a second of measurements.

Chosen Ciphertext as Order-8 Element. While we did not investigate passing in order-8 elements as inputs to the decryption routine, these points would also introduce mathematical structure into the operands of the elliptic curve operations in the scalar-by-point multiplication. We expect that a similar attack would at least achieve partial key recovery.

Future Work. Our attack uses multiple decryption traces and averages the results to reduce the error rate. Overcoming side-channel noise to enable an attack with only a single trace is an open problem. Our attack relies on the special mathematical properties of the representation of the elements of order 4. Rejecting these points is an effective countermeasure to our attack; however, it does not address the underlying problem of having vulnerable arithmetic operations. It may be possible to extend our work to attack the arithmetic operations without using a low-order group element. Finally, our techniques should also be applicable for mounting low-bandwidth key extraction attacks against Libgcrypt's implementation of Curve25519 using physical side channels. Mounting such attacks remains an open problem.

ACKNOWLEDGMENTS

We thank Eric Wustrow for pointing out the existence of low-order elements in Curve25519.

Luke Valenta was supported by an internship at Cisco during part of the paper revision process.

Yuval Yarom performed part of this work as a visiting scholar at the University of Pennsylvania.

This work was supported by the an Endeavour Research Fellowship from the Australian Department of Education and Training; by National Science Foundation under Grant No. CNS-1408734; by the 2016/2017 Rothschild Postdoctoral Fellowship; by the Warren Center for Network and Data Sciences; by the financial assistance award 70NANB15H328 from the U.S. Department of Commerce, National Institute of Standards and Technology; by the Defense Advanced Research Project Agency (DARPA) under Contract #FA8650-16-C-7622 and by a gift from Cisco.

REFERENCES

[1] git-crypt — Transpartent File Encryption in git. https://www.agwa.name/projects/git-crypt/.

[2] GNU Privacy Guard. https://www.gnupg.org

[3] GnuPG Frontends. https://www.gnupg.org/related_software/frontends.html

[4] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. 2010. New Results on Instruction Cache Attacks. In *CHES*. 110–124.

[5] Toru Akishita and Tsuyoshi Takagi. 2003. Zero-Value Point Attacks on Elliptic Curve Cryptosystem. In *ISC 2003*. 218–233.

[6] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop van de Pol, and Yuval Yarom. 2016. Amplifying Side Channels Through Performance Degradation. In *ACSAC*. Los Angeles, CA, US.

[7] Jean-Philippe Aumasson. 2017. Should Curve25519 keys be validated? (April 2017). https://research.kudelskisecurity.com/2017/04/25/should-ecdh-keys-be-validated/

[8] Elaine Barker, Lily Chen, Allen Roginsky, and Miles Smid. 2013. NIST SP 800-56A: Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography (Revision 2). (2013).

[9] Pierre Belgarric, Pierre-Alain Fouque, Gilles Macario-Rat, and Mehdi Tibouchi. 2016. Side-Channel Analysis of Weierstrass and Koblitz Curve ECDSA on Android Smartphones. In *CT-RSA 2016*. Springer, 236–252.

[10] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. 2014. "Ooh Aah... Just a Little Bit" : A Small Amount of Side Channel Can Go a Long Way. In *CHES 2014*. 75–92.

[11] Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. https://cr.yp.to/ecdh.html

[12] Daniel J. Bernstein. A state-of-the-art Diffie-Hellman function. https://cr.yp.to/ecdh.html.

[13] Daniel J. Bernstein. 2005. Cache-timing attacks on AES. (2005). http://cr.yp.to/papers.html#cachetiming.

[14] Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *PKC*. New-York, NY, US, 207–228.

[15] Daniel J Bernstein, Chitchanok Chuengsatiansup, and Tanja Lange. 2014. Curve41417: Karatsuba revisited. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 316–334.

[16] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. 2012. The Security Impact of a New Cryptographic Library. In *LatinCrypt'12*. Santiago, CL, 159–176.

[17] Ingrid Biehl, Bernd Meyer, and Volker Müller. 2000. Differential fault attacks on elliptic curve cryptosystems. In *Annual International Cryptology Conference*. Springer, 131–146.

[18] Olivier Billet and Marc Joye. 2003. The Jacobi Model of an Elliptic Curve and Side-Channel Analysis. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes (AAECC) 2015*. Springer, 34–42.

[19] Billy Bob Brumley and Risto M. Hakala. 2009. Cache-Timing Template Attacks. In *ASIACRYPT 2009 (Lecture Notes in Computer Science)*, Vol. 5912. Springer, 667–684.

[20] Billy Bob Brumley and Nicola Tuveri. 2011. Remote Timing Attacks Are Still Practical. In *ESORICS 2011*. Springer, 355–371.

[21] David Brumley and Dan Boneh. 2005. Remote timing attacks are practical. *Computer Networks* 48, 5 (2005), 701–716.

[22] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. 2007. OpenPGP Message Format. RFC 4880. (Nov. 2007).

[23] David V Chudnovsky and Gregory V Chudnovsky. 1986. Sequences of numbers generated by addition in formal groups and new primality and factorization tests. *Advances in Applied Mathematics* 7, 4 (1986), 385–434.

[24] Mathieu Ciet and Marc Joye. 2003. (Virtually) Free Randomization Techniques for Elliptic Curve Cryptography. In *ICICS 2003*. Springer, 348–359.

[25] Pidgin Community. Pidgin. https://www.pidgin.im/

[26] Jean-Sébastien Coron. 1999. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In *CHES 1999*. 292–302.

[27] Thai Duong. 2015. Why not validate Curve25519 public keys could be harmful. (Sept. 2015). https://vnhacker.blogspot.ch/2015/09/why-not-validating-curve25519-public.html

[28] M. Elkins, D. Del Torto, R. Levien, and T. Roessler. 2001. MIME Security with OpenPGP. RFC 3156. (2001). http://www.ietf.org/rfc/rfc3156.txt

[29] Junfeng Fan, Benedikt Gierlichs, and Frederik Vercauteren. 2011. To Infinity and Beyond: Combined Attack on ECC Using Points of Low Order. In *Cryptographic Hardware and Embedded Systems CHES 2011*. Springer, 143–159.

[30] Junfeng Fan, Xu Guo, Elke De Mulder, Patrick Schaumont, Bart Preneel, and Ingrid Verbauwhede. 2010. State-of-the-art of Secure ECC Implementations: A Survey on Known Side-channel Attacks and Countermeasures. In *HOST 2010*. 76–87.

[31] Junfeng Fan and Ingrid Verbauwhede. 2012. An Updated Survey on Secure ECC Implementations: Attacks, Countermeasures and Cost. In *Cryptography and Security: From Theory to Applications - Essays Dedicated to Jean-Jacques Quisquater on the Occasion of His 65th Birthday*. 265–282.

[32] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2016. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering* - (2016).

[33] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Adi Shamir, and Eran Tromer. 2016. Physical key extraction attacks on PCs. *Commun. ACM* 59, 6 (2016), 70–79.

[34] 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*. 207–228. Extended version: Cryptology ePrint Archive, Report 2015/170.

[35] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. 2016. ECDH Key-Extraction via Low-Bandwidth Electromagnetic Attacks on PCs. In *CT-RSA 2016*. 219–235.

[36] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. 2016. ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels. In *ACM Conference on Computer and Communications Security CCS 2016*. 1626–1638.

[37] Daniel Genkin, Itamar Pipman, and Eran Tromer. 2014. Get Your Hands Off My Laptop: Physical Side-Channel Key-Extraction Attacks on PCs. In *CHES 2014*. Springer, 242–260. Extended version: Cryptology ePrint Archive, Report 2014/626.

[38] Daniel Genkin, Adi Shamir, and Eran Tromer. 2014. RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. In *CRYPTO 2014*. Springer, 444–461 (vol. 1).

[39] T. Glaser. OpenPGP plugin for Pidgin. https://packages.debian.org/sid/pidgin-openpgp

[40] Louis Goubin. 2003. A Refined Power-Analysis Attack on Elliptic Curve Cryptosystems. In *PKC 2003*. 199–210.

[41] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. 2016. Flush, Gauss, and Reload — A Cache Attack on the BLISS Lattice-Based Signature Scheme. In *CHES*. 323–345.

[42] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security*. Washington, DC, US, 897–912.

[43] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE S&P*. Oakland, CA, US, 490–505.

[44] Mike Hamburg. 2015. Ed448-Goldilocks, a new elliptic curve. *IACR Cryptology ePrint Archive* 2015 (2015), 625.

[45] IANIX. 2017. Things that use Curve25519. https://ianix.com/pub/curve25519-deployment.html. (March 2017).

[46] Mehmet Sinan İnci, Berk Gülmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *CHES*. 368–388.

[47] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2014. Wait a minute! A fast, Cross-VM attack on AES. In *RAID*. Gothenburg, Sweden, 299–319.

[48] A. Jivsov. 2012. Elliptic Curve Cryptography (ECC) in OpenPGP. RFC 6637. (June 2012).

[49] Marc Joye and Sung-Ming Yen. 2002. The Montgomery Powering Ladder. In *Cryptographic Hardware and Embedded Systems (CHES) 2002*. Springer, 291–302.

[50] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. 2016. When Constant-Time Source Yields Variable-Time Binary: Exploiting Curve25519-donna Built with MSVC 2015. In *CANS'16*. 573–582.

[51] Cameron F Kerry and Charles Romine Director. 2013. NIST SP 186-4: Digital Signature Standard (DSS). (2013).

[52] Donald E. Knuth. 1981. The Art of Computer Programming. *Seminumeral Algorithms* 2 (1981), 257–258.

[53] Adam Langley and Mike Hamburg. 2016. Elliptic Curves for Security. RFC 7748. (Jan. 2016).

[54] Chae Lim and Pil Lee. 1997. A key recovery attack on discrete log-based schemes using a prime order subgroup. *Advances in CryptologyâĂŤCRYPTO'97* (1997), 249–263.

[55] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy 2015*. IEEE.

[56] Johannes Merkle and Manfred Lochter. 2010. Elliptic curve cryptography (ECC) Brainpool Standard Curves and Curve Generation. RFC 5639. (March 2010).

[57] Peter L. Montgomery. 1987. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Math. Comp.* 48, 177 (Jan. 1987), 243–264.

[58] Phong Q Nguyen. 2004. Can we trust cryptographic software? Cryptographic flaws in GNU Privacy Guard v1. 2.3. In *EUROCRYPT*, Vol. 4. Springer, 555–570.

[59] Katsuyuki Okeya, Hiroyuki Kurumatani, and Kouichi Sakurai. 2000. Elliptic Curves with the Montgomery-Form and Their Cryptographic Applications. In *Public Key Cryptography (PKC) 2000*. Springer, 238–257.

[60] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA 2006*. Springer, 1–20.

[61] Colin Percival. 2005. Cache missing for fun and profit. (2005). Presented at BSDCan. http://www.daemonology.net/hyperthreading-considered-harmful.

[62] César Pereida García and Billy Bob Brumley. 2017. Constant-Time Callees with Variable-Time Callers. In *USENIX Security Symposium 2017*. 83–98.

[63] César Pereida García, Billy Bob Brumley, and Yuval Yarom. 2016. "Make Sure DSA Signing Exponentiations Really are Constant-Time". In *CCS'16*. 1639–1650.

[64] Trevor Perrin. 2017. X25519 and zero outputs. (May 2017). https://moderncrypto.org/mail-archive/curves/2017/000896.html

[65] Joop van de Pol, Nigel P. Smart, and Yuval Yarom. 2015. Just a Little Bit More. In *CT-RSA 2015*. 3–21.

[66] The Enigmail Project. Enigmail: A simple interface for OpenPGP email security. https://www.enigmail.net

[67] Robert Ransom. 2014. Leading zero bits in the Montgomery ladder. IETF mailing list. (July 2014). https://www.ietf.org/mail-archive/web/cfrg/current/msg04749.html.

[68] Yukiyasu Tsunoo. 2002. Cryptanalysis of block ciphers implemented on computers with cache. *preproceedings of ISITA 2002* (2002).

[69] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. 2003. Cryptanalysis of DES implemented on computers with cache. In *CHES*, Vol. 2779. Springer, 62–76.

[70] Yuval Yarom. 2016. Mastik: A Micro-Architectural Side-Channel Toolkit. http://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf. (Sept. 2016).

[71] Yuval Yarom and Naomi Benger. 2014. Recovering OpenSSL ECDSA Nonces Using the Flush+Reload Cache Side-channel Attack. IACR Cryptology ePrint Archive, Report 2014/140. (Feb. 2014).

[72] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium 2014*. USENIX, 719–732.

[73] Sung-Ming Yen, Wei-Chih Lien, Sang-Jae Moon, and JaeCheol Ha. 2005. Power Analysis by Exploiting Chosen Message and Internal Collisions — Vulnerability of Checking Mechanism for RSA-Decryption. In *Mycrypt 2005*. Springer, 183–195.

[74] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. 2016. Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices. In *CCS'16*. Vienna, AT, 858–870.

[75] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM side channels and their use to extract private keys. In *CCS'12*. Raleigh, NC, US, 305–316.

[76] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *CCS'14*. Scottsdale, AZ, US.