PARASITE: PAssword Recovery Attack against Srp Implementations in ThE wild

Daniel De Almeida Braga
daniel.de-almeida-braga@irisa.fr
Univ Rennes, CNRS, IRISA
Rennes, France

Pierre-Alain Fouque
pierre-alain.fouque@irisa.fr
Univ Rennes, CNRS, IRISA
Rennes, France

Mohamed Sabt
mohamed.sabt@irisa.fr
Univ Rennes, CNRS, IRISA
Rennes, France

ABSTRACT

Protocols for password-based authenticated key exchange (PAKE) allow two users sharing only a short, low-entropy password to establish a secure session with a cryptographically strong key. The challenge in designing such protocols is that they must resist offline dictionary attacks in which an attacker exhaustively enumerates the dictionary of likely passwords in an attempt to match the used password. In this paper, we study the resilience of one particular PAKE against these attacks. Indeed, we focus on the Secure Remote Password (SRP) protocol that was designed by T. Wu in 1998. Despite its lack of formal security proof, SRP has become a de-facto standard. For more than 20 years, many projects have turned towards SRP for their authentication solution, thanks to the availability of open-source implementations with no restrictive licenses. Of particular interest, we mention the Stanford reference implementation (in C and Java) and the OpenSSL one (in C).

In this paper, we analyze the security of the SRP implementation inside the OpenSSL library. In particular, we identify that this implementation is vulnerable to offline dictionary attacks. Indeed, we exploit a call for a function computing modular exponentiation of big numbers in OpenSSL. In the SRP protocol, this function leads to the call of a non-constant time function, thereby leaking some information about the used password when leveraging cache-based Flush+Reload timing attack. Then, we show that our attack is practical, since it only requires one single trace, despite the noise of cache measurements. In addition, the attack is quite efficient as the reduction of some common dictionaries is very fast using modern resources at negligible cost. We also prove that the scope of our vulnerability is not only limited to OpenSSL, since many other projects, including Stanford's, ProtonMail and Apple Homekit, rely on OpenSSL, which makes them vulnerable. We find that our flaw might also impact projects written in Python, Erlang, JavaScript and Ruby, as long as they load the OpenSSL dynamic library for their big number operations. We disclosed our attack to OpenSSL who acknowledged the attack and timely fixed the vulnerability.

CCS CONCEPTS

• **Security and privacy → Security protocols**.

KEYWORDS

SRP, PAKE, Flush+Reload, PDA, OpenSSL

1 INTRODUCTION

1.1 Context and Motivation

Passwords are by far the most popular method of end-user authentication on the web. Password breaches, due for instance to sophisticated phishing attacks, seriously multiply the risk of authentication compromise. Aware of such issues, a growing number of service providers couple passwords with another authentication factor: this is known as two-factor authentication (2FA). However, a longitudinal analysis in [15] highlights that the adoption rate of 2FA has been mostly stagnant over the last years. This is partly due to the fact that 2FA requires unwilling users to modify their way to interact with the web. An alternative approach is to keep relying solely on passwords, but to change how they are verified in a way that protects against phishing attacks as well as server breaches and dictionary attacks. The advantage of such an approach is that it does not change users experience. However, this does not come without cost; the burden is now shifted to service providers that should support new cryptographic protocols. These protocols are called password-authenticated key exchange (PAKE).

In its simplest form, a PAKE protocol allows two parties sharing nothing but a low-entropy string (i.e., password) to mutually authenticate each other and to establish a secure session. Thus, PAKE protocols remove the need of public-key infrastructure (PKI). In addition, passwords, or even their salted hash values, are not stored on servers. This implies that an attacker who breaks into a server no longer has any direct access to passwords. Instead, they still require to perform a per-user dictionary attack to retrieve the password, which makes PAKEs appealing for numerous industrial solutions. An eminent example here is the recently released Wi-Fi Protected Access 3 (WPA 3) standard that relies on PAKE to improve security upon WPA 2. There are also plentiful of RFC publications [28, 41, 43] standardizing PAKE and their usage.

The concept of PAKE is not new; it was initially described by Bellovin and Merritt in 1992 [10] and has received considerable attention since then. Nevertheless, they still suffer from slow adoption [18]. Indeed, the early protocols were lacking important security properties, and the PAKE world was patent-encumbered for some period of time [10, 30]. This resulted in protocols, such as SRP [48] and J-PAKE [27], that included additional complexity solely for patent circumvention, thereby making security proofs more complex or impractical. The availability of efficient implementations was also negatively impacted by this complexity.

However, interest has been renewed in PAKE recently. The CFRG (Crypto Forum Research Group) started a competition in 2019 with the goal to recommended a small set of PAKE protocols for the IETF community [16]. It is still early to measure the positive impact of this competition on PAKE adoption. However, it allowed the community to carefully review the various existing PAKEs with respect to some specific criteria [41]. Unfortunately, most of the reviews are based on the abstract design of the proposed candidates, and not on their reference or deployed implementations. This is regrettable for two reasons. First, the RFC 8125 [41] explicitly requires (refer to section 4.1) that any PAKE shall protect against timing and side-channel attacks. Second, it is well established now that breaking cryptography is not merely a matter of theoretically breaking a cryptographic algorithm. Implementation pitfalls matter in order to guarantee that the running code does not leak any secret information. Our paper aims at bridging this gap by scrutinizing the different implementations in the wild of one particular PAKE, namely the Secure Remote Password protocol (SRP).

1.2 Our Contribution

The Secure Remote Password protocol (SRP 6a, SRP) was introduced by Wu [48]. Since SRP first iteration back in 1998, new versions have been defined to address vulnerabilities and support stronger algorithms. The Stanford SRP homepage lists four different versions of SRP, with the last one being SRP 6a. The SRP version 3 is specified in RFC 2945 [46], and the version 6 is described in RFC 5054 [43].

The case of SRP is quite peculiar. On the one hand, it has been highly disapproved by some of the crypto community [25]. It was not even considered by the CFRG PAKE competition, mainly because of its lack of formal security proof. On the other hand, SRP is arguably the most widely used protocol of its type. It draws its popularity from its patent-free definition, quite straightforward design, and the availability of efficient open-source implementations with no restrictive licenses in many programming languages, including C#, .NET, Java, C/C++, Python, Go, JavaScript, Rust, Erlang, and Ruby.

SRP is not only included in some amateur code on Github. It is actually used as security branding by various solutions. These include: ING bank for their mobile app [29], the password manager 1Password [21], AWS (Amazon Web Service in its cognito authentication extension) [6], Apple Homekit when pairing with the IoT devices [5], the secure email system of ProtonMail [40], the GoTo tools (GoToMeeting, GoToTraining, GoToWebinar) [34], etc.

Therefore, it becomes quite timely to look deeper into the different SRP implementations, as any identified vulnerability might lead to serious consequences. Despite being widely deployed, these implementations do not seem to have received any review for a long time. Our goal is to consider a modern threats model, especially cache-based attacks, to analyze SRP. Nevertheless, it is quite challenging to exploit all the existing implementations for two main reasons. First, designing exploits that work for different programming languages is not straightforward. Second, SRP is used in diverse contexts; from mobile banking app to video conferencing systems. Thus, any identified vulnerability would be hard to generalize.

In this paper, we adopt another approach, since we notice that most SRP implementations are rather similar. Indeed, we mainly focus on one reference implementation: OpenSSL's (in C). We show that the core part of the OpenSSL SRP has inspired 18 other projects in 5 programming languages (refer to Table 2 for the complete list). Moreover, these projects dynamically load the OpenSSL library in order to call its big integers API. We underline that this implies that any vulnerability that can be identified in the OpenSSL code may constitute a generic attack vector for all these implementations, regardless of their programming language.

The SRP property that we attempt to attack is its resistance against offline dictionary attack. In theory, an attacker eavesdropping all messages within the authentication process can deduce no information about the shared password. Our contribution is to show that numerous SRP implementations do not satisfy such an important property. Indeed, we first identify some leakage vector in the client side of the OpenSSL SRP. Then, we exploit this leakage through micro-architectural cache-based attack. We improve the effectiveness of our attack in a real-world scenario, since it only requires a single measurement to guess the secret password with high probability. We simplify the execution of our exploit by automating the whole process, including spying on executed instructions, interpreting the noisy cache measurements, and performing the offline dictionary attack. Finally, we show the large scope of our attack that goes beyond OpenSSL, and concerns widely deployed solutions, such as Proton Mail and Apple Homekit.

In summary, this paper makes the following contributions:

- we perform a thorough analysis of the OpenSSL implementation of SRP, we report undocumented details about their implementations, and we identify an execution flow in the client side that depends on some sensitive data;
- we show how this non-constant time implementation can leak some information about the secret password, we exploit our findings through cache-based techniques using Flush+Reload and a Performance Degradation Attack (PDA);
- we perform an offline dictionary attack against SRP, study the theoretical amount of information that might be leaked during an authentication exchange and experimentally validate our analysis on OpenSSL 1.1.1h;
- we identify 18 projects and industrial solutions that depend on the flawed implementation of OpenSSL and report on unpublished attacks and weaknesses in the analyzed implementations. For each project, we point out the vulnerable component and the overall security implication of the attack;
- we implement a Proof of Concept (PoC) of our vulnerability on various impacted projects and make it open-source [1];
- we propose a (straightforward) mitigation of the attack and empirically evaluate its overhead. We notified the concerned projects following our policy of responsible disclosure, and provided support in the patch integration. Two CVE IDs have been reserved for us by GoToMeeting and Digi Xbee.

1.3 Attack Scenario

To illustrate our vulnerability, we propose the following attack scenario. Note that this only serves as an example, and it does not limit the scope of our work. We suppose a classical SRP use case, where a client tries to authenticate themselves to a server using a password. The goal is to recover the client's password.

To this end, the attacker exploits a non-constant time execution during an insecure modular exponentiation. Here, the attacker will leverage some particular properties of the victim's micro-architecture to monitor the execution, and extract information about some sensitive data. The attacker is then able to use this leaked information to perform an offline dictionary attack in order to recover the shared password with high efficiency.

[1] https://gitlab.inria.fr/ddealmei/poc-openssl-srp

Our experiments show that a single trace is mostly enough to recover the password from any practical dictionary size. Once the attacker recovers the password, they can impersonate both server and client in all future sessions.

2 BACKGROUND

In this section, we provide a brief description of the SRP protocol, as well as its current limitations. We also introduce the cache attacks that we will leverage to exploit our identified vulnerability.

2.1 Password-Authenticated Key Exchange

A PAKE protocol is a cryptographic protocol that allows two parties sharing a secret password to mutually authenticate each other without explicitly revealing the password in the process.

The challenge of PAKE schemes is the low entropy of passwords that makes them weak against dictionary attacks. In particular, there are two types of attacks: offline and online. In an offline attack, the adversary observes a protocol run, and then goes offline to do password testing on their own. In an online setting, the attacker requires to run the protocol with a password guess as input, and observes whether the protocol run succeeds or fails. It is clear that such an attack is unavoidable. However, online attacks could be mitigated by rate-limiting countermeasures, e.g., by applying wait times after failed login attempts. This countermeasure could not be applied with offline attacks. Therefore, a PAKE protocol must be designed to be resilient against offline dictionary attacks. More discussions may be found in [30, 48]. Our attack breaks this property for several SRP implementations by recovering some leaked information about the password, making offline attacks possible.

PAKE protocols essentially come in two variants. Firstly, so-called balanced or symmetric PAKE protocols, introduced in [10] and formalized in [9], require both parties to know the shared password in plaintext. This implies that a break into the server leaks the password for all users. Examples include Dragonfly [28] and CPace [26]. Secondly, so-called augmented or asymmetric PAKE protocols, for instance SRP [48] and OPAQUE [31], are introduced in [11] and later formalized in [13, 24]. These protocols are designed such that the server is given access only to a password-verifier, and the password itself is only available to the client. Thus, upon a server compromission, an attacker is still forced to perform an exhaustive offline dictionary attack. Such an additional protection makes the task significantly more difficult for the attacker. Thus, augmented PAKE protocols are generally preferred.

2.2 Secure Remote Password (SRP) Protocol

The SRP protocol is an augmented PAKE protocol introduced in 1998 [48] to bypass existing patents. Some argue that SRP is the most widely used and standardized protocol of its type, despite its lack of formal proof. SRP was standardized for the first time in 2000 in RFC 2944 [47] – Telnet Authentication: SRP. Nowadays, it is not part of the recommended protocols by the CFRG PAKE selection competition [16] in 2020.

Below, we describe the two phases of SRP: registration and login. Before, we discuss the assumptions made for its design:

- a group G of prime order p, which is a safe prime (i.e. a number of the form $p = 2q + 1$, where q is also prime), and generator g are publicly known.
- the particular group may be negotiated before the exchange, or transmitted by the client.
- a public group-related parameter $k = H(g \mid\mid p)$ is computed by both entities upon the choice of the group, where H is any cryptographic hash function.
- the client knows a secret password.

Registration. In SRP, the server never gets the password in plaintext. Instead, it stores the password in the form $(client\ id, v, salt)$, where v is a password verifier and *salt* is a random string. Let x be $x = H(salt \mid\mid H(client\ id \mid\mid password))$. The verifier is generated as follows: $v = g^x \bmod p$. We notice that v is bounded to both the identifier and the password. The registration phase is assumed to be performed in a third-party secure channel, so an attacker cannot have access to this verifier. For all following login attempts, we assume this registration step has been performed successfully.

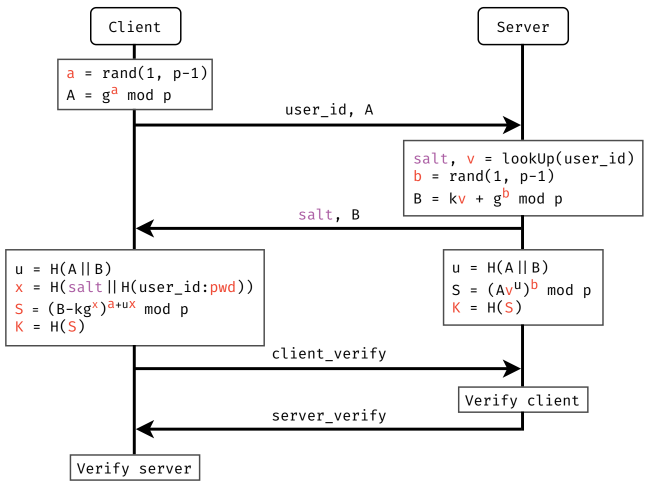

Figure 1: SRP login workflow. Group parameters g and p are negotiated before-hand, and $k = H(g||p)$ is computed after the negotiation. Secret values are displayed in red, while the salt, as a sensitive value, is displayed in purple.

Login. Before starting this phase, the client and the server agree upon the group parameters g and p, as well as their digest $k = H(g \mid\mid p)$. During login, first, the user generates a random number a uniformly chosen from $(1, p-1)$, computes $A = g^a \bmod p$ and sends it to the server along with their identifier. Upon reception, the server looks up to the user's password entry and fetches the corresponding password verifier and salt. Then, the server generates a random number b uniformly chosen from $(1, p-1)$, computes $B = kv + g^b \bmod p$, and sends both the salt and B to the user. Using the salt and the password, the client is able to recover the verifier $v = g^x \bmod p$. Now, both parties can compute the shared key K. Finally, they end up by exchanging confirmation messages to verify the good proceedings of the protocol. Figure 1 summarizes SRP workflow, with secret information in red (the salt is a sensitive information, but not confidential).

2.3 SRP Limitations

In this part, we outline some inherent flaws of SRP. These flaws are well known, but do not result in any severe vulnerability. We stress that none of these flaws is necessary for our attack; only the first one is used for optimization purposes.

2.3.1 Variation of SRP Settings. Since the first messages sent by the server are not authenticated, an attacker might modify them. This concerns both the salt and the group parameters (base and modulus). Of course, such a modification will end up in failure in the session establishment. However, it might be interesting to have several runnings of SRP for the same password, which might allow an attacker to increase the recovered leaked data.

2.3.2 Pre-computation Attacks. SRP as standardized does not protect from leak of server-side data. Namely, the attacker can precompute a table of values based on the salt and a passwords dictionary. This implies that as soon as the attacker succeeds in compromising a server, they can instantly find a user's password without any further effort. This is due to the fact that anyone, including the attacker, can ask for any user's salt. Such an attack is unfortunate, despite SRP being an augmented PAKE. To address the above threat, SRP would need a modern password-based key derivation function, such as Scrypt, so that pre-computations become expensive.

2.4 Cache Attacks

Micro-architectural attacks have long been used to gain information about sensitive data in cryptographic implementations. Below, we will only present the attacks that we use in our contribution.

2.4.1 Micro-architectural leaks. In modern CPUs, the time required to access a particular memory address may vary significantly depending on whether the data is already in cache (cache hit) or not (cache miss). CPUs access memory to fetch the program data, or the instructions to be executed. In both cases, if the access depends on a secret value, it might leak some information by using the time needed to load the required memory address. Indeed, the attacker can use a spy process interacting with the cache, in such a way that triggers these different time accesses. In most instruction-driven attacks, the spy process is used to probe specific part of the victim code in order to follow the execution of the process, and thereby deducing which part of the code was executed.

2.4.2 Flush+reload and Performance Degradation Attack. In 2014, Yarom and Falkner presented a ground breaking approach called Flush+Reload [50]. The idea is to directly monitor memory accesses in the inclusive L3 cache of Intel's CPU to accurately determine which memory addresses are accessed by the victim. To do so, a spy process targets some specific memory lines (corresponding to a sensitive instruction for instance) and repeatedly flush it out of the CPU cache and reload it. If the victim tried to access it in the meantime, the reload will be fast (cache hit). Otherwise, the reload would look for the memory line in upper memory level, making the operation significantly longer (cache miss).

In 2016, Allan et al. used a Performance Degradation Attack [3] that constantly flush some memory line out of the cache in order to make some operations longer to execute. Since then, the PDA is often paired with other attacks to achieve better results.

3 ATTACKING SRP IMPLEMENTATION OF OPENSSL

As mentioned previously, SRP has become widely deployed, and therefore numerous implementations exist in various programming languages. Here, we provide an in-depth look at the OpenSSL implementation of SRP, describe why it is vulnerable and explain how we can efficiently exploit it. In particular, OpenSSL provides support to TLS-SRP [43] through this implementation. Because of the popularity of OpenSSL, it is fair to assume that the codes have inspired other implementations, or simply being reused as is (see section 5). We study OpenSSL implementation up to version 1.1.1i (included), which is the last version at the time of writing.

3.1 Threat Model

We assume that the attacker be able to perform a Flush+Reload attack [50]. Since the operations required for this attack (cache access and eviction) do not rely on particular permissions, a common assumption is that the attacker deploys an unprivileged user-mode program on the targeted device. This spyware runs as a background task and records the CPU cache accesses to some specific functions.

One might argue that this is a strong threat model. However, we insist on the fact that any unprivileged code can play the role of our spyware, and therefore can be hidden into any third-party program running on the victim's computer. Moreover, previous papers in the literature suggest that such memory accesses can be remotely granted through a JavaScript code injection in web browser [38]. However, we did not investigate the effectiveness of our attack in such a context. It is worth noting that the vulnerability itself stems from a non-constant-time control flow. Sensitive information may leak through the overall execution time, hence may be recovered remotely. We did not explore this attack scenario neither, since it would require a significant amount of measurements to get reliable results (due to network noise and the low execution time difference).

To increase the amount of leaked data, the attacker may repeat their measurements on a modified version of the session establishment (by changing the salt as described in subsection 2.3). We stress that this assumption is only needed to increase the efficiency of our attack. To our surprise, it turns out that we seldom require this; both theoretical (subsection 3.5) and experimental (subsection 4.3) amount of information that is leaked from a *single* session establishment allows us to fully reduce a large password dictionary.

3.2 OpenSSL Implementation

In TLS-SRP, OpenSSL uses SHA-1 as a hash function H, and supports all standardized groups in [43]. The code concerning SRP can be found in crypto/srp/srp_lib.c and crypto/srp/srp_vfy.c, and the standardized groups are hardcoded as big numbers in crypto/bn/bn_srp.c. In particular, the client-side function that computes the shared key (and the verifier v), `SRP_Calc_client_key`, is illustrated in Listing 1. We notice that all big numbers involved in the verifier (line 3) have default (non-secure) flags set. In particular, the `BN_FLG_CONSTTIME` flag is not set.

The modular exponentiation function `BN_mod_exp`, defined in crypto/bn/bn_exp.c, only calls the constant-time exponentiation as a fallback, if a secure flag is set on one of the big numbers (base, exponent or modulus), or if the base is too large for the computation to

```
def SRP_Calc_client_key(N, g, B, x, a, u):
    # Compute the verifier $v = g^x \bmod N$
    v = BN_mod_exp(g, x, N)

    # Compute k = H(N || g)
    k = srp_Calc_k(N, g)

    # Compute the base $base = B - kv \bmod N$
    tmp = BN_mod_mul(v, k, N)
    base = BN_mod_sub(B, tmp, N)

    # Compute the exponent $exp = a + ux$
    tmp = BN_mul(u, x)
    exp = BN_add(a, tmp)

    # Compute the shared $S = base^{exp} \bmod N = (B - g^x)^{a+ux} \bmod N$
    S = BN_mod_exp(base, exp, N)

    return S
```

Listing 1: OpenSSL v1.1.1i implementation of the shared key computation by the client. A python-like syntax has been used to save up space, and add clarity. Variable names have been replaced to fit the protocol description in section 2.

be optimized (more than one processor word). Otherwise, the optimized modular exponentiation `BN_mod_exp_mont_word`, defined in the same file, is called. This is the case for the SRP implementation.

This fast exponentiation, summarized in Listing 2, uses a square-and-multiply (from MSB to LSB) approach, with Montgomery multiplication [35]. For maximum efficiency, the leading zero bytes are ignored, and the exponentiation only starts after the first MSB. Then, a word w (whose bit-size is defined by the processor architecture) is used as an accumulator to perform fast multiplications and squaring until its overflow (first loop from line 10 to 23). Once it overflows, a big number r (long-term accumulator) is initialized with the Montgomery representation of w. The big number r will, at the end, represent the result of the modular exponentiation. The word accumulator w is reset to either 1 (if it overflowed on a squaring) or g (if it overflowed on a multiplication). For the remaining iterations (line 32 to 51), the process is fairly similar: fast operations are performed on the accumulator, and every time it overflows, the big number is updated accordingly ($r = r \times w$). However, in this second part, the big number r is also squared at each iteration (line 42) to keep it updated, making the overall iteration longer.

3.3 Vulnerability Details

3.3.1 Targeting the modular exponentiation. Since no constant-time flag is set, and the base g is either 2, 5 or 19, the optimized exponentiation is performed during the verifier computation. In particular, the secret exponent $x = H(salt \mid\mid H(id \mid\mid pwd))$ is directly related to the password, which implies that recovering information on the exponent would provide the attacker enough information to perform an offline dictionary attack reducing the set of possible passwords. Information is leaked in several ways.

First of all, skipping the leading zero bits makes the overall execution time depends on $\log_2(x)$: the number of required iterations gives us the number of leading 0s in the exponent.

Moreover, we note that square-and-multiply might leak information as well, since the execution of each iteration depends on the value of the bit being processed. Unfortunately, performing these operations on processor words makes them very fast. Consequently,

```
1   def BN_mod_exp_mont_word(g, x, n):
2       nbits = BN_nb_bits(x)
3
4       # w is the word accumulator, used to make fast computations
5       # Set it to g since the exponentiation starts after the first bit to 1
6       w = g
7
8       # Do the optimized square and multiply on processor word only,
9       # until w overflows
10      for b in range(nbits-2, -1, -1):
11          # Square at every step
12          next_w = w*w
13          if (next_w / w) != w: # overflow
14              w = 1
15              break
16          w = next_w
17          # Multiply if bit is 1
18          if BN_is_bit_set(x, b):
19              next_w = w * g;
20              if (next_w / g) != w: # overflow
21                  w = g
22                  break
23              w = next_w
24
25      # Once the word accumulator overflowed, we need a big number to
26      # store its Montgomery representation as a "long term" accumulator.
27      r = BN_to_montgomery_word(w, n)
28
29      # Then get back to the square-and-multiply, with the additional
30      # constraint to update the long term accumulator r every time the
31      # word accumulator overflows.
32      for bit in range(b, -1, -1):
33          # Square at every step on the processor word
34          next_w = w * w;
35          if (next_w / w) != w:
36              # If it overflows, update r, and reset the word
37              r = BN_mod_mul_word(r, w, n)
38              next_w = 1
39          w = next_w;
40
41          # We update the Montgomery representation by squaring at each step
42          r = BN_mod_mul_montgomery(r, r, n)
43
44          # If the bit is set, we multiply by g
45          if BN_is_bit_set(x, bit):
46              next_w = w * g;
47              if (next_w / g) != w:
48                  # If it overflows, update r, and reset the word
49                  r = BN_mod_mul_word(r, w, n)
50                  next_w = g;
51              w = next_w;
52
53      # Finally, set r = r*w
54      if w != 1:
55          r = BN_mod_mul_word(r, w, n)
56
57      return BN_from_montgomery(r)
```

Listing 2: Modular exponentiation as performed by OpenSSL. Code has been reorganized to be more readable, and a python-like syntax has been adopted to save up space.

it is hard to distinguish them in practice, even by using high resolution cache attacks, such as Flush+Reload. Hence, recovering information during the first loop (line 10 to 23 of Listing 2) is hard.

However, starting from the first overflow on the accumulator w, the Montgomery representation is involved, making each iteration significantly longer, therefore more distinguishable. Hence, an attacker can exploit the Flush+Reload attack to monitor the memory line corresponding to the squaring of r at each iteration (call to `BN_mod_mul_montgomery` on line 42 of Listing 2). This function being only called once every iteration, knowing when it is called allows the attacker to distinguish iterations from one another.

In addition, if w overflows, an additional call to `BN_mod_mul_word` is performed (line 37 and 49 of Listing 2), making the particular iteration significantly longer. Therefore, the delay between two different iterations (or two calls to `BN_mod_mul_montgomery`) will be longer. This delay can be increased by performing a Performance Degradation Attack (PDA) on `BN_mod_mul_word` to make overflow iterations even longer to execute, hence easier to identify.

Consequently, an attacker is able to distinguish iterations, and to determine whether an overflow occurs. The principle of our attack is based on the fact that the number of iterations between two overflows is directly related to the binary sequence of the exponent being processed. Thus, the attacker can guess some bit patterns inside the exponent, which we detail below.

3.4 Exploiting the Leakage

The possible bit patterns we can observe between two overflows of the accumulator w depend on two parameters: the generator g, and the bit size of the processor word in a particular architecture. To demonstrate the attack, and without loss of generality, we fix the values for both parameters in the following sections. In order to meet the experimental setup we were using, we will consider w as a 64-bit word, which is the case for any x64 processor.

3.4.1 Choice of the Generator. The choice of the used generator is motivated by two major criteria. First, we consider the security/performance ratio offered by all standardized groups: groups 1024, 1536 are discouraged because of their low security. Similarly, group 2048 only provides enough security until 2030 [7]. Hence, higher order groups are more likely to be used. Finally, groups 6144 and 8192 may cause performance issues for some usage and only bring unnecessary security. Hence, a reasonable choice would be to use either group 3072 or 4096, both using a generator $g = 5$.

Next criterion is only valid from an attacker perspective: an attacker would prefer a generator allowing to recognize as many patterns as possible. Indeed, using $g = 19$ allows us to extract more information on average using the patterns alone. However, we show that $g = 5$ is the most interesting candidate when combining the pattern-related leak and the leak from the leading bits format. Additionally, it is more likely to be used in practice ($g = 19$ being reserved to group 8192). Henceforth, we fix the generator to $g = 5$ in the following sections. We stress that for other values of g, the same reasoning can be applied, but results in different possible patterns, and a lower average leak. For the sake of completeness, we provide a full study for all standard values of g in Appendix A.

3.4.2 Identifying bit patterns. Assume we start with a fresh accumulator $w = g$, meaning we either start the exponentiation, or we overflowed and processed a bit set to 1 (otherwise $w = 1$, and processing 0 bits would not modify the value w).

Now, we can generate all combinations leading to an overflow on w by applying the square and multiply on different bit sequences, and stop when we overflow. Thus, we identified two cases: (i) if the first two bits are 1, an overflow occurs after 3 iterations (for 111 and 110); (ii) otherwise, an overflow occurs after 4 iterations.

Let ℓ be the maximum bit length of the exponent, and let $T = \{v, V\}^{\ell}$ be the trace representing the list of iterations corresponding to the exponentiation. Here, v denotes a classic iteration (processing either a 0 or a 1), and V denotes an iteration where an overflow of w occurs, i.e., an iteration where line 37 or 49 of Listing 2 is executed.

An attacker is able to recover such a trace by using Flush+Reload. Let $b \in \{0, 1\}$ denote a bit of unknown value, and t be a chunk of the trace representing the operations between two overflows. We can convert our trace T into a set of possible values by extracting information on each chunk t as follow:

- A chunk $t = Vvv$, $t = Vv$ or $t = V$ is impossible, since at least 4 operations are needed to get the overflow.
- If $t = Vvvv$:
  - We know that $w = g$ at the first iteration following the reduction (otherwise an additional iteration is needed to overflow), hence the first bit processed is a 1: $t = 1bbb$.
  - An overflow in 3 iterations with $w = g$ is only possible if the next two bits are 11. We end up with $t = 111b$.
- If $t = Vvvvv$, we cannot recover the exact bit values, but we can reduce the set of possible values:
  - If $V = 1$, t will be in the set $\{110bb,\ 10bbb\}$. Otherwise, the accumulator would have overflowed one iteration before.
  - Otherwise, we meet the previous condition with a leading zero and get $t = 0111b$.

  We note $yyyy \in \{110b,\ 10bb,\ 0111\}$ the bits that take one of these seven possible patterns. Hence, $t = yyyyb$
- If $t = Vv...v$ with $len(t) = \lambda > 5$, the firsts $\lambda - 5$ bits of t (including the leading V) are 0. Hence $t = 00..0yyyyb$

On the last chunk of a trace, we will only be able to guess that leading bits are zeros if its length is greater than five, because we do not know when the accumulator would have overflowed.

3.4.3 Bits Pattern Parser. We designed an automated parser to interpret the information leaked by our spy process. Our parser produces guesses for as many bits values as possible.

```
Vvvvv Vvvv Vvvvvv Vvvvv Vvvvv Vvvvv Vvvv
yyyyb 111b 0yyyyb yyyyb yyyyb yyyyb bbbb
```

Listing 3: Example of trace interpretation. First line is the original trace, recovered using Flush+Reload, the second one is the interpretation.

Listing 3 presents an example of how we can interpret a trace to recover part of the bits. Strictly speaking, in our example, we are able to only recover 4 bits. However, each $yyyy$ can only take seven possible values (i.e., $110b$, $10bb$ and 0111), which greatly limits the possible values of the exponent. As mentioned previously, we cannot deduce information on the last chunk.

3.4.4 Acquiring more information. In case the information recovered from a single session establishment does not allow to prune enough passwords from the dictionary, the attacker can simply perform the attack on a second session, and change the salt transmitted by the server. Thus, the attacker will get different execution traces corresponding to the same password. Next, we discuss the amount of leaked data from a single session both theoretically (subsection 3.5) and experimentally (section 4).

3.5 Theoretical Amount of Leaked Information

The value of the exponent x is the output of a cryptographic hash function of bit-length ℓ. In this context, we can compute the average leakage based on the length ℓ and the probability to meet the different patterns in a uniformly distributed bit string (defined by the value g of the generator). All following statements hold for $g = 5$, and detailed computations for all standard values of g can be found in Appendix A.

Our spy process does not build a trace for the leading bits. The reason is twofold. First, the exponentiation starts at the MSB of the exponent. Second, the first iterations are fast and hardly observable, since their corresponding computations are performed without Montgomery representation. For $g = 5$, we observe on average 6.75 *leading* bits, and can recover $\mathcal{L}_{lead} = 4.5$ bits of information.

Next, on the remaining bits, we are able to identify patterns as described in subsubsection 3.4.2. Given a trace of k bits, we can estimate the average length $\mathbb{E}[N]$ of a pattern and the average information $\mathbb{E}[L]$ leaked by each pattern to get the information we can recover: $\mathcal{L}_{patterns}(k) = \mathbb{E}[L] \times k/\mathbb{E}[N]$.

By combining these two sources of leakage, we get the total leakage based on the bit-length ℓ of the exponent:

$$\mathcal{L}(\ell) = \mathcal{L}_{lead} + \mathcal{L}_{patterns}(\ell - 6.75) \approx 0.4 \cdot \ell + 2.06.$$

We stress that using a hash function with larger output would only increase the exponent size, thus leak more information.

3.6 Mounting an Offline Dictionary Attack

The above leakage allows an attacker to recover a significant part of x. Since the secret value is only defined by the password and some public values, they can compute various candidates for x and check whether the outcomes would match the recovered part.

If multiple passwords match the pattern, the attack is repeated by modifying the salt (which is sent unprotected by the server), and thus a new x corresponding to the same password is observed.

Since a cryptographic hash function is expected to behave like a random oracle, the digest corresponding to an invalid password will match the n bits of our trace with probability 2^{-n}. Therefore, an attacker recovering k bits of information on an exponent can expect to eliminate all candidates in a dictionary of 2^k entries.

Pre-computations are still possible. Indeed, hashing password candidates can be performed before the attack. However, this needs to be done for each client username and each salt. In order to reduce the effort of pre-computations, the attacker can force the use of a fixed salt, and only varies username. This pre-computation is pretty fast considering there is only two hashes to compute on small data. Specialized hardware instructions may be available on recent platforms, significantly speeding up the process.

3.7 Consequence of Password Recovery

As stated is in section 2, SRP authentication is based on the knowledge of the password (by the client) and the verifier (by the server). The vulnerability we described allows an attacker to recover enough information about a password-related value to perform a dictionary attack on any practical dictionary size, even with the use of a small output hash function such as SHA-1.

Assuming the attacker knows the password, they can: (i) impersonate the client; (ii) impersonate the server (since the attacker is able to compute the verifier from the password).

The required measurements being as fast as the session establishment, the overall complexity is dominated by the dictionary reduction. If an attacker has a large computational power, this attack can be performed on the fly. Hence, the attacker could gain a full MitM position and passively or actively process all data transmitted through the secure channel. The attack is still interesting even when it might take some time because of bigger dictionaries or more restricted computational power. Indeed, recovering the password after the sessions still allows the attacker to impersonate the client and/or gain active MitM position in all future sessions.

4 EXPERIMENTAL RESULTS

4.1 Experimental Setup

All experiments were performed on a Dell XPS13 7390 running on Ubuntu 20.04, kernel 5.4.0-58, with an Intel(R) Core(TM) i7-10510U and 16 GB of RAM. Binaries were compiled with gcc version 9.3.0-17ubuntu1 20.04 using their build script with the default configuration (optimization included).

We successfully reproduced our experiments in a Docker container running on the same hardware. This Docker is available in our github repository[2]. The container includes all elements needed to reproduce our experiments, from data acquisition with the victim program, to the dictionary attack recovering the password.

We used the FR-trace program of Mastik v0.02 [49] as a spy process performing both Flush+Reload and PDA. We ran all tests on OpenSSL version 1.1.1h, but a quick code analysis suggests that all versions of OpenSSL up to v1.1.1i included are also vulnerable.

4.2 Practical Attack on OpenSSL

4.2.1 Trace collection. To demonstrate our attack, we used a simple C program with passwords drawn randomly from a dictionary to compute the verifier by the OpenSSL function `SRP_compute_client_key`. As motivated earlier, we used the generator $g = 5$. Specifically, we worked with the 6144 bit MODP group, described in [43], so the client uses the largest modulus available with this generator. Group upgrade/downgrade attacks are discussed in subsection 2.3. Other parameters (the salt for instance) are randomly generated to avoid the setup of a server.

We then run the spy process collecting our traces that distinguish the different iterations and that determine in which one an accumulator overflows.

Once the spyware runs as a background process, the execution of our victim program will produce an execution trace. Figure 2 depicts a part of such a trace. The red line represents the threshold we used during the Flush+Reload: any value below this threshold means an execution of the monitored code (*i.e.* a new iteration). Any value above the threshold means that the victim is performing some other computation. In this trace sample, we can identify two groups of iterations, separated by a significant time gap (horizontal gap). This gap is what identifies an overflow: the iteration is slightly longer because of the additional operations and the delay caused

[2]https://gitlab.inria.fr/ddealmei/poc-openssl-srp

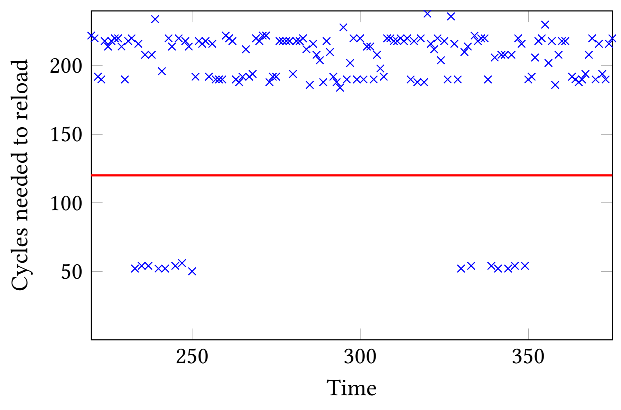

Figure 2: Partial execution trace using the password `dAdinretAm`, username `admin` and salt `0102030405060708`. Any value below the threshold represents an execution of the monitored instruction (i.e. a different bit being processed). We can identify two groups, with the horizontal gap between them representing an overflow of the accumulator.

by the PDA. The full trace is presented in Appendix B. We note that the trace collection is as fast as an SRP session establishment.

4.2.2 Trace interpretation. From the data acquired by the spy process, we need to extract exploitable information on the exponent. To avoid the cumbersome task of manually interpreting the trace graph, we implemented a parser script taking a raw trace, and outputting the list of pattern length composing the exponent.

The parser works as follows: after sanitizing the raw trace (by removing trailing measurement detecting no activity, and the trace header containing system information), we used some heuristic methods to produce a more reliable trace.

First, outlier values are removed, and impossible patterns close to the minimal pattern size are rounded up (these errors are usually caused by the spy missing an iteration). In practice, it means that a three-bit pattern is considered as a four-bit pattern. Likewise, any isolated iteration (one or two-bit pattern) is removed from the trace, since it can be considered as an error in the measurement, or the consequence of a misprediction caused by the processor.

Then, we convert the trace into a readable form by only noting the number of iterations for each chunk in the trace. We used this representation as an input to our dictionary reduction program.

4.2.3 Dictionary reduction. This part of the attack is generic, since it behaves as a classical dictionary partitioning attack, as seen in [14, 44]. The idea is to take a list of potential passwords, and see if each candidate matches the trace we collected on the secret password. If the candidate does not match, we discard the password. Otherwise, we keep it and can apply the same process on a new leak (changing the salt for instance) to further reduce the list of candidates until an online attack becomes reasonable.

Precise microarchitectural measurements being hard to get perfect, some minor errors may slip into our traces. Hence, in a realistic scenario, we cannot expect to have a perfect match between the trace we collect, and the real trace of the password. To circumvent this constraint, we build a scoring system, by computing a differential score for each candidate based on how different it is to the reference trace (see Appendix C for the detailed computation). For each candidate yielding a score below a predefined threshold (set to 10 in our experiments), the password is returned, along with its score, as a valid candidate. If no candidate is returned (meaning there is an issue with the trace), the lowest-score candidate is returned as the most probable password.

4.3 Results

We tested our Proof of Concept on 100 different passwords, drawn randomly from the Rockyou dictionary [37]. Since the exponent generation behaves as a random oracle seeded by the password, the username and the salt, we kept a constant value for the salt and the username. Moreover, the password length and format do not impact the exponent, nor the leaked information.

For each password, we repeated our attack 15 times, yielding a total of 1500 different traces. Then, we parsed each trace independently, and tried to recover the password in the dictionary. We also computed the average number of traces needed to achieve a reliable result by obtaining multiple traces on the same password.

4.3.1 Reliability of our measurements. In 96.9% of our experiments, a single measurement is enough to find the password with high probability. On the remaining 3.1%, namely the errors, the result is clearly inconclusive since the score is above 20, whereas we often get a score below 10 when the valid password is found. Repeating the measurement twice was enough to achieve 100% accuracy.

It is worth noting that some measurements were totally inaccurate for unknown reasons, resulting in an non-exploitable trace. In this case, we performed the measurement again. This only represents a negligible amount of our measurements, and only occurred during intensive testing sessions.

4.3.2 Average information per trace. Due to the quick iterations prior to the use of the Montgomery representation (*i.e.* before the first accumulator overflows), the beginning of the trace tends to be noisy. Thus, we were not able to reliably and consistently guess exactly how many MSB were skipped. Instead, we considered the total number of detected patterns, otherwise stated, the number of overflows on the accumulator during the exponentiation.

Our experiments show that we can guess 63.9 bits per trace on average. However, not all these guessed bits are accurate. Indeed, because of the noisy measurements, we notice an average difference of 3.9 bits with the reference hash value computed from the secret password. This small difference did not prevent us from correctly recovering the password (thanks to our scoring technique).

It is interesting to see that, in theory, given an 160-bit exponent, we expect an average leakage of 65.5 bits of information on x (see subsection 3.5 for details). Our practical results show that we are able to extract almost as much information as we could expect from a single measurement. The small loss of information, nearly 8.4%, comes from the lack of precision in the beginning of the trace.

We stress that using a stronger hash function (SHA-256 or SHA-512 for instance) would result in a longer digest, allowing to acquire even more information on the exponent and the password.

Table 1: Cost of dictionary reduction for various dictionaries

Dictionary	Nb. entries	Reduction time	Cost ($)
Rockyou	$1.4 \cdot 10^7$	0.00029 s	$2.6 \cdot 10^{-6}$
CrackStation	$3.5 \cdot 10^7$	0.00073 s	$6.6 \cdot 10^{-6}$
HaveIBeenPwned	$5.5 \cdot 10^8$	0.014 s	$1 \cdot 10^{-4}$
8 characters	$4.6 \cdot 10^{14}$	2h40	87.23

4.3.3 Practical dictionary attack. As discussed in subsection 3.6, recovering k bits of information on the exponent allows an attacker to lower the probability of having a false negative to 2^{-k}, while eliminating wrong passwords.

The bottleneck of the attack being the dictionary reduction, we estimated its complexity in practice. As OpenSSL relies on SHA-1 in SRP, the cost of testing each password is dominated by the complexity of computing two SHA-1 digests on small data. For the following computation, we will assume that our attacker uses a single, publicly available, Amazon AWS p4d.24xlarge instance that runs eight NVIDIA A100 GPUs. Based on Hashcat benchmark for SHA-1 [17], a single GPU can compute $6.1 \cdot 10^9$ password candidates per second. Therefore, a single AWS instance can roughly evaluate $4.8 \cdot 10^{10}$ password candidates per second.

Table 1 presents the cost for reducing some relevant dictionaries. We used the current price of an Amazon AWS p4d.24xlarge instance as a reference [4], which is currently 32.77$ per hour. The dictionaries that we present have been used in previous work [14, 44] to demonstrate how practical offline dictionary attacks are on PAKEs.

Considering the size of these dictionaries and our experimental results, a single (reliable) measurement allows an attacker to recover enough information on the password to reduce the entire dictionary.

5 IMPACT ANALYSIS

OpenSSL is one of the most popular and deployed cryptographic libraries [36]. This explains why their SRP implementation is used as a reference for many third-party projects: both amateur and industrial. More interestingly, other popular languages, such as JavaScript, Ruby and Erlang rely on OpenSSL for big numbers operations. To our surprise, we discovered that the scope of our vulnerability is much broader than the OpenSSL TLS-SRP.

Since our attack targets the dynamic library, our exploit works on other vulnerable codes, regardless the used programming language as long as it is linked to the vulnerable version of OpenSSL. Due to the wide range of impacted projects, we did not implement each attack scenario, but designed an exploit for OpenSSL, node-bignum, Apple HomeKit ADK, and PySRP and only evaluated the impact for other projects.

In this section, we go through the most popular open-source and proprietary projects that we found to be vulnerable. In most cases, we noticed that the big numbers or the cryptographic libraries of a specific language is based on OpenSSL, making any SRP project based on this standard library vulnerable. Due to the wide variety of projects, and the low code transparency of closed-source solutions, the real scope of our vulnerability is hard to establish. Thus, we underline that this impact analysis is far from being exhaustive. Furthermore, practical attack consequences may vary between applications depending on the context, and thus can hardly be narrowly defined. A general note on password recovery consequences in SRP can be found in subsection 3.7. Table 2 sums up the vulnerable projects that we identified in the wild.

Table 2: Non-exhaustive list of vulnerable projects. Projects with a * are not patched. Vulnerable versions are inclusive.

Software/Library	Language	Vulnerable versions
OpenSSL	C	Up to v1.1.1i
Apple HomeKit ADK	C	no release
CMaNGOS	C	no release*
GoToMeeting	closed source	Up to v10.15.0*
GoToWebinar	closed source	Up to v10.15.0*
GoToTeaching	closed source	Up to v10.15.0*
GoToRemote	closed source	Up to v10.15.0*
node-bignum	JavaScript	Up to 0.13.1*
node-srp	JavaScript	Up to 0.2.0*
node-srp-typescript	JavaScript	Up to 0.3.0*
caf_srp	JavaScript	no release
OTP	Erlang	Up to 23.2.1
strap	Elixir	no release*
srp-elixir	Elixir	0.2.0*
ruby/openssl	Ruby	Up to 2.2.0*
sirp	Ruby	Up to 2.0.0*
pysrp	Python	Up to 1.0.16
xbee-python	Python	Up to 1.3.0*
proton-python-client	Python	no release

5.1 TLS-SRP using OpenSSL

As release 1.0.1, OpenSSL offers an implementation for TLS-SRP [43] that uses a shared password to establish a secure session. There are several reasons to use TLS-SRP. First, using password-based authentication does not require reliance on certificate authorities. Second, it provides mutual authentication, while TLS with server certificates only authenticates the server to the client. Third, phishing attacks are harder to perform, since the authentication does not involve to check the URL being certified.

Our main contribution is illustrated by exploiting this SRP. Thus, one might argue that the scope of our work is limited for two reasons. Firstly, TLS-SRP is not supported by any browser, and therefore is not widely used over the Internet. Secondly, TLS 1.3 does not provide any support for PAKE yet [8], and might never support SRP, since it is not part of the CFRG PAKE selection [16].

Despite these reasons, we argue that the interest of our work is much broader than just TLS-SRP. Indeed, we found that the OpenSSL implementation is used as is in many other projects. In particular, the vulnerability is caused by an insecure call to the function `BN_mod_exp`, without setting the constant-time flag. Developers are not to blame here, because the documentation of `BN_mod_exp` never mentions that, by default, this function is not constant time if the base fits in a processor word. This is unfortunately the case in SRP, and might be also in other protocols.

5.2 Stanford Reference Implementation

The Stanford University has dedicated a web page for SRP, since the project started at it. The website hosts some pages describing the protocol evolution and its various use-cases. Namely, it provides reference implementations: a standalone Java implementation (relying on the default JDK), as well as a C implementation (relying on a third-party library to handle big numbers and cryptographic operations). We have studied both.

The Java implementation is completely flawed, but out of the scope of our contribution. We give an overview of the problems in Appendix D, and strongly discourage its usage.

As for the C implementation, it only performs SRP-related operations: manages messages and parameters, keeps track of the protocol state and calls the appropriate callbacks. The underlying mathematical operations are provided by a third-party library. The project supports and recommends the following libraries: OpenSSL, Cryptolib, GNU_MP, MPI and TomMath/TomCrypt. Among these libraries, OpenSSL is likely to be the choice of many developers, due to its large support in many systems. Our study shows that the modular exponentiation API for OpenSSL uses the vulnerable function call of `BN_mod_exp`, making this default reference implementation vulnerable. Moreover, we verified that the generic attacks discussed in subsection 2.3 are possible, since both the salt and the group parameters are transmitted without any protection.

The Stanford implementation draws its importance from the fact that it is one of the first available C implementations with no restrictive license. Therefore, it is fair to assume that it be using elsewhere. Nevertheless, it is not clear how many projects are impacted, especially the proprietary ones whose code is not open-source.

5.3 Apple HomeKit Accessory Development Kit

Apple HomeKit allows users to communicate with and control accessories (going from garage doors to smart light-bulb or doorbell) through a dedicated application. This communication goes through the HomeKit Accessory Protocol (HAP) that is deployed on more than a billion Apple devices.

When a new device is first introduced into the network, the user must pair it with the application. To do so, a secure session is established with a modern variant of SRP. Namely, the following changes are made (Stanford's website is explicitly cited): (i) SHA-1 is replaced by SHA-512, (ii) only the 3072-bit modulus group from [43] is supported. In this setup, the application acts as a client, the device is the server holding the verifier, and the password is a code displayed on the device. The SRP session is used to exchange long-term Ed25519 keys, that are later used for all future sessions.

Apple's specification mandates a device code conforming to `XXX-XX-XXX`, meaning only 10^8 passwords are possible. Considering such a restricted set of password candidates and the reliability of our measurement, an attacker would be able to guess the code on-the-fly from a single session, and recover the long term key.

If commercial accessories must use Apple's HomeKit ADK through the MFi program, the open-source version relying on OpenSSL is available on Github [5]. This implementation calls `BN_mod_exp` on fresh big numbers (without modifying any flag) on the client device. Hence, it is vulnerable to our attack. The large exponent size, because of SHA-512, allows the attacker to get about 205.7 bits of information, which are more than needed to recover the device code. In this context, the consequences of our attack are disastrous: an attacker would have full control on the targeted smart device if they can impersonate the user during the pairing.

As a side note, we stress that our current attack focuses on particular properties found on Intel processors, which are usually not valid for ARM processors (commonly used on iPhones, iPads and recent Mac). Namely, non inclusiveness of the LLC and the absence of a user-land flush instruction make it harder to perform cross core attacks. However, all these obstacles can be circumvented as demonstrated in the work of Lipp *et al.* [32]. We encourage readers to refer to this article for further details on how to adapt the attack for ARM architecture. Furthermore, Mac computers manufactured before late-2020 integrates an Intel processor, and the Home application can be installed on them, making the current attack valid on a significant number of machines.

5.4 LogMeIn Tools

LogMeIn provides various tools for remote communications and collaborations services. We analyzed four of these tools: GoToMeeting (designed for visio-conferences), GoToWebinar (designed for webinar), GoToTraining (designed for teaching classes) and GoToAssit (remote IT assistance). All these tools have a common point: they rely on SRP for user authentication [33]. Since these tools are proprietary, we performed some reverse-engineering on their Windows 10 binaries, and found that they use OpenSSL for SRP. The impact of our attack varies according to the targeted tool.

Concerning GoToMeeting, SRP is used to authenticate users in their visio-conference solutions. Our attack allows the recovery of users' passwords, and therefore the solution would suffer from users impersonation. Similar consequences are related to GoToWebinar and GoToTraining, since they share a significant part of their security specifications with GoToMeeting [34].

As for GoToAssist, SRP is used to establish the secure session protecting screen-sharing, keyboard/mouse control, and diagnostic data and text chat [33]. Thus, an attacker recovering the secret password may be able to put themselves in an active MitM position, achieving complete control over the victim computer by impersonating the IT support, until the end of the connection.

5.5 ProtonMail Python Client

According to the security description of ProtonMail [1], each user defines two passwords: the mailbox password that is used to protect the private key encrypting messages, and the login password that is used along with SRP to provide remote authentication.

ProtonMail customizes their SRP implementation with several technical choices. First, they replaced SHA-1 by a combination of bcrypt and expanded SHA-512, so the attacker would require more time to compute the digest for each candidate (an attacker could test roughly $4.2 \cdot 10^5$ passwords per second, on the setup described in subsubsection 4.3.3). Note that using such a long exponent (2048 bits exponent) would leak a huge amount of information through our attack. Second, they use non-standard group parameters. Indeed, in order to avoid large scale pre-computation attacks, they generate a new 2048-bit modulus for each user. However, since the generator is fixed to $g = 2$, our attack still applies.

Note that recovering the login password would allow an attacker to connect to the victim account, but not have access to their messages or their private key (assuming a different password has been used). However, the login password is shared with ProtonVPN. Therefore, recovering it allows the attacker to take complete control of the victim's ProtonVPN account, thereby benefiting from their subscription and viewing sensitive data, such as billing information

It is worth noting that not all the implementations of ProtonMail are concerned by our attack. For instance, we did not find any OpenSSL call in the JavaScript client. In fact, only the python client is vulnerable, as it is based on the project pysrp. More details about this project are found in Appendix E.

5.6 Big Numbers Libraries and SRP in the Wild

To handle big integers, high-level programming languages typically have two choices: (i) re-implement a library or a package to give support; (ii) use an implementation that is provided by a low-level language. In practice, the latter is a quite popular approach, especially when we consider efficiency during cryptographic operations.

A preliminary review of core libraries of various open-source languages reveals that OpenSSL, through different wrappers, is often used to manipulate big numbers. Therefore, these projects inherit the OpenSSL vulnerability. In particular, we observed that all SRP projects based on OpenSSL, or using a vulnerable SRP project, can be exploited by our attack. Namely, we identified the vulnerability in a JavaScript node to handle big numbers (node-bignum, used in more than 2500 other projects that we did not analyze), Ruby's official OpenSSL package, Erlang and Elixir OTP library and a popular Python SRP implementation. For more information on these vulnerable projects, we defer the reader to Appendix E.

6 MITIGATION & DISCLOSURE

Our attacks were performed on the most updated versions of the evaluated projects, as published at the time of discovery. Following the practice of responsible disclosure, we timely disclose our findings to all projects mentioned in Table 2. We further participated in the design and the empirical verification of the proposed countermeasures. We describe a brief overview of our communications.

The Case of OpenSSL. We started by contacting OpenSSL following their security policy. They acknowledged the attack and were able to reproduce it using our docker container. Therefore, they decided to mitigate the attack and include the patch in the next release. To our surprise, they refused to publish a related CVE because they claimed that cache attacks are out of scope of their threat model. We argued back that a CVE may help to shed some light on the problem, as we cannot reach all the proprietary solutions relying on OpenSSL. However, the final decision was not to issue any CVE.

Then, we discussed with OpenSSL the best fix to deploy. Everyone agreed that the mitigation should be to prevent the function `BN_mod_exp` from calling the vulnerable function `BN_mod_exp_mont_word`. There are two options to apply this. First, the function `BN_mod_exp` is re-written, so that it never calls the vulnerable function. Second, the call of the function `BN_mod_exp` is modified, so that the flag `BN_FLG_CONSTTIME` is set on the exponent.

No option was perfect. Indeed, the first option would have increased the execution time of other code using the optimized modular exponentiation. However, it might fix all projects implementing SRP through the big integers API of OpenSSL. The latter option is easier to test and validate, as it solely concerns the SRP implementation in crypto/srp/srp_lib.c. However, it only fixes the particular SRP implementation of OpenSSL, but not the projects directly calling `BN_mod_exp`. OpenSSL maintainers took a variant of the second option, as they also set the constant-time flag on the result of the function `BN_mod_exp`. Thus, we had no choice, but to contact the vulnerable implementations that we previously identified.

It is worth noting that the patch introduced some overhead, since the optimized function is not called anymore. We empirically evaluated the incurred overhead by running our benchmark 30000 times with random values. Our results show an overhead of 11.18% on the execution of `SRP_Calc_client_key`. This overhead might explain the rational about keeping the function `BN_mod_exp_mont_word` that might be used in applications requiring fast execution.

The Remaining Projects. We ended up by contacting the remaining projects, since the OpenSSL patch does not cover them. Two projects are distinguished: Ruby and Erlang/OTP. Indeed, they quickly patched their implementation after our report and our validation of their suggested fix. Ruby ranked our vulnerability 8.7/10 in terms of severity, and took care of notifying all Ruby-based projects. We designed and integrated a patch for four other projects: Proton Mail (who acknowledge our vulnerability through their bug bounty program), the JavaScript node-bignum, the Python pysrp, and the Apple Homekit. The patch for node-bignum is still to be integrated. As for World of Warcraft, they took some time to acknowledge our attack, but we are still in discussion to agree on the best solution for their project. The issues identified in GoTo tools and Digi Xbee have been assigned CVE-2021-25279 and CVE-2021-25280 respectively, but we did not get further answer from them. Finally, we mention the case of cafjs, whose maintainer recognized the vulnerability, but never replied to our subsequent emails proposing a mitigation.

7 RELATED WORK

Cache attacks represent a special case of timing attacks on software implementations, exploiting cache and micro-architectural properties of the victim station. First introduced by Bernstein on AES [12] in 2005, they took some time to be fully popularized. The Flush+Reload attack, first proposed by Yarom and Falkner in [50], leverages specific properties, found in Intel processors, to soften the requirements and allow high resolution cross-core attacks. Yarom *et al.* demonstrated their attack by extracting nearly all bits of RSA secret exponent in GnuPG library. This attack has been extended and made more reliable by pairing it with a Performance Degradation Attack in [3] by Allan *et al.* as explained in section 2. Yarom proposed the Mastik toolkit for exploiting such vulnerabilities [49]. For the ARM processors, this kind of attack has been ported by Lipp *et al.* in [32], explaining how to bypass many obstacles.

The big integer modular exponentiation of OpenSSL has been a popular target for side channel attacks since the work of Percival in 2005 [39], recovering secret exponent bits used as lookup indexes during the Sliding Window Exponentiation (SWE). Consequently, OpenSSL introduced a constant-time flag to identify sensitive BIGNUM and process them securely. It also resulted in a constant-time alternative for the modular exponentiation. However, related vulnerabilities continue to exist. Indeed, flag inheritance, legacy code and independent update of the complex code base have repeatedly led to using insecure variant of sensitive operations on secret inputs. Focusing only on the modular exponentiation, the work of Garcia *et al.* on DSA [22] describes how a lack of flag inheritance caused DSA to use the insecure SWE when signing, thereby exposing to the attack described by Percival [39]. Using Flush+Reload and a PDA, they were able to improve the original attack, and recover the full private key with a reasonable amount of measurements. In 2019, Aldaya *et al.* outlined the use of several known side-channel vulnerable functions in OpenSSL [2]. Although the core of their work is the RSA key generation process (particularly the binary GCD), they mention a security risk related to the use of an insecure SWE during the Miller-Rabin primality test. This could be exploited to leak information on the primes, using a similar attack as described in previous works. In 2020, Garcia *et al.* described another attack on DSA [23] due to an insecure call to the modular exponentiation when processing a DSA private key of a specific format. In this context, the default modular exponentiation (SWE) is called to compute the corresponding public key, and the key can be recovered accordingly.

It is true that our contribution also arises from bad flag management, but we successfully show another scenario in which different insecure functions are called because of the exponent size. In contrast to previous work, we are able to recover enough information with one single trace, by leveraging pattern identification rather than bit-by-bit recovery. Moreover, since we consider the case of PAKE, any bit of information can be used to drastically reduce the set of possible inputs. Thus, we provide a practical use-case, where we do not need to recover the complete value that we are looking to guess. Instead, any bit of leaked information actually breaks the security property of the protocol that, we recall, is designed to resist offline dictionary attacks.

Recently, some software attacks have targeted the Dragonfly PAKE implemented in WPA3 by Vanhoef and Ronen in [44]. The cache attack they described has been extended to a more efficient attack on FreeRadius and iwd implementations in [14] by Braga *et al.* Both works exploit the hash-to-curve function mapping a password to an elliptic curve point. These works show that PAKE protocols are particularly vulnerable to any secret leakage because of the nature of passwords.

SRP is a widely used protocol without security proof despite a formal analysis performed by Sherman *et al.* in [42]. Some attacks have been identified and lead to the version 6a of this protocol. However, there is still no real attack on this protocol. In [45], a bug has been identified on cSRP (affecting also PySRP) and srpforjava implementations. Instead of computing $kv + g^b \bmod p$, the implementations calculate $kv + (g^b \bmod p)$, allowing to recover the most significant bits of v and hence perform a dictionary attack. To the best of our knowledge, our attack is the first cache attack to be performed on SRP.

8 CONCLUSION

Despite being disapproved by some of the crypto community, the SRP protocol is still widely deployed in practice. Our work analyzes the security of some popular SRP implementations. In this paper, we identified that the function BN_mod_exp of OpenSSL, which is often used in SRP code, is called in a way that leads to non-constant time execution of the protocol. Then, we exploited this property through cache attacks in order to leak some sensitive data. The leakage revealed enough data about the password to perform a complete offline dictionary attack against SRP. We showed that our attack is practical, since it only requires one single trace, despite the noise of cache measurements. In addition, it is quite efficient as the reduction of some common dictionaries is almost instantaneous with modern resources at negligible cost.

Our contribution is part of a long line of attacks caused by a flawed mitigation system, designed to prioritize performance over security, even for sensitive operations. Indeed, sensitive inputs need to be explicitly flagged to call secure implementations. This particular design makes the patching process messier, since the OpenSSL patch is only valid for their SRP implementations. Hence, all vulnerable projects need to independently provide their own patch. This unfortunate situation makes it tedious to fix all the impacted projects, as we might have missed most of the proprietary ones. Thus, we believe that secure implementations should be provided by default when the associated overhead can be afforded.

A final long-term lesson of our work is that cache attacks are now proved to be practical, and represent a real danger. We also show that any leakage inside PAKE protocols can be successfully exploited because of the low entropy of passwords. Standards and RFC proposals already take these threats into account, and require cryptographic implementations to be resilient against side-channel attacks. We regret that mainstream libraries, such as OpenSSL, do not include them on their threat model. Indeed, acknowledging and patching is a good first step, but issuing a CVE would help raising awareness within the community about these issues.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their time and constructive feedback. Daniel De Almeida Braga is funded by the Direction Générale de l'Armement (Pôle de Recherche CYBER). This work was supported in part by the French ANR grant Cyberschool ANR-18-EURE-0004.

REFERENCES

[1] Proton Technologies A.G. 2016. ProtonMail Security Features and Infrastructure. https://protonmail.com/blog/wp-content/uploads/2016/12/ProtonMail_Authentication_excerpt.pdf

[2] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. 2019. Cache-Timing Attacks on RSA Key Generation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019, 4 (2019), 213–242.

[3] Thomas Allan, Billy Bob Brumley, Katrina E. Falkner, Joop van de Pol, and Yuval Yarom. 2016. Amplifying side channels through performance degradation. In *ACSAC*. ACM, 422–435.

[4] Amazon. 2020. Amazon EC2 P4d Instances. https://aws.amazon.com/fr/ec2/instance-types/p4/

[5] Apple. 2020. HomeKit Accessory Development Kit (ADK). https://github.com/apple/HomeKitADK

[6] Hamza Assyad. 2018. Now generally available: Amazon CognitoAuthentication Extension Library. https://aws.amazon.com/blogs/developer/now-generally-available-amazon-cognitoauthentication-extension-library

[7] Elaine B. Barker and Quynh Dang. 2015. *Recommendation for Key Management, Part 3: Application-Specific Key Management Guidance.* Technical Report. NIST. SP800-57 Part 3 Rev.1.
[8] Richard Barnes and Owen Friel. 2018. *Usage of PAKE with TLS 1.3.* Internet-Draft draft-barnes-tls-pake-04. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-barnes-tls-pake-04 Work in Progress.
[9] Mihir Bellare, David Pointcheval, and Phillip Rogaway. 2000. Authenticated Key Exchange Secure against Dictionary Attacks. In *EUROCRYPT (Lecture Notes in Computer Science)*, Vol. 1807. Springer, 139–155.
[10] Steven M. Bellovin and Michael Merritt. 1992. Encrypted key exchange: password-based protocols secure against dictionary attacks. In *IEEE Symposium on Security and Privacy.* IEEE Computer Society, 72–84.
[11] Steven M. Bellovin and Michael Merritt. 1993. Augmented Encrypted Key Exchange: A Password-Based Protocol Secure against Dictionary Attacks and Password File Compromise. In *CCS.* ACM, 244–250.
[12] Daniel J. Bernstein. 2005. Cache-timing attacks on AES. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf
[13] Victor Boyko, Philip D. MacKenzie, and Sarvar Patel. 2000. Provably Secure Password-Authenticated Key Exchange Using Diffie-Hellman. In *EUROCRYPT (Lecture Notes in Computer Science)*, Vol. 1807. Springer, 156–171.
[14] Daniel De Almeida Braga, Pierre-Alain Fouque, and Mohamed Sabt. 2020. Dragonblood is Still Leaking: Practical Cache-based Side-Channel in the Wild. In *ACSAC.* ACM, 291–303.
[15] Elie Bursztein. 2018. The bleak picture of two-factor authentication adoption in the wild. https://elie.net/blog/security/the-bleak-picture-of-two-factor-authentication-adoption-in-the-wild
[16] CFRG. 2020. PAKE Selection. https://github.com/cfrg/pake-selection
[17] Chick3nman. 2020. Hashcat v6.1.1 benchmark on the Nvidia Tesla A100 PCIE variant GPU. https://gist.github.com/Chick3nman/d65bcd5c137626c0fcb05078bba9ca89
[18] John Engler, Chris Karlof, Elaine Shi, and Dawn Song. 2009. Is it too late for PAKE?. In *W2SP.* The Internet Society.
[19] Rob Faludi. [n.d.]. Who uses Erlang for product development? https://erlang.org/faq/introduction.html#idp32560400
[20] Rob Faludi. 2017. Introducing the Official Digi XBee Python Library. https://www.digi.com/blog/post/introducing-the-official-digi-xbee-python-library
[21] Rick Fillion. 2018. Developers: How we use SRP, and you can too. https://blog.1password.com/developers-how-we-use-srp-and-you-can-too
[22] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. 2016. "Make Sure DSA Signing Exponentiations Really are Constant-Time". In *CCS.* ACM, 1639–1650.
[23] Cesar Pereida García, Sohaib ul Hassan, Nicola Tuveri, Iaroslav Gridin, Alejandro Cabrera Aldaya, and Billy Bob Brumley. 2020. Certified Side Channels. In *USENIX Security Symposium.* USENIX Association, 2021–2038.
[24] Craig Gentry, Philip D. MacKenzie, and Zulfikar Ramzan. 2006. A Method for Making Password-Based Key Exchange Resilient to Server Compromise. In *CRYPTO (Lecture Notes in Computer Science)*, Vol. 4117. Springer, 142–159.
[25] Matthew Green. 2018. Should you use SRP? https://blog.cryptographyengineering.com/should-you-use-srp
[26] Björn Haase and Benoît Labrique. 2019. AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019, 2 (2019), 1–48.
[27] Feng Hao and Peter Ryan. 2010. J-PAKE: Authenticated Key Exchange without PKI. *Trans. Comput. Sci.* 11 (2010), 192–206.
[28] Dan Harkins. 2015. Dragonfly Key Exchange. *RFC* 7664 (2015), 1–18.
[29] ING Bank. 2018. InsideBusiness App security. https://new.ingwb.com/en/service/insidebusiness-app/insidebusiness-app-security
[30] David P. Jablon. 1996. Strong password-only authenticated key exchange. *Comput. Commun. Rev.* 26, 5 (1996), 5–26.
[31] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. 2018. OPAQUE: An Asymmetric PAKE Protocol Secure Against Pre-computation Attacks. In *EUROCRYPT (3) (Lecture Notes in Computer Science)*, Vol. 10822. Springer, 456–486.
[32] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium.* USENIX Association, 549–564.
[33] LogMeIn. 2017. Remote Support and Service Desk Security. https://assets.cdngetgo.com/b0/b1/97dd1f0e42fa8b8e5433333dffae/gotoassist-remotesupport-servicedesk-security-white-paper.pdf
[34] LogMeIn. 2020. Web conference security. https://logmeincdn.azureedge.net/gotomeetingmedia/-/media/pdfs/UCC_security_white_paper_snapshot_April2020.pdf
[35] P. Montgomery. 1985. Modular multiplication without trial division. *Math. Comp.* 44 (1985), 519–521.
[36] Matús Nemec, Dusan Klinec, Petr Svenda, Peter Sekan, and Vashek Matyas. 2017. Measuring Popularity of Cryptographic Libraries in Internet-Wide Scans. In *ACSAC.* ACM, 162–175.
[37] Cubrilovic Nik. 2009. RockYou Hack: From Bad To Worse. https://techcrunch.com/2009/12/14/rockyou-hack-security-myspace-facebook-passwords/
[38] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *CCS.* ACM, 1406–1418.
[39] Colin Percival. 2005. Cache missing for fun and profit.
[40] Proton Mail Technologies. 2016. ProtonMail Security Features and Infrastructure. https://protonmail.com/blog/wp-content/uploads/2016/12/ProtonMail_Authentication_excerpt.pdf
[41] Jörn-Marc Schmidt. 2017. Requirements for Password-Authenticated Key Agreement (PAKE) Schemes. *RFC* 8125 (2017), 1–10.
[42] Alan T. Sherman, Erin Lanus, Moses Liskov, Edward Zieglar, Richard Chang, Enis Golaszewski, Ryan Wnuk-Fink, Cyrus J. Bonyadi, Mario Yaksetig, and Ian Blumenfeld. 2020. Formal Methods Analysis of the Secure Remote Password Protocol. *CoRR* abs/2003.07421 (2020).
[43] David Taylor, Thomas Wu, Nikos Mavrogiannopoulos, and Trevor Perrin. 2007. Using the Secure Remote Password (SRP) Protocol for TLS Authentication. *RFC* 5054 (2007), 1–24.
[44] Mathy Vanhoef and Eyal Ronen. 2020. Dragonblood: Analyzing the Dragonfly Handshake of WPA3 and EAP-pwd. In *IEEE Symposium on Security and Privacy.* IEEE, 517–533.
[45] Ronald Volgers. 2016. Exploiting two buggy SRP implementations. https://www.computest.nl/nl/knowledge-platform/blog/exploiting-two-buggy-srp-implementations/
[46] Thomas Wu. 2000. The SRP Authentication and Key Exchange System. *RFC* 2945 (2000), 1–8.
[47] Thomas Wu. 2000. Telnet Authentication: SRP. *RFC* 2944 (2000), 1–7.
[48] Thomas D. Wu. 1998. The Secure Remote Password Protocol. In *NDSS.* The Internet Society.
[49] Yuval Yarom. 2016. Mastik: A Micro-Architectural Side-Channel Toolkit. https://cs.adelaide.edu.au/~yval/Mastik/
[50] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium.* USENIX Association, 719–732.

A AVERAGE LEAKAGE FOR STANDARD GENERATORS

In this appendix, we detail how we estimated which one of the defined generators offers the best average leak for an attacker.

We start by implementing a script to produce all possible bit patterns leading to an overflow of the accumulator. Assuming we start with a fresh accumulator $w = g$, we notice that: (i) $g = 2$ implies that w always overflows after five more iterations, (ii) $g = 5$ implies that w can overflow after three or four iterations, and (iii) $g = 19$ implies that w always overflows after three more iterations.

We will start by the case $g = 5$ since it is the more complex. Computation are fairly similar for $g = 2$ and $g = 19$. We recall that the information given by an event E of probability p_E can be computed as $I(E) = -\log_2(p_E)$.

Let k denote the number of iterations in the trace, and ℓ be bit-length of the exponent (including zero bits needed to match the length defined in the implementation specification).

A.1 g = 5

A.1.1 Leaks from the leading bits. On average, we can expect $\sum_{i=0}^{\ell} i2^{-i} = 2$ leading zero bits. Then, since the square-and-multiply loop starts after the first bit sets to one, the accumulator is initialized to $w = g$, and three patterns are possible before the first overflow: (i) $111b$ with probability 2^{-2}, (ii) $110bb$ with probability 2^{-2}, (iii) $10bbb$ with probability 2^{-1}. The average length of the first undetected pattern is therefore 4.75.

Hence, for a random exponent, we expect to detect $\ell - 6.75$ iterations on average. In addition, the average information we can recover from the leading bits is expressed as

$$\mathcal{L}_{lead} = 2 + 3 \times 2^{-2} + 3 \times 2^{-2} + 2 \times 2^{-1} = 4.5.$$

Note that, assuming ℓ is large enough, its impact on the leakage through the leading bits is negligible.

A.1.2 Leaks from the patterns. Next, we can determine the leakage by identifying each pattern as described in subsubsection 3.4.2. Let $\lambda \geq 4$ denote the number of iterations in the pattern.

- $\lambda = 4$: the three first bits are set to 1, which occurs with probability $p_4 = 2^{-3}$, thereby leaking 3 bits of information.
- $\lambda = 5$: the pattern is in the set $\{10bbb,\ 110bb,\ 0111b\}$. This occurs with probability $p_5 = 2^{-2} + 2^{-3} + 2^{-4}$ and leaks approximately 1.2 bits of information.
- $\lambda \geq 6$: the $\lambda - 5$ firsts bits of the pattern are 0. This occurs with probability $p_\lambda = 2^{-\lambda+5}$ and leaks $\lambda - 5$ bits of information. Furthermore, the last 5 bits verify the previous condition, meaning we get 1.2 additional bits of information.

The last two cases can be combined to get a pattern with $\lambda \geq 5$ bits and probability $p_\lambda = \left(2^{-2} + 2^{-3} + 2^{-4}\right) 2^{-(\lambda-5)}$, leaking $\lambda - 5 + 1.2$ bits of information.

Let L be the random variable corresponding to the information leaked by a pattern. We get the average leak from a pattern with

$$\mathbb{E}[L] = \sum_{i=4}^{\ell} p_i I(p_i) = p_4 I(p_4) + \sum_{i=5}^{\ell} p_i I(p_i) \approx 2.29.$$

Let N be the random variable corresponding to the length of a pattern. The average length of a pattern is:

$$\mathbb{E}[N] = \sum_{i=4}^{\ell} i p_i = 4p_4 + \sum_{i=5}^{\ell} i p_i \approx 5.75.$$

Finally, we can get the amount of leaked information by estimating the number of patterns we can expect based on the length k of the bit string we are considering:

$$\mathcal{L}_{patterns}(k) = \mathbb{E}[L] \times \frac{k}{\mathbb{E}[N]} \approx 0.4 \cdot k.$$

Hence, the average leakage from an ℓ-bit random exponent is

$$\mathcal{L}(\ell) = \mathcal{L}_{lead} + \mathcal{L}_{patterns}(\ell - 6.75) \approx 0.4 \cdot \ell + 2.06.$$

A.2 g = 2

Since all overflows occur after five iterations once $w = g$, the pattern in our trace looks like $t = 0 \ldots 01bbbbb$, with $|t| = \lambda > 5$, with probability $p_\lambda = 2^{-\lambda+6} \times 2^{-1}$, leaking $I(p_\lambda) = \lambda - 5$ bits of information. The average information leak per pattern is therefore

$$\mathbb{E}[L] = \sum_{i=6}^{\ell} p_i \times I(p_i) = \sum_{i=6}^{\ell} 2^{-i+5} \times (i-5) \approx 2.$$

Following the same reasoning, we can get the average pattern length $\mathbb{E}[N]$ and deduce the expected leakage for an ℓ-bit exponent:

$$\mathbb{E}[N] = \sum_{i=6}^{\ell} p_i \times i = \sum_{i=6}^{\ell} 2^{-i+5} \times i \approx 7.$$

Hence, the average leakage from the patterns is equal to

$$\mathcal{L}_{patterns}(k) = \mathbb{E}[L] \times \frac{k}{\mathbb{E}[N]} \approx \frac{2}{7}k$$

The average number of leading zero bits does not depend on the generator, but the information leaked on the first pattern leading to the first overflow does. Here, we only have one possible pattern length, which does not leak any information, therefore:

$$\mathcal{L}(\ell) = 2 + \mathcal{L}_{patterns}(\ell - 8) \approx 0.29 \cdot \ell - 0.29.$$

A.3 g = 19

Computations are very similar for $g = 19$ except all overflows occur after three iterations once $w = g$, the pattern in our trace looks like $t = 0 \ldots 01bbb$, with $|t| = \lambda > 3$, with probability $p_\lambda = 2^{-\lambda+4} \times 2^{-1}$, leaking $I(p_\lambda) = \lambda - 3$ bits of information. The average information leak per pattern is therefore

$$\mathbb{E}[L] = \sum_{i=4}^{\ell} p_i \times I(p_i) = \sum_{i=4}^{\ell} 2^{-i+3} \times (i-3) \approx 2.$$

Following the same reasoning, we can get the average pattern length $\mathbb{E}[N]$ and deduce the expected leakage for an ℓ-bit exponent:

$$\mathbb{E}[N] = \sum_{i=4}^{\ell} p_i \times i = \sum_{i=4}^{\ell} 2^{-i+3} \times i \approx 5.$$

Hence, the average leakage from the patterns is equal to

$$\mathcal{L}_{patterns}(\ell) = \mathbb{E}[L] \times \frac{\ell}{\mathbb{E}[N]} \approx \frac{2}{5}\ell.$$

Similarly, the average total leakage is:

$$\mathcal{L}(\ell) = 2 + \mathcal{L}_{patterns}(\ell - 5) \approx 0.4 \cdot \ell.$$

B SAMPLE OF AN EXECUTION TRACE

Figure 3 illustrates the execution trace acquired when the following parameters: password `dAdinretAm`, username `admin`, and salt `0x0102030405060708`.

C DIFFERENTIAL SCORE COMPUTATION

In this appendix, we explain how we computed the differential score between the trace of a password candidate, and the trace we actually measured.

Let x be the secret password, and p be a password candidate.

(1) We perform the same modular exponentiation on p, and output an array containing the length of each pattern during the exponentiation: pat_p.
(2) Having pat_p and pat_x, we initialize the score s with the difference in the total number of the observed iterations

$$s = \left| \sum_{i\ in\ pat_p} i - \sum_{j\ in\ pat_x} j \right|$$

(3) For each pattern, starting from the last one, we add to the score the difference between the measured pattern's length, and the candidate's.

$$s+= \sum_{i=1}^{len(pat_x)} |pat_x[-i] - pat_p[-i]|$$

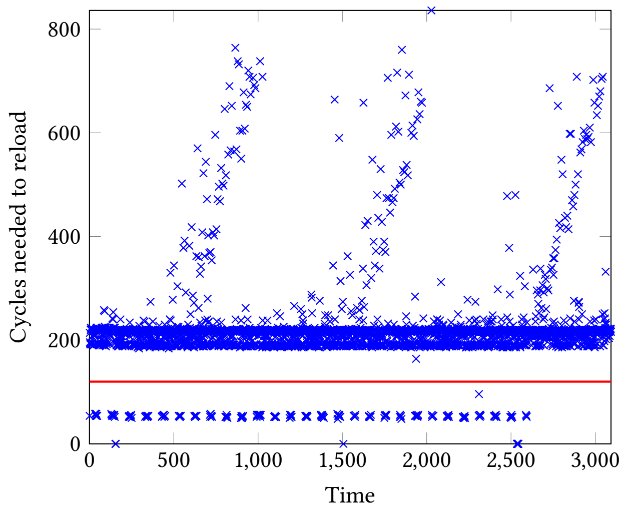

Figure 3: Full execution trace using the password `dAdinretAm`, username `admin` and salt `0102030405060708`. Values below the threshold means the victim executed the monitored instruction.

Thus, any minor issue during the measurement (missing one iteration for instance) has little impact on the overall score, since the comparison is re-synchronized at each pattern. We start at the end to avoid the lack of precision of the first traces, so that a potential missing part at the beginning does not de-synchronize the entire trace comparison.

D STANFORD'S JAVA IMPLEMENTATION

We present a quick overview of the major flaws in Stanford's Java reference implementation of SRP.

First, it is worth noting the implementation is obsolete: some mitigation for existing attacks are missing, indicating we are not dealing with SRP-6a, but a previous version.

- Very basic API, with poor security.
- Use java.math.BigInteger to handle big number (modular exponentiation, ...)
- Not up to date:
  - server sends $B = v + g^b$, in recent version of SRP, $B = kv + g^b$ to prevent an attacker to check two passwords in an online dictionary attack
  - u = 32 MSB of H(B), instead of $u = H(A||B)$, so an attacker recovering v can impersonate any client
  - No verification on A and B values
    - Possibility to fix the key to $K = H(0)$ on the server side (if $A = 0$)
    - u = 32 MSB of H(B), instead of $u = H(A||B)$, so an attacker recovering v can impersonate any client
    - Offline dictionary attack if $B = 0$
- No group verification
  - Possible to use small order groups
  - Possible to perform small subgroups attacks
- The exponent used to compute the DH key shares are only 64 bits long (hardcoded value)
  - Given $A = g^a \bmod p$ (transmitted in clear), attackers are able to recover a in $O(2^{32})$.
  - Knowing a, the only unknown in the client side computation of S is x, which is directly related to the password. Hence, attackers may be able to perform an offline dictionary attack to recover the password, by trying to compute S with a given password, and verifying the confirmation message $M1$ sent by the client.
  - Attack complexity: $O(2^{32} + n \times O(3 * \texttt{SHA1} + 2 * \texttt{mod\_exp})) = O(2^{32})$ with n the number of password in the dictionary. $O(3 * \texttt{SHA1} + 2 * \texttt{mod\_exp})$ is the cost of computing $K = H(S)$ from a password candidate

This implementation should be either completely re-implemented, or deleted, but it should not be presented as a reference since it does not include good security practice, is severely outdated and presents very weak parameters by default.

E IMPACT ANALYSIS ON BIG NUMBERS LIBRARIES AND SRP PROJECTS IN THE WILD

E.1 JavaScript's node-bignum

Among the nodes providing big numbers support, some provide a full big integers implementation in pure JavaScript (bignumber.js or jsbn.js), while others rely on native libraries such as GMP (node-bigint) or OpenSSL (node-bignum). In this paper, we only assess the security of node-bignum.

Although the project node-bignum may not be the most popular one to manipulate big integers in JavaScript (compared to bignumber.js for instance), we were able to find notable SRP projects relying on it. For instance, both Mozilla's node-srp and a more active fork node-srp-typescript leverage node-bignum to provide cryptographic operations on big integers. Thus, these two reference projects are directly vulnerable to our attack.

E.2 Ruby Openssl library

Reviewing the official Github repository of Ruby, we identified that it provides wrappers for some the OpenSSL API, including the big integers operations. We found that these wrappers are used by the sirp project, in such a way that makes vulnerable to our attack. In particular, Ruby does not provide any wrapper for the OpenSSL function that sets the flags to be constant time. Therefore, no SRP project written in Ruby can be plausibly assumed to be secure, provided that the reference OpenSSL package is used.

E.3 Erlang and Elixir

Erlang is mainly built upon the Open Telecom Platform (OTP), which is a large collection of general purposes libraries for Erlang. All the cryptography-related functionality ensured by OTP are implemented through callbacks to OpenSSL. Hence, any projects based on OTP are vulnerable to our attack. Namely, OTP provides support for TLS-SRP, with the same implementation (and the same vulnerability) as OpenSSL. Given the list of the main companies using Erlang (e.g., Whatsapp) [19], the range of the vulnerability may be considerable.

Elixir is built on top of Erlang, and, in particular, its cryptographic operations are provided by Erlang's OTP. Thus, some Elixir projects providing SRP implementations, such as strap and srp-elixir, are vulnerable to our attack.

E.4 Python SRP package

Although Python offers its own cryptographic packages, some projects choose to directly call OpenSSL in order to provide support for big numbers. More notably, the pysrp project implements SRP by leveraging OpenSSL, especially for the big integers operations. This package being the default Python implementation for SRP, it is used in at least 10 other open-source projects[3], accumulating hundreds of stars on Github.

One project of particular interest is xbee-python (the official python library designed to interact with Digi XBee's radio frequency modules) that requires the vulnerable python package to authenticate with XBee devices over Bluetooth Low Energy. These radio devices can be deployed in a wide range of products, for a wider range of applications (from intelligent lighting controls to storage tank monitoring and orbital experiments by the NASA [20]).

[3]https://libraries.io/pypi/srp