Checking Passwords on Leaky Computers: A Side Channel Analysis of Chrome's Password Leak Detect Protocol

Andrew Kwong, *UNC Chapel Hill;* Walter Wang, *University of Michigan;* Jason Kim, *Georgia Tech;* Jonathan Berger, *Bar Ilan University;* Daniel Genkin, *Georgia Tech;* Eyal Ronen, *Tel Aviv University;* Hovav Shacham, *UT Austin;* Riad Wahby, *CMU;* Yuval Yarom, *Ruhr University Bochum*

https://www.usenix.org/conference/usenixsecurity23/presentation/kwong

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

Checking Passwords on Leaky Computers: A Side Channel Analysis of Chrome's Password Leak Detection Protocol

Andrew Kwong*
UNC Chapel Hill
andrew@cs.unc.edu

Walter Wang
University of Michigan
walwan@umich.edu

Jason Kim
Georgia Tech
nosajmik@gatech.edu

Jonathan Berger
Bar Ilan University
jonathann1@walla.com

Daniel Genkin
Georgia Tech
genkin@gatech.edu

Eyal Ronen
Tel Aviv University
eyal.ronen@cs.tau.ac.il

Hovan Shacham
UT Austin
hovav@cs.utexas.edu

Riad Wahby
CMU
rwahby@andrew.cmu.edu

Yuval Yarom†
Ruhr University Bochum
yuval.yarom@rub.de

Abstract

The scale and frequency of password database compromises has led to widespread and persistent credential stuffing attacks, in which attackers attempt to use credentials leaked from one service to compromise accounts with other services. In response, browser vendors have integrated password leakage detection tools, which automatically check the user's credentials against a list of compromised accounts upon each login, warning the user to change their password if a match is found. In particular, Google Chrome uses a centralized leakage detection service designed by Thomas et al. (USENIX Security '19) that aims to both preserve the user's privacy and hide the server's list of compromised credentials.

In this paper, we show that Chrome's implementation of this protocol is vulnerable to several microarchitectural side-channel attacks that violate its security properties. Specifically, we demonstrate attacks against Chrome's use of the memory-hard hash function *scrypt*, its hash-to-elliptic curve function, and its modular inversion algorithm. While prior work discussed the theoretical possibility of side-channel attacks on *scrypt*, we develop new techniques that enable this attack in practice, allowing an attacker to recover the user's password with a single guess when using a dictionary attack. For modular inversion, we present a novel cryptanalysis of the Binary Extended Euclidian Algorithm (BEEA) that extracts its inputs given a single, noisy trace, thereby allowing a malicious server to learn information about a client's password.

1 Introduction

The past few decades have witnessed a drastic increase in the amount of usernames and passwords leaked via various data breaches. This in turn led to an increase of credential stuffing attacks, where attackers try using leaked credentials from one service to breach accounts on other services. Prior works have demonstrated that even post compromise, 6.9% of credentials remain valid due to reuse, often for years [50].

However, the wide availability of datasets of breached credentials also has the potential to enable browsers and password managers to actively alert users when their specific credentials are present in the dataset, protecting their account from the risk of compromise. Indeed, most browsers have launched some type of password alerting service, automatically checking all credentials entered for prior vulnerabilities.

Their inclusion in a browser's default configuration, however, raises significant privacy concerns from users, as passwords have to be shared with a credential checking service. This prompted Google to incorporate a Private Set Intersection (PSI) protocol as part of Chrome's Password Leak Detection mechanisms [50], removing the need to share users' passwords or the server's list of compromised credentials.

Another emerging threat to modern systems is the risk of side-channel attacks. With a plethora of microarchitectural attacks on cryptographic implementations, both from native code [2, 6, 20, 24, 38, 39, 40, 43, 51, 52, 57], and from the browser [3, 26, 28, 29, 37, 48, 53], it is imperative that cryptographic protocols use side-channel hardened implementations when processing sensitive information. With Google's Password Leak Detection protocol [50] using highly customized cryptographic implementations, in this paper we ask:

Is Chrome's Password Leak Detection protocol vulnerable to side-channels? If so, how can attackers exploit the Password Leak Detection protocol to recover the users' passwords?

*Work partially done while affiliated with the University of Michigan
†Work partially done while affiliated with the University of Adelaide

1.1 Our Contribution

We analyze Google's Password Leak Detection protocol, enabled by default in Chrome version 106 (latest at the time of writing), and find that Chrome's Password Leak Detection protocol leaks information via microarchitectural side-channels. In particular, by monitoring the cache access patterns while running this protocol, we are able to reduce the complexity of brute-forcing user credentials such that the attacker often succeeds on the very first login attempt. Since Google's Password Leak Detection protocol is active on default in nearly all modern versions of the Chrome browser, our attack is applicable to nearly every login attempt on the targeted machine.

In our analysis of Chrome's Password Leak Detection service, we found that the client side of the protocol contains three different components that are vulnerable to side-channel attacks, all leaking independently from one another: (i) using *scrypt* on users' input credentials (ii) using *hash2curve* on the output of *scrypt*, and (iii) using BEEA to compute the modular inverse of the value used to blind the output of *hash2curve*. The results of these three attacks are summarized in Table 1.

Attacking Memory Hard Hash Functions. In order to prevent attackers from extracting breached credentials from Google's servers by repeatedly querying the Password Leak Detection protocol, Google requires that all queries hash the checked credentials using a memory hard hash function [44]. This has the effect of slowing down attackers interested in extracting the server's credential list via dictionary attacks, as the computation of the hash digest for each dictionary entry requires a large amount of memory.

Unfortunately, Chrome uses *scrypt* to hash the user's credentials, a design that is inherently non constant time. By observing *scrypt*'s input-dependent memory access patterns using Prime+Probe, in Section 4 we demonstrate how attackers can significantly reduce the cost to brute-force the target's credentials. While prior works have alerted to potential side channel issues with *scrypt*'s design [11, 25], these approaches fail in practice due to limitations in the bandwidth and accuracy of the current state-of-the-art side-channel techniques. As such, we develop novel techniques that account for noisy signals with a highly restricted view of the victim's memory access patterns, resulting in the first end-to-end cache attack against the *scrypt* algorithm.

Attacking Hash2Curve. In addition to memory hard hashing, Google's Password Leak Detection protocol also requires computing a Private Set Intersection (PSI) between the client's credentials and the server's list of compromised accounts. To that end, Google's uses a hash to curve algorithm, converting the output of *scrypt* into points on an elliptic curve. Google then computes the intersection in a homomorphic manner, avoiding the need to share user's passwords or the server's list of compromised credentials.

The *hash2curve* algorithm, however, uses rejection sampling [15], which is non-constant time and is explicitly discouraged [35, Appx. A] due to side-channel concerns. In Section 5 we empirically demonstrate the implications of side-channel leakage due to rejection sampling by presenting attacks on Chrome's hash to curve implementation, using both native and browser-based cache attacks. We also demonstrate that browser-based cache-attacks are still possible on the latest version of Chrome, despite multiple attempts at hardening.

Attacking Modular Inversion. As a final contribution, we analyze Chrome's modular inversion operation used during the blinding of the hash digest of the client's credentials. In Section 6, we attack the Binary Extended Euclidian Algorithm (BEEA) to show how a malicious password server can obtain a digest of the client's credentials using just a single side channel trace. To accomplish this, we developed a novel cryptanalysis of BEEA that allows an attacker to completely recover the inputs, given only a single, noisy trace.

This directly violates the security guarantees of Chrome's Password Leak Detection, which aims to let clients query the server without leaking information on their passwords. Using our techniques, the server can recover a hash of the client's credentials, thereby enabling an offline dictionary attack.

Summary of Contributions: In summary, this work makes the following contributions.

- We present the first side channel analysis of Chrome's Password Leak Detection protocol.
- We empirically demonstrate the first end-to-end cache attack against Chrome's usage of the *scrypt*, allowing us to practically brute force the client's credentials (Section 4).
- We present an attack on Chrome's *hash2curve*, using both native code Flush+Reload and browser-based Prime+Probe (Section 5).
- We present a novel cryptanalysis of the Binary Extended Euclidian Algorithm (BEEA) that recovers inputs using only a single noisy trace, and show that its usage in Chrome leaks information about the client's password, allowing malicious servers to potentially breach the client's credentials. (Section 6).

1.2 Responsible Disclosure

We disclosed the vulnerabilities described in this paper to Google through a Crbug report, and shared our paper with a number of their engineers. We were able to contact the team handling the backend of Chrome's Password Leak Detection service, and suggested and discussed potential mitigations. Google stated that they intend to switch to a variant of Argon2 hash function to mitigate our attack from Section 4, and will use a constant-time *hash2curve* algorithm from [34] to mitigate our attack from Section 5. At the moment, however, Google's protocol remains unchanged, as Google believes that given the cost to mount our dictionary attack and the scope of the threat model, the risk to users is minimal.

However, in response to our paper, Chrome mitigated our attack on BEEA from Section 6 through computing the modular inverse by exponentiating by $p-2$, thereby removing BEEA entirely from Password Leak Detection.

Section	Component	Attacker Capabilities	Average Entropy Reduction (bits)	Required # of Traces
4	Scrypt	Native Code	23.41	5
5	Hash-to-Curve	Browser Code	0.24	5
6	BEEA	Native Code + Blinded Hash	Entire Hash[1]	1

Table 1: Summary of attacker capabilities and results. All three attacks are mutually independent, but can be combined for greater information disclosure. The entropy reduction assumes a dictionary attack from the popular "rockyou.txt" password list, which contains 14,341,564 passwords, or 23.77 bits of entropy. The average is calculated from the experimental outcomes weighted by their probabilities, assuming that the attacker observes the number of login attempts indicated in the final column.

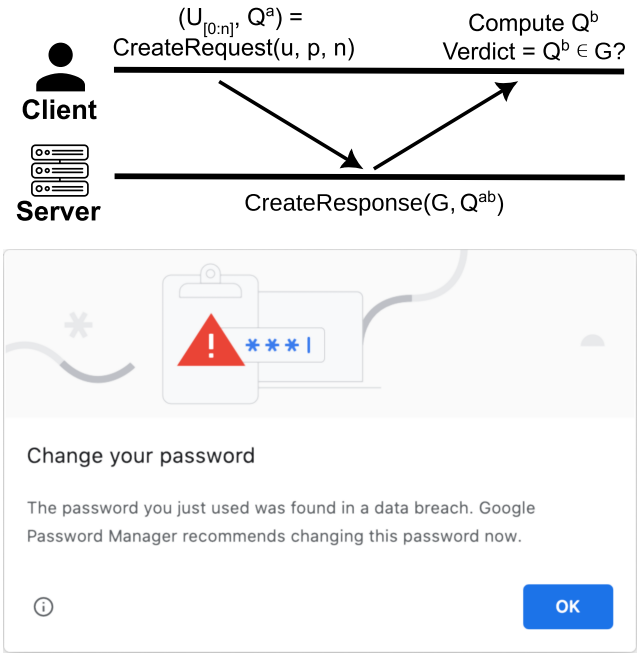

Figure 1. (Top) Overview of steps in Google's Password Leak Detection protocol. (Bottom) User interface displayed by Chrome if the verdict is true.

2 Background

2.1 Chrome's Password Leak Detection

In order to perform privacy-preserving password checking in the Chrome browser, Google uses a custom protocol called Password Leak Detection [50]. More specifically, Chrome's Password Leak Detection combines anonymity sets, memory-hard hashing, and Private Set Intersection (PSI) [36] in order to check if the given username and password pair is present in a data set of compromised credentials, all while preserving privacy against both malicious clients and malicious servers. In this section we describe the Password Leak Detection protocol as implemented in Chrome 106, which differs slighty from the description given in [50].

Protocol Overview. Figure 1 presents a high level overview of Google's Password Leak Detection protocol. The protocol consists of four steps, which we now outline:

CreateDatabase. Before the Password Leak Detection server can handle any requests, it must first hash, blind, and partition its database of leaked credentials. Given the set of credentials, $S = \{(u_0, p_0), (u_1, p_1), ..., (u_\ell, p_\ell)\}$, the server first hashes each username:password pair and partitions the database on the n-bit prefix of the hashes of the usernames by computing:

$$S' = \{(H(u_i)_{[0:n]}, H(u_i, p_i)) \ : \ (u_i, p_i) \in S\}$$

where $H()$ is the same hash2curve algorithm the client used to compute Q. Then, the server generates its own secret key for blinding, b, and blinds the database by computing:

$$S'' = \{(U_{i[0:n]}, H_i^b) \ : \ (U_{i[0:n]}, H_i) \in S'\}.$$

CreateRequest. For each request, the client hashes their username and password pair to an elliptic curve point Q using a hash2curve algorithm, and then blinds Q with a secret key a by computing Q^a. The client then constructs request $= (U'_{[0:n]}, Q^a)$, where $U'_{[0:n]}$ is the prefix of a hash of the username. Finally, the client sends request to Google's Password Leak Detection server.

CreateResponse. The Password Leak Detection server responds to the request by blinding Q^a with b to generate Q^{ab}. The server then computes G, the set of blinded credentials with username hash prefixes equal to the request's as:

$$G = \{H_i^b \ : \ (U_{i[0:n]}, H_i^b) \in S'' \text{ and } U_{i[0:n]} = U'_{[0:n]}\}.$$

The server then returns the tuple response $= (Q^{ab}, G)$.

Verdict. Upon receiving the response, the client uses Diffie-Hellman private set intersection [36] to determine whether its credentials have been leaked. The client calculates the modular inverse of its secret key a, uses it to unblind the doubly blinded hash by computing $(Q^{ab})^{a^{-1}} = Q^b$, and finally checks if $Q^b \in G$, indicating compromise.

Our focus is on analyzing the side channel leakage from the CreateRequest phase; we now describe this phase in greater detail. See [50] for the complete protocol details.

CreateRequest in Detail. Algorithm 1 outlines the CreateRequest phase. The client begins by generating a random nonce a (Line 2), then creates the string s by canonicalizing

1: **function** CREATEREQUEST(u, p, n)
2: $a \leftarrow \mathsf{RAND}()$
3: $u' \leftarrow \mathsf{CANONICALIZE}(u)$
4: $U \leftarrow \mathsf{SHA256}(u')$
5: $H \leftarrow \mathsf{scrypt}(u', p)$
6: $Q \leftarrow \mathit{hash2curve}(H)$
7: $Q^a \leftarrow \mathsf{BLIND}(Q, a)$
8: $U_{[0:n]} \leftarrow \mathsf{BYTESUBSTRING}(U, n)$
9: **return** $(U_{[0:n]}, Q^a)$

Algorithm 1. CreateRequest: The client uses this function to hash and blind the username:password pair to send to the server. The deployed version in Chrome uses this algorithm with the prefix length of the username set to $n = 26$.

the username u and appending the password p to the canonicalized username. More specifically, for a username password pair $(u, p) = (\text{“}\mathit{user@gmail.com}\text{”}, \mathit{secret})$, the canonicalized username and concatenated password is $s = \text{“}\mathit{usersecret}\text{”}$.

The client then passes the canonicalized tuple as input to *scrypt*, a memory-hard hashing algorithm, to compute $H \leftarrow \mathit{scrypt}(s)$. This digest is input to Chrome's *hash2curve* algorithm, producing a point $Q \leftarrow \mathit{hash2curve}(H)$ on the NIST P-256 elliptic curve. Next, the client blinds the hashed point by computing Q^a, where we use multiplication to denote the elliptic curve group operation. Finally, the client computes request $= (\mathsf{SHA256}(u)_{[0:n]}, Q^a)$ where $\mathsf{SHA256}(u)_{[0:n]}$ denotes the n least significant bits of the SHA256 hash of u.

Comparison With the Original Protocol. Thomas et al. [50] evaluated a slightly different version of this protocol, where the client computes request $= (H_{[0:n]}, Q^a)$, with $H_{[0:n]}$ being the digest resulting from computing *scrypt* over both the client's username *and* password; this has the downside that it leaks information about the user's password. To address this, the authors also propose a *Zero-Password Leakage Variant* [50, §3.2] that is similar to Algorithm 1 except that it uses a memory-hard hash function for computing both U and H (lines 4–5, Alg. 1); the authors cite the cost of a second memory-hard hashing step as a disadvantage. Algorithm 1 which is deployed in modern versions of Chrome, *does not* use a memory-hard hash function to compute U, thus avoiding the downside of the original Zero-Password Leakage Variant.

2.2 Cache Attacks

A large body of literature examines how two processes can inadvertently reveal sensitive information to each other due to them sharing the same memory cache [23, 30, 31, 43, 57]. These works show how a *victim process*'s memory accesses influence the state of the cache, and that an attacking process, called the *spy process*, can deduce what the victim accessed by indirectly measuring the state of the cache. The three cache attacks relevant to this paper are described below.

Prime+Probe [43]. This attack only requires that the spy and victim share some level of the cache hierarchy. To carry out this attack, the spy first builds an eviction set, which is a set of addresses that are “congruent" (i.e. mapped to the same cache set) with the targeted cache line.

The attacker then *primes* the cache set by accessing each address in the eviction set, thereby bringing them into the cache. To determine if the victim accessed memory that is congruent with the targeted cache set, the spy process *probes* the cache set by timing accesses to each element in the eviction set; if any access takes longer than an L3 hit, the attacker infers that the victim brought a cache line into the probed cache set and evicted an element in the eviction set, thereby revealing the victim's access to the targeted cache set.

Flush+Reload [57]. This attack has stricter requirements than Prime+Probe, as it needs the spy and the victim to share memory (e.g., the spy and the victim access a de-duplicated library). In this attack, the spy process prepares the cache by flushing the targeted cache-line before the victim performs some action, and then reloads it after the victim finishes. By measuring the reload speed, the spy learns whether the victim brought the targeted cache-line into the cache. Compared to Prime+Probe, Flush+Reload samples more quickly, and works with cache-line, rather than cache-set, granularity.

Flush+Flush [30]. This attack is similar to Flush+Reload, with the difference that instead of reloading the targeted cache-line, the spy simply flushes it again and measures the execution time of the clflush instruction; a longer time indicates that the cache-line was present in the cache. While Flush+Flush samples at a rate nearly 7 times faster [30] than Flush+Reload, it suffers from a lower accuracy [21].

3 Threat Model

In our analysis, we uncovered three separate components within Chrome's Password Leak Detection that leave the client vulnerable to three separate attacks. Each attack reveals information about the client's credentials, and the attacks can all be launched independently from one another. As such, each attack assumes a different threat model, with differing preconditions, and extracts different amounts of information from the victim. See Table 1.

Attack on *scrypt*. Following the standard threat model for microarchitectural attacks, in Section 4 we assume that the attacker can execute native code under the context of an unprivileged user process on the client's machine. Furthermore, we assume that the victim is submitting his credentials while logging into a website on a completely unmodified Chrome browser. The attacker then uses side-channels to extract information on the victim's execution of *scrypt* that will reduce the attacker's search space when conducting a dictionary attack on the victim's input credentials.

Attack on *hash2curve*. For our attack on *hash2curve* (Section 5), we again assume that the victim is submitting his credentials to a completely unmodified Chrome browser. We relax the assumptions on the attacker, however, and only assume that the attacker has Javascript / Web Assembly code running within the victim's browser. This can occur when

the victim navigates to a web page controlled by the attacker. The attacker aims to accomplish the same goal as with the *scrypt* attack: extract information on the victim's execution of *hash2curve* to reduce the search space for a dictionary attack.

Attack on BEEA. The design of Chrome's Password Leak Detection is such that both client and server are mutually untrusting. That is, even the server should not learn anything about the clients' credentials, and Password Leak Detection is designed with a malicious server in mind.

Thus, for our attack on BEEA, we assume that the attacker has access to the blinded output of the *hash2curve*, which is true when the attacker colludes with the server. We also note that this access to the blinded point is safeguarded only by TLS; as such, any attacker that can compromise the connection can also access the blinded point. This could occur via collusion with a TLS middlebox, or even a TLS Enterprise Root CA certificate installed on the victim's machine.

For this attack, in Section 6 we assume the attacker has native, unprivileged code running on the victim's machine, and that he attempts to extract information on the victim's execution of BEEA in order to recover the blinding factor.

4 Attacking Scrypt

In this section, we explore how to leverage a side-channel attack against *scrypt* as used in Chrome to leak information about its inputs. We empirically demonstrate how to use a combination of cache attacks to recover a small subset of the accesses into *scrypt*'s internal memory, which in turn enables an adversary to launch an efficient, offline dictionary attack against the username:password pairs used as input into *scrypt*.

After examining how *scrypt* leaks to an ideal side-channel attacker, we relax the requirements and demonstrate how we performed the attack in practice, with a much weaker attacker.

***scrypt* in Chrome.** Chrome's Password Leak Detection protocol uses *scrypt* in Line 5 of Algorithm 1, when the client hashes the username and password together. In this scenario, *scrypt* serves the function of preventing an adversarial client from efficiently using the Password Leak Detection service to confirm the validity of guessed leaked credentials.

More specifically, the *scrypt* algorithm [45] is a key-derivation function (KDF) that is memory-hard [44]. In contrast to password hashing algorithms that rely on being computationally expensive, *scrypt* was designed to require a large amount of memory, thus making parallelism impractical, assuming it is harder to scale up memory than to scale up computing power. This allows Password Leak Detection to better resist attackers employing ASICs or FPGAs to brute force the server's dataset with massive parallellism.

Achieving Memory Hardness. At a high level, *scrypt* achieves its memory-hard property by requiring input-dependent random accesses into a very large array. While this forces attackers to store large arrays in memory, it also means that *scrypt* is non–constant-time. As noted above, previous works [11, 25] have theorized that cache attacks against *scrypt* are possible due to its inherently non–constant-time nature. Our attack, however, is the first concrete, end-to-end attack on a memory-hard hash function.

[1]This attack recovers all 256 bits of the password hash. In reality, however, the users' passwords have far less than 256 bits of entropy.

1: **function** SCRYPT($P,S,N,r,p,dklen$)
2: $B[0]\,||\,B[1]\,||\,...\,||\,B[p-1] \leftarrow$ PBKDF2$(P,S,1,128*r*p)$
3: **for** $i = 0$ to $p-1$ **do**
4: $B[i] =$ scryptROMix$(r,B[i],N)$
5: $DK \leftarrow$ PBKDF2$(P,B[0]\,||\,B[1]\,||\,...\,||\,B[p-1],1,dkLen)$

Algorithm 2. Scrypt: The function first uses the PBKDF2 key-derivation function to create p blocks each of length $128*r$ bytes. Each block is then transformed by the scryptROMix function. The output is the derived key DK.

4.1 The Scrypt Algorithm

Before describing our attack on *scrypt*, we now outline the relevant portions of the *scrypt* algorithm as it pertains to our attack against Chrome's Password Leak Detection. We refer the reader to RFC7914 [45] for a more complete specification.

Notation. Following the notation of [45], the *scrypt* algorithm takes the following parameters: P, the passphrase to be expanded; S, the salt; N, the CPU and memory cost parameter; r, the block size parameter; p, the parallelization parameter; and *dklen*, the length of the derived key.

Scrypt Overview. Algorithm 2 is an overview of the *scrypt* algorithm. With the exception of P, the password, all parameters, including the salt, are publicly accessible values that are fixed across all users. Thus, the password value P is the only variable input to *scrypt* as used in Password Leak Detection.

In Line 2, B, an array of length p where each element is a block $128*r$ bytes in length is initialized to the output of the PBKDF2 key derivation algorithm. Then, the loop on Line 3 iterates over each block and replaces it with the value obtained by calling the function scryptROMix$(r,B[i],N)$. Finally, the password P and the blocks B, along with the desired output length *dklen*, are passed to PBKDF2 to produce the derived key DK. Most relevant to our attack is Line 4 of Algorithm 2, as *scrypt*'s scryptROMix is highly non constant-time.

scryptROMix. The scryptROMix function (Algorithm 3) is responsible for both *scrypt*'s memory hardness and its non–constant-time-ness. X is first set to the input block B. Then, the For-loop at Line 3 initializes V, an array of size N, by setting $V[i]$ at each iteration to be equal to scryptBlockMix$^i(X)$. We will refer to this first For-loop as the *Initialization Phase*. The scryptBlockMix function takes an input array of a given size, mixes the bytes, and returns an array of equal size.

The second For-loop, beginning at Line 6, iterates N times and makes a non–constant-time, input-dependent memory access (IDMA) each time. We call this second For-loop the *Access Phase*. The IDMA occurs at Line 8, where j was computed by the previous line as $j =$ Integerify$(X) \mod N$. In

```
1:  function scryptROMix(r, B, N)
2:      X ← B
3:      for i = 0 to N − 1 do                          ▷ Initialization Phase
4:          V[i] ← X
5:          X = scryptBlockMix(X)
6:      for i = 0 to N − 1 do                          ▷ Access Phase
7:          j = Integerify(X)  mod N
8:          T = X ⊕ V[j]        ▷ Input-Dependent Memory Access
9:          X = scryptBlockMix(T)
10:     return X
```

Algorithm 3. scryptROMix: The *Initialization Phase* sets each $V[i]$ to $\mathsf{scryptBlockMix}^i(X)$. The *Access Phase* uses j, a function of the input to *scrypt*, as an index into V, thereby making *scrypt* non constant-time.

turn, $\mathsf{Integerify}(X)$ simply returns the final 4 bytes of X, interpreted as a little-endian unsigned integer. Since X comes from the output of PBKDF2 via B, which is dependent upon the input to *scrypt*, j is also a function of *scrypt*'s input. Thus, when j is used as the index into V in Line 8, Algorithm 3 makes an IDMA that is dependent upon the password P, making the entire hashing operation non constant-time.

4.2 Idealized Side Channel Analysis of Scrypt

In this section we perform a side channel analysis of the *scrypt* hashing algorithm assuming an all powerful attacker that can perfectly observe every single memory access into V.

Memory Layout Mapping. *scrypt*'s leakage stems from the IDMA. As the victim executes the For-loop of the *Initialization Phase*, it accesses each memory location in V sequentially (Line 4), with a call to scryptBlockMix in between each access (Line 5). These sequential accesses result in the diagonal line of hits comprising the left part of Figure 2. As the elements of V are accessed sequentially, an attacker that observes the memory accesses in the *Initialization Phase* can learn which elements of V correspond to which memory locations.

Obtaining an Input Dependent Access Pattern. The second half of the accesses in Figure 2 comprises the IDMAs. These accesses are governed by the values of j, which in turn depend on *scrypt*'s secret input P, the password. The attacker learned which memory locations correspond to which indexes and can thus correlate these accesses to the Initialization accesses. This allows the recovery of the values of j at each iteration of the loop in the *Access Phase*. We refer to the sequence of j indexes $(j_0, j_1, \ldots, j_{N-1})$ into V as the *V-Access-Pattern*.

Observing Line 5 of Algorithm 1, the *V-Access-Pattern* is dependent upon the client's canonicalized user name u' and password p. For a specific user name u and password p we denote by $VAP(u,p)$ the access pattern to V resulting from the invocation of Password Leak Detection on u and p.

Leakage Quantification. The precise amount of leakage (in bits) available from $VAP(u,p)$ depends strongly on the concrete parameter choices used by Chrome's Password Leak Detection protocol. First, we note that Chrome sets $N = 4096$

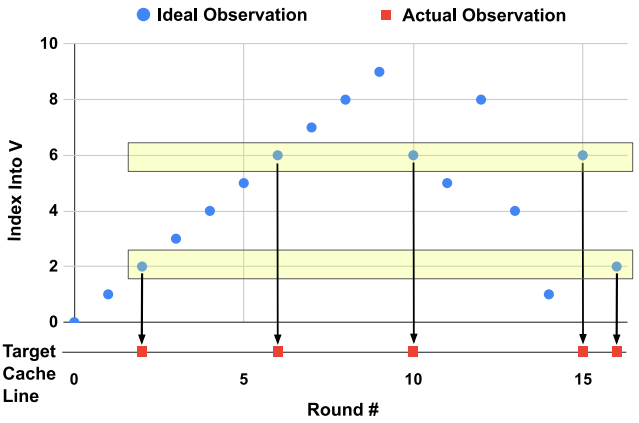

Figure 2. While an idealized attacker can observe each memory access indicated by the blue dots, a realistic attack can only observe memory accesses into a single cache set (yellow boxes) projected onto a single-dimensional trace, represented by the red dots.

when invoking scryptROMix (Algorithm 3). Thus, the leakage available via a perfect observation of $VAP(u,p)$ is theoretically upper bounded by $log_2(4096^{4096}) = 49152$ bits. However, this theoretical limit is further bounded by the size of the input space into *scrypt*'s scryptROMix function. Analyzing the parameter choices for Algorithm 2, we observe that Chrome sets $p = 1$ and $r = 8$. Thus, the call to the PBKDF2 routine in Line 2 of Algorithm 2 produces a total of $128 \cdot 8$ bytes of output, mapping passwords to a digest space of size 2^{8192}. Finally, given that [50] estimate roughly 23.4–31.2 billion unique credential pairs, we expect each username and password pair (u,p) to create its own distinct $VAP(u,p)$.

Credential Recovery via Dictionary Attacks. As each username and password pair (u,p) creates it own distinct $VAP(u,p)$, an attacker can recover u and p from their $VAP(u,p)$ by mounting a dictionary attack. That is, given a plain-text file F of compromised usernames and passwords, an attacker can pre-compute the dictionary

$$DICT(F) := \{(u^*, p^*, VAP(u^*, p^*)) : \forall (u^*, p^*) \in F\}. \quad (1)$$

Next, during the online phase, in case the attacker obtains some *V-Access-pattern* $VAP(u,p)$ corresponding to an attacker-unknown credential (u,p), it is possible to recover (u,p) with high probability by performing a search of $VAP(u,p)$ in F. Finally, we note that this attack violates the requirement that the Password Leak Detection server acts as an inefficient oracle, as only lookup operations over $DICT(F)$ are used during the online phase, while the pre-computation of $DICT(F)$ from a list of compromised credentials F can be done entirely offline via Equation (1).

4.3 The Reality of Cache Attacks

The previous subsection analyzed *scrypt* through the lens of a perfect microarchitectural adversary, capable of completely

reconstructing *scrypt*'s memory access patterns. While similar approaches were proposed in prior works [11, 25], in this section we outline the challenges in empirically realizing this theoretical attack. As we show, these challenges result in an extremely limited view into the *V-Access-pattern* for any given *scrypt* execution, necessitating a different approach. In the following subsections, we outline our solutions and demonstrate the first end-to-end attack on *scrypt*.

Challenge 1: Memory Coverage. The theoretical attack described in Section 4.2 assumes that the attacker can probe all cache sets in between every iteration of both the For-loops in Algorithm 3. This would be required to ensure that no memory access into V is missed, especially during the IDMA where the value of j cannot be predicted by the attacker ahead of time. In practice however, the loop in the *Access Phase* executes in less time than it takes to Prime+Probe a set, so an attacker must somehow slow the victim in order to have a chance to probe even a *single* cache set at each iteration of the loops. From a signal analysis perspective, this results in the attacker's view of Figure 2 being limited to accesses to a single cache set, i.e. the yellow boxes.

Challenge 2: Congruent Cache Lines. There are two yellow boxes, indicating that the attacker views multiple indexes into V, because the array V is large enough to span multiple *congruent cache lines*. Congruent cache lines are lines that map to the same cache set, thereby preventing a Prime+Probe attacker from distinguishing between accesses to addresses that map to congruent cache lines.

As Chrome parameterizes *scrypt* with $N = 4096$, $r = 8$, and $p = 1$, this results in V being an array of 4096 elements with each element 1024 bytes in length, spanning 4MiB total. Next, as the typical L3 cache on Intel machine uses 0.5 MiB ways, we expect that on average any given element of V will be share the same cache set with $(4096 * 1024)/(0.5 * 1024 * 1024) = 8$ other elements. Thus, rather than observing a single hit during *scrypt*'s *Initialization Phase* followed by a single hit during the *Access Phase*, the attacker should expect to see 8 hits during the *Initialization Phase* and an average of 8 corresponding hits during the *Access Phase*.

Empirically demonstrating this issue, we executed an instrumented version of Chrome's *scrypt* code while monitoring the accesses to a single cache set. The top graph in Figure 3 illustrates the access times, where we added the results of 150 *scrypt* traces corresponding to the same cache set. The y-axis measures the number of times a cache-hit was observed, while the x-axis indicates which round it was recorded in. In this example, there are roughly 9 peaks for the *Initialization Phase* followed by another 9 for the *Access Phase*, which is close to what is expected.

Challenge 3: Noise. Further complicating the attack is the presence of noise, which prevents us from perfectly learning which n indices are accessed at which rounds in the *Access Phase*. This can be seen in the Figure 3(bottom), which shows a single trace containing 80 hits, though only 16 are expected.

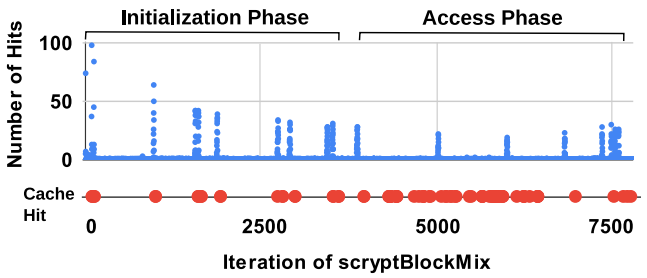

Figure 3. On top is what is obtained from averaging across 150 samples of a custom *scrypt* victim that used the same memory locations for V each time. In reality, we have to use a single-trace, as seen below in red, because Chrome uses different memory locations for V every run.

A natural approach to mitigating this issue is averaging the access time across multiple experiments, as was done in the top graph of Figure 3. However, this is not possible in the case of attacks on unmodified versions of Chrome because Chrome's implementation of *scrypt* allocates a different set of physical memory locations to store V each time. Thus, we developed techniques to overcome the noise and analyze each experiment in isolation, as opposed to combining the results across multiple experiments.

4.4 Attacking Scrypt in Chrome

Having outlined the issues with the theoretical attacks on *scrypt* considered in prior works, we now demonstrate how to attack the *scrypt* implementation used in an unmodified Chrome browser's Password Leak Detection protocol.

Step 1: Observing Control Flow. In order to observe the memory access patterns into the V array during every iteration of both the For-loops in Algorithm 3, we first need to establish a *ticker*, or a marker that indicates the beginning of a new round in either of the For-loops. Since the scryptBlockMix function is called once per iteration, we created our ticker by repeatedly using the Flush+Reload attack on a memory addresses holding the code of this function. We will use this ticker as an indicator of start every iteration of the For-loops in Algorithm 3, allowing us to assign every subsequent side channel observation to its corresponding loop iteration.

Step 2: Performance Degradation. Next, as the iterations of the For-loops in Algorithm 3 are so short in duration, we also had to slow the execution of the scryptBlockMix function using a performance degradation attack [8]. To that aim, we repeatedly used the clflush instruction in order to flush a code region corresponding to the Salsa 20 Core function, which is repeatedly called inside scryptBlockMix.

Step 3: Prime+Probe. With the ticker and performance-degradation established, we choose a random cache set and mount a Prime+Probe attack on it after each occurrence of the ticker. As outlined in Section 4.3, the memory layout of V inside the CPU's cache implies that mounting a Prime+Probe attack on a given cache set discloses when an access is per-

formed to any of the roughly 8 congruent elements of V, without the ability to further distinguish between the elements.

4.5 Handling Noise

As outlined in Section 4.3, a further issue that complicates our attack is the presence of noise, which takes the the form of offsets in the ticker and additions or deletions in the Prime+Probe cache hits. As noted above, we cannot simply average out the noise over multiple traces, since Chrome's *scrypt* implementation ends up using a different set of physical memory locations to store V each time. Thus, rather than combining several measurements into a clean trace, we instead overcome the noise while analyzing each trace in isolation.

Prime+Probe Noise. When we conduct the Prime+Probe attack on a cache set, we see a large amount of false positive noise, where we record cache hits during rounds where there should not be. In addition, we observed a minimal amount of false negative noise, where cache hits are missing.

To overcome this, we implement a scoring system for our dictionary attack, where we assign points to candidate passwords based off how well they "fit" the results from a trace. Given an index j_0 that is accessed during the *Initialization phase*, the attacker can pre-compute at which rounds during the *Access Phase* $V[j]$ will be accessed in case the password candidate is correct. For each of those rounds in the *Access Phase* which the Prime+Probe trace contains a memory access, the candidate password's score is incremented by 1.

We repeat the above approach for each index, $j_0, j_1, ..., j_{n-1}$, for which accesses are detected during *Initialization Phase*, in order to compute the candidate's final score. Finally, after applying this approach on all password candidates in the dictionary, we rank the highest scoring candidates as the most likely passwords.

Ticker Noise. While the above approach is useful for handling noise present in the accesses to V obtained using the Prime+Probe channel, we must adapt this algorithm to also account for noise that is present in our Flush+Reload ticker.

We begin by recalling that we use the ticker to determine at which round the accesses into V are made, during both Initialization and Access phases. Due to both false negatives and false positives, the round corresponding to the memory access is unfortunately rarely correct.

Denoising the Ticker During Initialization. We observe that ticker noise is most damaging during the *Initialization Phase*. If the ticker is off, then our algorithm ends up assigning points for the trace fitting the wrong *V-Access-Pattern*. Fortunately for the attacker, however, the accesses to V during the *Initialization Phase* occur deterministically, with elements of V accessed sequentially (see Line 4 of Algorithm 3). Thus, the attacker learns some of the low bits of the index due to the elements of V being smaller than one page. In particular, each element of V is 1024 bytes, meaning that 4 fit into each page, and for each j_i accessed during the *Initialization Phase*, the attacker learns $j_i (mod\ 4)$. With this optimization, we empirically found it optimal to expanding our scoring algorithm to also consider the indexes that match in the low 2 bits immediately above and below where the hit occurred.

Determining the Transition Point. Next, after denoising the ticker during the *Initialization Phase* of Algorithm 3, we must locate the "halfway point": the point in Line 6 in Algorithm 3, just before the second For-loop starts. This allows us to realign the trace after drifting for 4096 rounds in the *Initialization Phase*. We do so by exploiting the fact that one iteration of the loop in the *Initialization Phase* (Line 3) executes more quickly than one iteration in the *Access Phase* (Line 6) due to the additional code at Lines 7 and 8. Thus, by looking for a point where the time between ticker hits consistently increases, we are able to identify the halfway point, allowing us to identify the ticker's transition to the *Access Phase*.

Denoising the Ticker During Accesses. Ticker noise during the *Access Phase* has a more straightforward solution. When searching for the hits in the *Access Phase* that correspond to the indices found during the *Initialization Phase*, the attacker simply expands her search for any hit within 10 rounds of the expected location. We empirically found 10 to be a good compromise between being too small to overcome the noise, and being so large as to generate too many false positives.

Avoiding Averaging. Overall, we were able to combine the above denoising and scoring techniques into an algorithm to identify candidate passwords based on how well they fit the limited, noisy information the attacker gained on the *V-Access-Pattern*. This improves on prior theoretical attacks on *scrypt* [11, 25], which assumed the attacker has a noiseless and perfect access to the *V-Access-Pattern*.

4.6 Empirical Evaluation

We now evaluate the effectiveness of our attack on Chrome's Password Leak Detection protocol.

Experimental setup. We conducted our attack against an unmodified Chrome binary running on an Acer Aspire E 15 laptop, equipped with 8GiB of DDR4 memory and an Intel i5-8250U CPU. The i5-8250U features 4 cores and 8 threads and is equipped with a 6MiB 12-way set associative L3 cache. Our machine was running Ubuntu 20.04.3 with Linux kernel version 5.8.0-44.

Attack Scenario. Assuming an attacker with native unprivileged code execution on the target machine, we implemented the attack described in this section using the Mastik toolkit [56]. We ran our attacker against an unmodified version of the Chrome browser, and evaluated our attack against 10 randomly chosen passwords from the "rockyou.txt" dictionary. For each password, the victim browser submitted the username:password pair of ("z", *pw*) into a website 5 times, where *pw* was the randomly chosen password from "rockyou.txt". This resulted in our side-channel attacker obtaining 5 traces for each password the victim submitted, where each trace is the result of using side-channels to leak information about the victim's *V-access-pattern*.

A Dictionary Attack. With the collected traces in hand, we applied the approach described in Section 4.5 and conducted an offline dictionary attack. We follow the precedent of previous works and benchmark our attack on the "rockyou.txt" password file, which contains 14,341,564 plaintext passwords stolen during the 2009 RockYou data breach.

We computed the *V-access-pattern* for every entry in the entire "rockyou.txt" file and scored them against the 10 sets of traces corresponding to the 10 passwords the victim submitted. We ran this computation on an Intel Xeon server machine, featuring a Platinum 8352Y CPU, 128 cores, and 1.8TB of memory. It took about 8400 core hours of offline computation to complete the dictionary attack, or about 3 days when parallelized across all 128 cores.

The results of our end-to-end attack are displayed in Figure 4. The correct password ended up scoring higher than all other passwords in the dictionary 80% of the time. Thus, if the attacker were to attempt to log in with the candidate passwords in descending order, the attacker would successfully log into the victim's account on the very first try, 80% of the time. In the worst case, the attacker would succeed on the 6th try. Finally, we note that with such a low number of attempts required for success, no reasonable amount of rate limiting on password attempts can prevent our attacker from compromising the target's account.

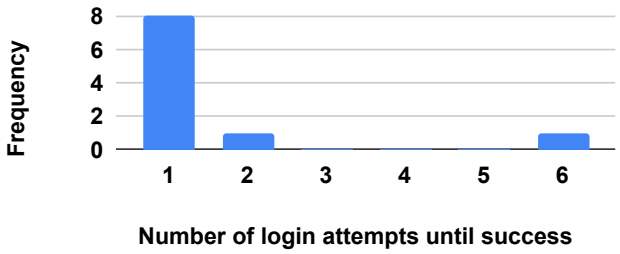

Figure 4. Histogram of the results from our attack on *scrypt* in an unmodified Chrome browser. 80% of the time, the attacker guesses the correct password on the very first attempt.

5 Attacking Hash2Curve

Moving away from attack Password Leak Detection scrypt implementation, in this Section we will examine how Chrome's *hash2curve* usage reveals bits of the user's credentials. Before demonstrating end-to-end attacks from both native code and the Chrome browser on this part of the Password Leak Detection protocol, we now proceed to review Google's *hash2curve* construction and implementation. We used the unmodified Chrome version 106, the latest at the time of writing, for all analyses and experiments in this section.

5.1 Hash2Curve Overview

The *hash2curve* algorithm takes an arbitrary length input string and deterministically outputs a point on a specified elliptic curve. Such a primitive is useful for a number of cryptographic applications; in the case of Password Leak Detection, Chrome's *hash2curve* algorithm serves the purpose of mapping the output of *scrypt* to a point on an elliptic curve to prepare it for use with Diffie-Hellman PSI [36]. This can be seen in Line 6 of Algorithm 1.

```
1: function hash2curve(m)
2:     p_x ← RandomOracleSHA256(m)
3:     while !OnCurve(p_x) do            ▷ Input-Dependent Loop
4:         p_x ← RandomOracleSHA256(p_x)
5:     return (p_x, p_y)
```

Algorithm 4. Google's *hash2curve* implementation.

Chrome's Hash2Curve Implementation. We describe the *hash2curve* algorithm used in Password Leak Detection in Algorithm 4. The variable p_x is first computed by passing the string m as input to the function RandomOracleSHA256, which repeatedly uses SHA-256 and modular addition to approximate a pseudo-random function (PRF). The output p_x is then repeatedly updated by being assigned the value RandomOracleSHA256(p_x) until p_x is a valid x coordinate of a point $P = (p_x, p_y)$ on the NIST P-256 curve (Line 3). The algorithm then outputs $P = (p_x, p_y)$ (Line 5). About half of the possible values of x correspond to points on the curve, so the while-loop in Line 3 of Algorithm 4 terminates with probability $1/2$ in each iteration.

Side Channel Vulnerability of Hash2Curve. While constant-time *hash2curve* algorithms do exist [54], we note that the *hash2curve* algorithm used by Chrome's Password Leak Detection is inherently not constant-time. More specifically, Algorithm 4 uses a rejection sampling method, repeatedly iterating over candidate p_x values until a suitable value is found. While this design pattern was originally proposed by [15], such an implementation is explicitly discouraged by [35, Appendix A] due to side channel considerations.

A Dictionary Attack on Hash2Curve. An attacker can exploit the rejection sampling design of Algorithm 4 for mounting dictionary attacks similar to those mounted in Section 4. Given a credential dataset D, an attacker can apply the *hash2curve* algorithm to every entry of D, obtaining the corresponding number of iterations taken by Line 3 of Algorithm 4, since it is an input-dependent loop (hereinafter IDL). Next, by using a side-channel attack to obtain the number of iterations taken by the IDL on the target's credentials, the attacker can eliminate candidates of D that do not induce the same number of iterations, thereby reducing the attack's search space.

5.2 Native Attack on Hash2Curve

In this section we describe a Flush+Reload based attack on Chrome's *hash2curve* algorithm, executed from unprivileged native code running in the target's machine. Here, we empirically found the strongest results when using Flush+Reload on a string that is touched by Chrome's GetPointByHashing-

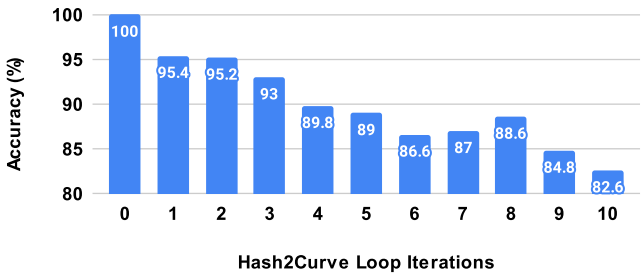

Figure 5. Flush+Reload attack accuracy as a function of the number of *hash2curve* loop iterations.

ToCurveInternal function, which corresponds to the OnCurve test within the IDL.

Attack Outline. To mount a Flush+Reload attack, we execute an unprivileged attacker process monitoring Chrome's GetPointByHashingToCurveInternal function on the target machine. We then open an exemplary login page and complete the login process with a set of user credentials. This triggers Chrome's Password Leak Detection protocol, allowing us to monitor the exact number of iterations made by the IDL.

Attack Evaluation. In order to measure the accuracy of our Flush+Reload attack, we gathered 5500 pairs of username and password, where every 500 pairs generate the same number of iterations of the IDL between 0 and 10 (inclusive). We then executed the attack on each credential pair, noting the amount of detected iterations compared to the ground truth.

Analyzing the results, our Flush+Reload attack was able to correctly identify the number of loop iterations for 4960 credentials, resulting in a total accuracy rate of 90.18%. Next, in Figure 5 we outline the accuracy of our attack as a function of the actual loop iterations performed by the IDL.

Finally, as the IDL exists with probability 1/2 for each loop iteration, we can use the data depicted in Figure 5 to compute our weighted success probability as

$$\mathbb{W}[SR] = 1*1/2+0.954*1/4+\cdots+0.826*1/2^{11}$$

which is roughly equal to 97%.

A Dictionary Attack. By recovering the number of iterations with 97% accuracy, the attacker is able to conduct a dictionary attack and filter out candidate passwords that don't have the same number of iterations. As the IDL exists with probability 1/2 for each loop iteration, the attacker essentially learns an additional bit about the password with each iteration; this means that in the ideal case where the attacker can perfectly recover the number of *hash2curve* loop iterations, the expected number of bits by which the password's entropy is reduced is:

$$\mathbb{E}[B] = 1*\frac{1}{2}+2*\frac{1}{4}+3*\frac{1}{8} = \sum_{i=1}^{\infty} i/2^i = 2.$$

5.3 Attacking Hash2Curve Within Chrome

Having established the vulnerability of Chrome's *hash2curve* algorithm to Flush+Reload attacks from native code, in this section we present a browser-based attack on Chrome's *hash2curve* implementation. More specifically, we attack *hash2curve* from within an unmodified Chrome browser, using an attacker webpage that executes malicious JavaScript and WebAssembly code.

This is a weaker assumption on the attacker's capabiltiies than in Section 5.2, as it is usually easier for an attacker to trick a victim into visiting the attacker's web page. As such, modern browser versions recognize the danger of browser-based side-channels, and employ a heavily sandboxed environment for code execution compared to a native scenario. Accordingly, browser-based adversaries face additional technical challenges, which we now describe.

Flushing Data in the Cache. The Flush+Reload technique used in Section 5.2 requires the clflush instruction and shared memory between attacker and target. However, neither is available in a browser environment.

Instead, we use Prime+Probe to observe the activity of Chrome's *hash2curve* algorithm, using the work of Vila et al. [53] in order to efficiently generate eviction sets.

High-Precision Timing Source. Measuring the cache access patterns of the *hash2curve* algorithm requires the attacker to distinguish cache hits from misses, necessitating a timer with a precision of tens of nanoseconds. However, modern Chrome versions deliberately limit the timer resolution to 5 μs, aiming to foil side channel attacks [55].

We sidestep this issue by using the SharedArrayBuffer JavaScript API, which provides a primitive for a precise counting thread on the order of nanoseconds [47]. While SharedArrayBuffer was previously disabled by Chrome in an attempt to mitigate speculative execution attacks, recent versions of Chrome re-enabled this primitive due to the presence of dedicated Spectre countermeasures [12, 46].

Just-In-Time Compilation. In contrast to attacks that are mounted using native code, which have near-complete control over the attack code executed by the target machine, browser-based adversaries are limited to code emitted by Chrome's JavaScript and WebAssembly execution engines [13, 49]. This introduces measurement noise, making traces obtained through side-channels unreliable and nondeterministic.

To overcome this issue, we observe that it is possible to introduce a warmup stage into our attack code, causing Chrome to always run its optimizing compiler over our high-level Prime+Probe implementation. Furthermore, we initialize our code in a way that Chrome's optimizing compiler will cache its output [17], allowing us to consistently use Prime+Probe across many attack runs. With both measures in place, we achieve a greater probing frequency compared to naive Prime+Probe implementations, allowing us to reliably monitor the execution of the IDL.

Automatically Selecting the Correct Eviction Set. Having generated eviction sets using Vila et al. [53], we must now determine the eviction set corresponding to the IDL. To that aim, our attacker page renders an attacker-controlled login page inside an iframe, populating it with dummy credentials known to the attacker. This triggers the execution of Chrome's Password Leak Detection protocol, eventually resulting in invocations of the IDL.

For each eviction set generated by Vila et al. [53], we execute a Prime+Probe attack on our dummy credentials using the above procedure, locating the eviction set that recorded the correct amount of cache misses corresponding to the execution of the IDL. With the correct eviction set in hand, we can now mount Prime+Probe attacks on the target's credentials.

Attack Evaluation. Our goal is to determine whether browser-based attacks on Chrome's *hash2curve* algorithm are capable of mirroring the performance of native attacks outlined in Section 5.2. With this in mind, we mount a Prime+Probe attack on a targeted username and password pair, using the eviction set found earlier.

Figure 6 presents a time series of Prime+Probe attack iterations; the y-axis plots the number of counting thread ticks elapsed while accessing the eviction set in each iteration. We note the 15 spikes of probes that have at least 500 ticks (in red), which corresponds to 15 iterations of the IDL when Chrome's Password Leak Detection is invoked on our credential. We can distinguish the spikes resulting from the target string being accessed from system noise, since the latter only results in access times no longer than 200 ticks. Finally, Figure 7 shows how many times out of 30 we were able to observe the correct number of spikes when running our attack during 1, 5, and 10 login attempts. While we could not distinguish targeted credentials inducing less than 5 IDL iterations, we observe that 5 logins suffice for the attack to succeed in at least half the trials. This results in our browser-based attack with 5 traces reducing the password's entropy by an expected: $\mathbb{E}[B] = 5 * \frac{19}{30*2^5} + 6 * \frac{19}{30*2^6} + ... + 10 * \frac{20}{30*2^{10}} = 0.24$ bits.

While this may seem like a trivially small amount of leakage in the average case, we contend that only examining the average case and neglecting to account for the danger to passwords with higher numbers of IDL iterations belies the severity of our attack. Instead, we emphasize that our results concretely demonstrate the risk of a significant amount of leakage for a non-trivial number of cases; as shown in Figure 7, when there were 10 IDL iterations and 5 login attempts were observed, the attack succeeded in 20 of 30 trials. This means that for these passwords, the attack reduced the entropy of the password by 10 bits, 2 out of 3 times.

Since 1 out of 1024 credentials result in 10 IDL iterations, and the attack succeeded 2 out 3 times, this means 1 out of 1536 passwords will leak 10 bits. Given the scale at which a browser-based attack can be launched, that fact that our attack can leak 10 bits from 1 in 1536 passwords is concerning, even if the average password leaks very little.

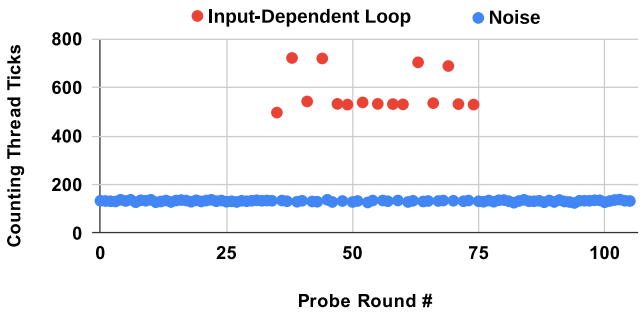

Figure 6. Elapsed counting thread ticks over Prime+Probe iterations, with the 15 spikes corresponding to the 15 IDL iterations highlighted in red.

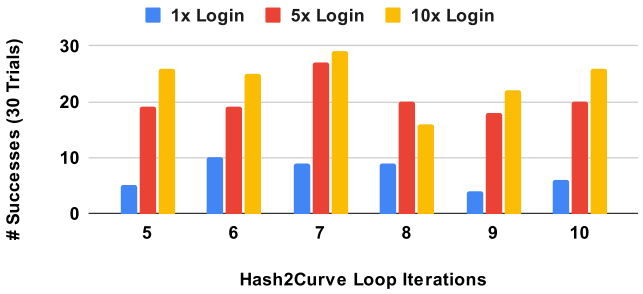

Figure 7. Effect of increasing the login attempts on the number of successes for each number of IDL iterations.

6 Attacking Blinded Hashes

We now examine how Chrome blinds the client's credentials before sending them to the server. In particular, we found that Chrome uses the Binary Extended Euclidian Algorithm (BEEA) to compute the modular inverse of its secret key for blinding, which is susceptible to side-channel attacks.

To investigate the extent to which this compromises Password Leak Detection's security guarantees, we developed a novel cryptanalysis technique that can recover BEEA's inputs after observing only a single, noisy trace obtained via a cache side-channel. We then used our technique to successfully demonstrate the first ever microarchitectural single-trace attack on the BEEA algorithm, improving on prior work that required a controlled execution environment (e.g. SGX [41]) or constraints only present during RSA key generation [7].

Blinding Chrome's Password Leak Detection Protocol. In the Password Leak Detection algorithm, after the client hashes the user's credentials to the point on the curve Q, the client must *blind* the hash before including it in the request to the server. This takes place on Line 7 of Algorithm 1. Specifically, the client blinds the hash by computing Q^a on the elliptic curve, where a is the secret key. This blinding serves the purpose of concealing all bits of the client's hashed credentials from the Password Leak Detection server.

This step is in fact critical, as the Password Leak Detection protocol was designed to preserve privacy in both directions,

meaning that the server should learn nothing about the client's credentials. If the server learns the hash of the client's credentials, the server can then launch a dictionary attack to brute force the client's plaintext password.

Computing Modular Inverses using BEEA. To unblind the response received from the server, the client also needs to compute $a^{-1} \bmod p$, where p is the prime modulus used for the NIST P-256 elliptic curve. The value of $a^{-1} \bmod p$ is then used to unblind the response received from the server in order to complete the Diffie-Hellman PSI [36]. Next, to perform the computation of $a^{-1} \bmod p$, Chrome uses the BEEA algorithm, as implemented by BoringSSL's BN_mod_inverse function, which in turn calls BoringSSL's BN_mod_inverse_odd to take advantage of an optimization for odd moduli.

Threat Model. Similarly to Section 4's threat model, we also assume a side-channel adversary that is able to run unprivileged native code on the victim's machine. Moreover, in this section we assume that the attacker has access to the blinded hash, and wants to obtain the value of the unblinded hash. Access to the blinded hash could occur in practice if the server participating in the Password Leak Detection protocol colludes with the attacker; unblinding the hash would allow the server to brute force the client's credentials via a dictionary attack, a scenario Chrome's Password Leak Detection protocol was specifically designed to protect against.

We note that the attacker can access the blinded hash in other ways, besides colluding with the server; if the attacker could somehow compromise the TLS connection (e.g. through a TLS Enterprise Root CA certificate, or a TLS middlebox), the attacker would gain access to the blinded hash.

6.1 Prior Attacks on the BEEA Algorithm

The BEEA Algorithm has been studied extensively from a side-channel perspective due to the widespread need for computing modular inversions in cryptographic systems (e.g. RSA, ECDSA). While prior side-channel attacks against BEEA are well known, the specific manner in which Google uses BEEA in Chrome's Password Leak Detection protocol prevents the application of prior attack techniques. More specifically, the theoretical analysis of BEEA done in [1] assumes that the attacker can obtain perfect, noiseless traces of the BEEA's execution, which is not possible with current attack techniques. Furthermore, Chrome generates a new random blinding factor a upon each generation of a request, thereby precluding combining information across traces via either averaging or lattice attacks [5, 27].

Thus, in order to attack BEEA as it is used in Chrome, a side-channel attacker must operate with only a single, noisy trace of the BEEA execution. In prior works, this has only ever been accomplished by either placing the victim inside of an SGX enclave [41] or by exploiting the redundancy and relations between various known parameters when BEEA is used during RSA key generation [7]. As neither of these scenarios apply to Chrome's Password Leak Detection, in this section we develop a novel noise-tolerant single-trace attack on the BEEA algorithm that enables extraction of modular inverses computed by BEEA.

```
1:  function BN_MOD_INVERSE_ODD(a, n)
2:      A ← n, B ← a, X ← 1, Y ← 0
3:      while B ≠ 0 do
4:          while even(B) do
5:              B ← B/2
6:              if odd(X) then                    ▷ Branch 1
7:                  X ← X + n                     ▷ Branch 5
8:              X ← X/2
9:          while even(A) do
10:             A ← A/2
11:             if odd(Y) then                    ▷ Branch 2
12:                 Y ← Y + n                     ▷ Branch 5
13:             Y ← Y/2
14:         if B ≥ A then
15:             X ← X + Y                         ▷ Branch 3
16:             B ← B − A
17:         else
18:             Y ← Y + X                         ▷ Branch 4
19:             A ← A − B
20:     return Y
```

Algorithm 5. Binary Extended Euclidian Algorithm: Pseudocode for Chrome's BEEA implementation, which is optimized for the odd modulus n used in NIST P-256. We make the observation that the conditional add, labeled as Branch 5, allows for error correction in a noisy trace.

6.2 Attacking BEEA

At a high level, our attack uses cache-attacks to determine the control flow of the BEEA, which in turn allows for the recovery of the inputs to BEEA. In this case, the inputs to BEEA are a, the client's blinding key, and n, the prime modulus for the elliptic curve NIST P-256.

Chrome's BEEA Algorithm. Algorithm 5 is a pseudocode of the BEEA algorithm. Borrowing notation from [1], we use SHIFTS[i] to denote how many times the branch at Line 4 or Line 9 was taken at the ith iteration of the outer while-loop at Line 3 (only one branch or the other will be taken during any given iteration, as one of A and B will be even and the other odd at the beginning of each iteration). We let SUBS[i] denote the outcome of the comparison at Line 14, such that SUBS[i] $= 3$ if branch 3 is taken, and 4 if if branch 4 is taken at the ith iteration. We note that for any iteration i, if SUBS$[i-1] = 3$, then the SHIFTS$[i]$ must all take place in Branch 1 due to the subtraction at Line 16 in Algorithm 5. Likewise, if SUBS$[i-1] = 4$, then the next iteration's $SHIFTS[i]$ take place in Branch 2.

Perfect Trace Requirement. Acıiçmez et al. [1] previously showed that if an attacker can perfectly recover the SUBS[] and SHIFTS[] for all iterations, they can reconstruct both inputs to BEEA in polynomial time. The downfall of

```
1:  function BRANCH AND PRUNE(Array Trace)
2:      pq ← PriorityQueue()
3:      pq.push(100, 0, Key([]))
4:      while notEmpty(pq) do
5:          (score, i, curKey) ← pq.pop()
6:          if i == len(T) then
7:              Output(curKey)
8:              Continue
9:          curBranch ← Trace[i]
10:         newKey ← curKey.append(curBranch)
11:         if correctXY(newKey) then
12:             if curBranch == 5 then
13:                 score ← score + 20
14:             pq.push(score, i + 1, newKey)
15:         if LastSub(curKey) == 3 then
16:             newKey ← curKey.append(1)
17:             X ← CalculateX(newKey)
18:             if isOdd(X) then
19:                 Continue
20:             pq.push(score − 20, i, newKey)
21:         else
22:             newKey ← curKey.append(2)
23:             Y ← CalculateY(newKey)
24:             if isOdd(Y) then
25:                 Continue
26:             pq.push(score − 20, i, newKey)
```

Algorithm 6. Branch and Prune: Pseudo-code for our branch and prune algorithm that recovers the complete input to BEEA, given a single noisy trace.

this analysis of BEEA is that a single error in either SUBS[] or SHIFTS[] completely foils the recovery of the inputs. Moreover, the attacker cannot determine if there were any errors, and thus whether or not the result is correct.

Prior work [41] that utilized the analysis of [1] was able to extract BEEA's inputs via perfect side channel traces, carefully controlling its execution within an SGX enclave. For attacking Chrome, however, the traces obtained via Flush+Flush contain a substantial number of errors over the course of BEEA's roughly 700 branches on its randomized inputs, preventing us from applying the analysis of [1]. Thus, we overcome the perfect trace requirement by presenting the first analysis of BEEA input recovery for the case of noisy traces.

6.3 Cryptanalysis of a Noisy Trace

To correct the errors present in a noisy BEEA trace, we draw inspiration from prior works on partial key recovery [32, 33] and develop a branch-and-prune style algorithm for BEEA. At a high level, this involves searching for the correct *key*, where a key is a sequence of branches that could have been taken by the execution of BEEA. Our algorithm repeatedly *branches* towards the most probable keys, as determined by how closely they align with the trace. We then exploit the relationship between different segments of the keys to *prune* key candidates, until the correct key is found.

Probing the BEEA Algorithm. While prior works [1, 7, 27, 41] only probed Branches 1, 2, 3, and 4 via cache attacks, we make the observation that detecting the conditional `add` in Branch 5 can be used to prune potential keys during our search algorithm. This is because the values X and Y are completely determined by all the previous branches taken. Furthermore, since X and Y start off initialized to 1 and 0 respectively, and the value n is known to be the prime modulus of NIST P-256, then if we know all the prior branches, we can determine if branch 5 can possibly be taken at the current iteration. We will now cover in detail exactly how our branch-and-prune algorithm works in Algorithm 6.

Algorithm Description. In Line 1, our algorithm takes *Trace*, an array of branches corresponding to the sequence of branches taken by BEEA, as input obtained through a side-channel attack. The branches are simply numbers from 1 through 5, corresponding to the labeled branches in Algorithm 5. We make the assumption that *Trace* only contains *deletion* errors, and only deletions of 1s and 2s occur; furthermore, 1s and 2s are never deleted when they are immediately preceding a 5. In Section 6.4 we explain why this was the case for when we used cache-attacks to obtain traces on BEEA.

The data structure that we use to process our candidate keys is the priority queue on Line 2, which is sorted by the *score* of each candidate key. A key's *score* is a measure of how close to the real key we believe the candidate key to be. In Line 3, we populate *pq* with with a key with score equal to 100, an iterator *i* equal to zero, and an array of branches equal to 0. The iterator is used to track how far along the trace the key has progressed through. The key's array of branches represents the sequence of branches taken by the BEEA algorithm. As the algorithm progresses, we will incrementally build up longer keys that get progressively closer to the real key.

Finding New Keys. At each iteration of the loop on Line 4, we process the highest scoring key, corresponding to the key that we currently believe to be the closest to the true key, and push additional keys onto the priority queue with one additional branch added at a time.

To generate these additional keys, our algorithm branches at each iteration of the outer while loop to create 2 additional keys. The first new key is formed by branching towards the *Trace* by appending the next branch within *Trace* to the current key, as seen on Lines 9 and 10. This candidate key is then pruned on Line 11 if it fails to satisfy *correctXY*().

Pruning on X and Y. The *correctXY*() function inspects the *newKey*'s array of branches. Assuming an execution of BEEA that follows those branches, *correctXY*() then determines if it is possible for all occurrences of Branch 5 to be at the locations that they are. That is, after every occurence of Branch 1 and Branch 2, it checks to see that X or Y, respectively, at that point are odd if and only if a Branch 5 occurs on the subsequent branch. If not, then *correctXY*() returns false, and the key is pruned. On the other hand, if the key is not pruned, then *newKey*'s score is incremented by 20 on Line

13, its position in *Trace* is incremented by 1, and the key is pushed onto *pq* on Line 14.

We note that in actuality, only the most recent branch and the resulting X and Y need to be checked in this manner. Any inconsistencies between X and Y and the sequence of branches earlier in the key would have resulted in that key already having been pruned.

False Positives. While it is possible for $correctXY()$ to return true for keys that do in fact have errors in them, as the distance from the errors grows, the chance of continual false positives is equal to $1/2^n$, where n is the number of 1 branches and 2 branches since the error. This is because X and Y are modified with each occurrence of branches 1 and 2, effectively randomizing whether or not they are odd at any given point. Furthermore, the subtractions in branches 3 and 4 ensure that errors in X propagate to errors in Y, and vice versa. As n grows, the probability of continuing to follow an incorrect key becomes vanishingly small, and the incorrect key values are pruned back to where the error occurred.

Inserting Branches. After branching towards the *Trace*, the second additional key is found by inserting a potential branch into *curKey*. This is how keys that contain the branches deleted by the noisy trace are discovered and added to *pq*. Since the *Trace* obtained in Section 6.4 only contains deletions of 1 Branches and 2 Branches, the $LastSub()$ function at Line 15 only needs to determine whether to insert a 1 Branch or a 2 Branch, which depends on whether the most recent branch is 3 or 4 respectively.

After inserting the potential branch at Line 16, Line 17 uses the function $CalculateX()$ to determine the value of X within BEEA after executing the branches in *newKey*. If this X is odd, this would induce a Branch 5 to follow. However, since there are no deletions of Branch 5 or the immediately preceding branch 1s and 2s in *Trace*, inserting an additional branch 5 automatically renders the key incorrect; as such, we prune the key at Line 19 if X is odd. Otherwise, we push the *newKey* onto the priority queue at Line 20. To prioritize keys that align more closely with *Trace*, we decrement the *newKey*'s score by 20, since for most branches the *Trace* is correct and does not require an insertion. We leave i untouched because we only inserted an additional branch, and did not progress through *Trace*. Lines 22 through 26 serve the same purpose, only for when $LastSub == 4$.

Termination. Once i iterates through the entirety of the *Trace*, the candidate key is output, along with its score. The algorithm can continue to run indefinitely, continuously outputting more complete keys as it explores them. The higher the score of a key, the more likely it is to be the correct one. Intuitively, the true key is the one that aligns most closely with *Trace*, with the insertions in the correct places that result in X and Y being odd whenever dictated by the occurrences of branch 5 in the *Trace*.

6.4 Implementing the Attack

In this section we describe how we implemented our cache attack to obtain a trace against BEEA, and how our branch-and-prune algorithm recovered its inputs.

Software Setup. Chrome is statically linked against BoringSSL, and as such the Password Leak Detection logic calls BoringSSL's BN_mod_inverse function to compute the modular inverse of the blinding factor. To benchmark our attack, we developed a test harness that calls BoringSSL's BN_mod_inverse, compiled with `gcc` version 9.4.0 using an `-O0` flag. Mirroring Chrome, we call BoringSSL's BN_mod_inverse(a, n) with random 256-bit values of a, where n set as NIST P-256's prime modulus.

This is in contrast to the prior two attacks on *scrypt* and *hash2curve*, where we conducted our attacks against unmodified versions of Chrome. We did this because BEEA sees widespread deployment across numerous commonly implemented cryptosystems, and we believe that our novel cryptanalysis has implications on these as well. Thus, there are broader impacts in analyzing how BEEA leaks using our attack technique. By benchmarking our attack against BoringSSL directly, we demonstrate that our attack applies to a wider variety of BEEA usages (any binary that uses BoringSSL's implementation, such as Chrome), and does not rely on nuances specific to Chrome.

Finally, we run experiments on a laptop featuring a Quad Core Intel i5-8250U CPU, and 4 GB of RAM.

Flush+Flush Probing Locations. In order to generate a trace that attempts to reconstruct the control flow of the the victim's BEEA execution, we used the Flush+Flush attack [30] in order to monitor the 5 branches marked in Algorithm 5. We note that doing so requires a total of 4 Flush+Flush probes, as the same probe can be used to monitor both Branch 1 and Branch 2 since these share the same call to BN_rshift1 on Lines 8 and 13. Similarly, a single probe monitors Branch 5 on both Line 7 and Line 12 as they make the same call to BN_uadd. We use the remaining two probes to detect branches 3 and 4, which in turn allows us to discern between branch 1 and branch 2, as described in Section 6.2.

Signal Amplification via Core Assignment. As our i5-8250U processor has 4 physical cores, we run each Flush+Flush probe on a separate core, while having the probe for Branch 5 running on the sibling virtual core to the process executing the BEEA algorithm. As the BEEA process now shares its caches with the probe for branch 5, due to [4] this results in the signal for branch 5 becoming unmistakably strong. This is important, as it virtually eliminates false positives or negatives for branch 5, allowing us to prune keys aggressively by error correcting with the occurrences of Branch 5. Furthermore, this also results in a trace where branch 1s and branch 2s immediately preceding branch 5s are not deleted in the trace, as branch 5 can only ever take place after a 1 or 2.

Gathering Traces. We then allow the BN_mod_inverse function to run while the probing processes monitor the branches.

We parse the resulting data from the Flush+Flush probes form the trace, which is the sequence of branches within BEEA observed by the attacker. We find that this results in no insertions or deletions of branches 3, 4, and 5. However, it is common for there to be insertions or deletions in the number of branch 1s and branch 2s in each round of the while-loop in Line 3 of Algorithm 5. This is because a series of 1 branches or 2 branches can execute in very quick succession when branch 5 is not taken inbetween them.

To make sure that the trace only contains deletions, we calibrate our parser to be extremely conservative with adding 1s and 2s to the trace, only adding them to the trace when the signal is extremely clear. This ensures that only deletions, and not additions, appear in the trace.

Attack Results. After collecting a trace with only deletions in branches 1 and 2, we passed the noisy trace to the branch-and-prune algorithm. Within just 34 ms, the algorithm found the correct key, with 18 branches inserted, and with all 703 branches correctly recovered. This key was also the first one output by the program, and after continuing to run the branch-and-prune program for 10 minutes, this key had the largest score, making it clear that it was the correct key. Having recovered all $SUBS[i]$ and $SHIFTS[i]$, we then used the method described by [1] to trivially recover BEEA's inputs.

Implications for Chrome's Password Leak Detection Protocol. A malicious server that successfully launches this attack can recover the client's secret, a which was used in Line 7 of Algorithm 1 to blind the hash of its credentials as Q^a. After Q^a is sent to the Password Leak Detection server as part of its `request`, the server can easily compute a^{-1} and use it to unblind the client's hash as $(Q^a)^{a^{-1}} = Q$, where Q is the unblinded hash of the client's credentials.

This completely violates the security guarantees of Chrome's Password Leak Detection protocol, as it was designed to allow client's to safely query the Password Leak Detection service without having to place any trust in the server. In this case however, the client is essentially sending a hash digest of their credentials to the server, allowing the server to run offline dictionary attacks using lists of compromised credentials aiming to breach the client's account.

7 Mitigations

The defacto standard for mitigating cache side-channel attacks in software is to make use of the constant-time programming paradigm. In this style of programming, the control flow of the program must not depend in any way upon the program's input; moreover, no accessed memory address can depend upon the input [42]; in other words, the execution of the program must be completely oblivious to its input.

Mitigating against the three vulnerabilities described in this paper, however, is not as easy as simply replacing vulnerable components with constant time implementations. This is because Chrome's usage of *scrypt* as a memory-hard hash function poses a difficult problem, with complex trade offs.

Scrypt. Chrome uses *scrypt* as its hash algorithm for Password Leak Detection due to its memory-hardness properties. A memory-hard hash function is one where the cost of evaluating the hash function is primarily dominated by the cost of memory, as opposed to the cost of compute power. While attackers can employ ASICS and FPGAs to gain a computing advantage of up to 100,000x [16] over general purpose computers, the cost of memory remains the same for both general purpose machines and ASICS/FPGAs. This makes it difficult for attackers to compute the memory-hard hash function at a significantly lower cost than honest users, who must compute the hash with general purpose computers. For this exact reason, *scrypt* is an attractive option for hashing passwords, as [10] proved that *scrypt* is maximally memory-hard under the parallel random oracle model.

This memory-hardness property of *scrypt* comes at a price, however. Namely, [9] show that no function can be both maximally memory-hard and input oblivious; as a consequence, *scrypt* is *inherently* vulnerable to cache side-channel attacks, and in order to mitigate our attack, a compromise is required between input obliviousness and memory-hardness.

As such, we recommend replacing *scrypt* with an alternative option, such as one of the side-channel resistant variants of Argon2 [14], the winner of the 2015 Password Hashing Competition. Argon2i is a variant of Argon2 that is completely constant-time; however, it offers the weakest memory-hardness of the Argon2 variants. This may be unappealing for Password Leak Detection, where resistance against parallel GPU cracking attacks is highly desirable.

A compromising solution is Argon2id, which aims to strike a balance between memory-hardness and side-channel resistance. This is accomplished by making the first pass over the input oblivious, while the second half is input dependent, thereby reducing the amount that can be learned by a side-channel attacker. This, however, means that Argon2id is not completely constant-time, and still leaks some amount of information to side-channel attackers. We caution against permitting any side-channel leakage at all; as we demonstrated with our attack against *scrypt*, even an extremely limited view of the victim's memory accesses can potentially lead to a complete breach of security.

Hash-to-Curve. In contrast, protecting the *hash2curve* portion of Password Leak Detection is comparatively simple; it does, however, require a slight change in protocol, due to the current *hash2curve* algorithm's usage of a rejection sampling method, which is inherently non constant-time. Instead, using one of the constant-time hash-to-curve implementations described in [34] is sufficient to mitigate our attack against the *hash2curve* portion of Password Leak Detection.

Modular Inversion. Similarly, the BEEA algorithm used for modular inversion by Chrome is inherently non constant-time; however, there are known alternatives for modular inversion that are indeed constant-time, and exchanging BEEA for one of these does not require any protocol change.

A potential solution is to make use of Fermat's Little Theorem and to compute the inverse of a as $a^{-1} \equiv a^{p-2} (\mathsf{mod}\ \mathsf{p})$ where the exponentation is performed in constant-time. We can compute this exponentiation both performantly and in constant-time by taking advantage of the fact that the modulus for Curve P-256 is fixed; by pre-computing an optimally short addition chain for the modulus, we can use the addition chain to exponentiate in constant time, with fewer multiplications than other methods [22].

8 Future Work

Other Browsers. Following Chrome's lead, both Microsoft Edge and Mozilla Firefox have implemented their own password leak detection functionality. At the moment, Firefox simply queries the HaveIBeenPwned database; Edge, on the other hand, developed their own novel cryptographic PSI system based off of homomorphic encryption [18, 19]. Investigating their novel cryptosystem's susceptibility to side-channels and other attacks could reveal new insights into how password leak detection systems must consider security and privacy.

Hashing Scheme Tradeoffs. Given our attacks on *scrypt* and Chrome's *hash2curve*, it is natural to wonder if there are any existing hash algorithms would be more suitable with regards to trade offs between performance, memory-hardness, and input-obliviousness. It could be interesting to augment existing hash algorithms, or perhaps even design new ones, that are more desirable for Password Leak Detection.

BEEA Partial Key Recovery. Due to the numerous existing attacks against BEEA, it is easy to imagine how our partial key recovery algorithm can prove useful to that direction of research. However, our key recovery algorithm is specifically tailored to account for the type of noise that we encountered within our traces; it is likely possible to expand upon our algorithms capabilities such that it can handle a broader variety of noisy traces.

Acknowledgments

This research was partially supported by the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425, an ARC Discovery Early Career Researcher Award DE200101577, an ARC Discovery Project number DP210102670, the Defense Advanced Research Projects Agency (DARPA) under Award number HR00112390029, the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972, the National Science Foundation under grant CNS-1954712, and gifts from Cisco, Google, Mozilla, and Qualcomm.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the U.S. Government.

References

[1] O. Acıiçmez, S. Gueron, and J.-P. Seifert, "New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures," in *IMA International Conference on Cryptography and Coding*, 2007, pp. 185–203.

[2] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *CT-RSA*, 2007, pp. 225–242.

[3] A. Agarwal, S. O'Connell, J. Kim, S. Yehezkel, D. Genkin, E. Ronen, and Y. Yarom, "Spook.js: Attacking chrome strict site isolation via speculative execution," in *IEEE SP*, 2022.

[4] A. C. Aldaya and B. B. Brumley, "HyperDegrade: From GHz to MHz effective CPU frequencies," pp. 2801–2818, 2022.

[5] A. C. Aldaya, A. J. C. Sarmiento, and S. Sánchez-Solano, "SPA vulnerabilities of the binary extended Euclidean algorithm," *Journal of Cryptographic Engineering*, vol. 7, no. 4, pp. 273–285, 2017.

[6] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," in *IEEE SP*. IEEE, 2019, pp. 870–887.

[7] A. C. Aldaya, C. P. García, L. M. A. Tapia, and B. B. Brumley, "Cache-timing attacks on RSA key generation," *TCHES*, vol. 2019, no. 4, pp. 213–242, 2019.

[8] T. Allan, B. B. Brumley, K. Falkner, J. Van de Pol, and Y. Yarom, "Amplifying side channels through performance degradation," in *ACSAC*, 2016, pp. 422–435.

[9] J. Alwen and J. Blocki, "Efficiently computing data-independent memory-hard functions," in *CRYPTO*, 2016, pp. 241–271.

[10] J. Alwen, B. Chen, K. Pietrzak, L. Reyzin, and S. Tessaro, "Scrypt is maximally memory-hard," in *EUROCRYPT*, 2017, pp. 33–62.

[11] M. M. Anderson, "Attacking scrypt via cache timing side-channel," https://crypto.stanford.edu/cs359c/17sp/projects/MarkAnderson.pdf, 2017.

[12] J. Archibald and E. Kitamura, "SharedArrayBuffer updates in Android Chrome 88 and desktop Chrome 92," https://developer.chrome.com/blog/enabling-shared-array-buffer/, 2021.

[13] C. Backes, "Liftoff: a new baseline compiler for webassembly in v8," https://v8.dev/blog/liftoff, 2018.

[14] A. Biryukov, D. Dinu, D. Khovratovich, and S. Josefsson, "Argon2 memory-hard function for password hashing and proof-of-work applications," RFC 9106, Sep. 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9106

[15] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the Weil pairing," in *ASIACRYPT*, 2001, pp. 514–532.

[16] D. Boneh, H. Corrigan-Gibbs, and S. Schechter, "Balloon hashing: A memory-hard function providing provable protection against sequential attacks," in *ASIACRYPT*, 2016, pp. 220–248.

[17] B. Budge, "Code caching for WebAssembly developers," https://v8.dev/blog/wasm-code-caching, 2019.

[18] H. Chen, K. Laine, and P. Rindal, "Fast private set intersection from homomorphic encryption," in *CCS*, 2017, pp. 1243–1255. [Online]. Available: https://www.microsoft.com/en-us/research/publication/fast-private-set-intersection-homomorphic-encryption/

[19] H. Chen, Z. Huang, K. Laine, and P. Rindal, "Labeled PSI from fully homomorphic encryption with malicious security," in *CCS*, 2018, pp. 1223–1237. [Online]. Available: https://www.microsoft.com/en-us/research/publication/labeled-psi-from-fully-homomorphic-encryption-with-malicious-security/

[20] S. Cohney, A. Kwong, S. Paz, D. Genkin, N. Heninger, E. Ronen, and Y. Yarom, "Pseudorandom black swans: Cache attacks on CTR_DRBG," in *IEEE SP*, 2020, pp. 1241–1258.

[21] G. Didier and C. Maurice, "Calibration done right: Noiseless Flush+Flush attacks," in *DIMVA*, 2021, pp. 278–298.

[22] Y. Ding, H. Guo, Y. Guan, H. Song, X. Zhang, and J. Liu, "Some new methods to generate short addition chains," *TCHES*, vol. 2023, pp. 270–285, 03 2023.

[23] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+Abort: A timer-free high-precision L3 cache attack using Intel TSX," in *USENIX Security*, 2017, pp. 51–67.

[24] M. Fahr Jr, H. Kippen, A. Kwong, T. Dang, J. Lichtinger, D. Dachman-Soled, D. Genkin, A. Nelson, R. Perlner, A. Yerukhimovich, and D. Apon, "When Frodo flips: End-to-end key recovery on FrodoKEM via Rowhammer," in *CCS*, 2022, pp. 979–993.

[25] C. Forler, S. Lucks, and J. Wenzel, "Memory-demanding password scrambling," in *ASIACRYPT*, 2014, pp. 289–305.

[26] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand pwning unit: Accelerating microarchitectural attacks with the GPU," in *IEEE SP*, 2018, pp. 195–210.

[27] C. P. García and B. B. Brumley, "Constant-time callees with variable-time callers," in *USENIX Security*, 2017, pp. 83–98.

[28] D. Genkin, L. Pachmanov, E. Tromer, and Y. Yarom, "Drive-by key-extraction cache attacks from portable code," in *ACNS*, 2018, pp. 83–102.

[29] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in JavaScript," in *DIMVA*, 2016, pp. 300–321.

[30] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: a fast and stealthy cache attack," in *DIMVA*, 2016, pp. 279–299.

[31] Y. Guo, A. Zigerelli, Y. Zhang, and J. Yang, "Adversarial prefetch: New cross-core cache side channel attacks," in *IEEE SP*, 2022, pp. 1458–1473.

[32] W. Henecka, A. May, and A. Meurer, "Correcting errors in RSA private keys," in *CRYPTO*, 2010, pp. 351–369.

[33] N. Heninger and H. Shacham, "Reconstructing RSA private keys from random key bits," in *CRYPTO*, Aug. 2009, pp. 1–17.

[34] A. Hernandez, S. Scott, N. Sullivan, R. Wahby, and C. A. Wood, "Hashing to elliptic curves," *Internet Engineering Task Force, Internet-Draft draft-irtf-cfrg-hash-to-curve-03*, 2019.

[35] ——, "Hashing to elliptic curves," https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-12, 2022.

[36] B. A. Huberman, M. Franklin, and T. Hogg, "Enhancing privacy and trust in electronic communities," in *ACM conference on Electronic commerce*, 1999, pp. 78–86.

[37] S. Islam, A. Moghimi, I. Bruhns, M. Krebbel, B. Gulmezoglu, T. Eisenbarth, and B. Sunar, "SPOILER: Speculative load hazards boost Rowhammer and cache attacks," in *USENIX Security*, 2019, pp. 621–637.

[38] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *CRYPTO*, 1996, pp. 104–113.

[39] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "RAMBleed: Reading bits in memory without accessing them," in *IEEE SP*, 2020.

[40] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, "Medusa: Microarchitectural data leakage via automated attack synthesis," in *USENIX Security*, 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-medusa

[41] D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar, "CopyCat: Controlled instruction-level attacks on enclaves," in *USENIX Security*, 2020, pp. 469–486.

[42] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *ICISC*, 2006, pp. 156–168.

[43] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *CT-RSA*, 2006.

[44] C. Percival, "Stronger key derivation via sequential memory-hard functions," 2009.

[45] C. "Percival, "The scrypt password-based key derivation function," Internet Requests for Comments, RFC 7914, August 2016. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7914

[46] C. Reis, A. Moshchuk, and N. Oskov, "Site isolation: Process separation for web sites within the browser," in *USENIX Security*, 2019, pp. 1661–1678.

[47] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript," in *FC*, 2017, pp. 247–267.

[48] A. Shusterman, A. Agarwal, S. O'Connell, D. Genkin, Y. Oren, and Y. Yarom, "Prime+Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses," in *USENIX Security*, 2021, pp. 2863–2880.

[49] L. Swirski, "Sparkplug — a non-optimizing javascript compiler," https://v8.dev/blog/sparkplug, 2021.

[50] K. Thomas, J. Pullman, K. Yeo, A. Raghunathan, P. G. Kelley, L. Invernizzi, B. Benko, T. Pietraszek, S. Patel, D. Boneh *et al.*, "Protecting accounts from credential stuffing with password breach alerting," in *USENIX Security*, 2019, pp. 1556–1571.

[51] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom, "SGAxe: How SGX fails in practice," https://sgaxe.com/, 2020.

[52] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "CacheOut: Leaking data on Intel CPUs via cache evictions," in *IEEE SP*, May 2021.

[53] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *IEEE SP*, 2019, pp. 39–54.

[54] R. S. Wahby and D. Boneh, "Fast and simple constant-time hashing to the BLS12-381 elliptic curve," *TCHES*, vol. 2019, no. 4, pp. 154–179, 2019.

[55] Y. Weiss and E. Kitamura, "Aligning timers with cross origin isolation restrictions," https://developer.chrome.com/blog/cross-origin-isolated-hr-timers/, 2021.

[56] Y. Yarom, "Mastik: A micro-architectural side-channel toolkit," 2016.

[57] Y. Yarom and K. Falkner, "Flush+Reload: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security*, 2014.