IACR Communications in Cryptology
ISSN 3006-5496, Vol. 3, No. 2, 23 pages.

https://doi.org/10.62056/a6hee0fgx

Check for updates

BF^2: A Bloom-Filtered Brute-Force Framework for Multi-Target Password Recovery

Cansu Karakuzu Aslan, Wenzel Pünter and Christian Dörr

Hasso Plattner Institute, University of Potsdam, Potsdam, Germany

Abstract. Password-based authentication remains widespread, and large-scale sets of leaked hashes enable practical offline brute-force attacks. Multi-target attacks, which check candidates against large sets of hashes simultaneously, are particularly effective. Understanding the capabilities of low-cost platforms for such attacks is important to assess real-world password security risks.
Therefore, we present BF^2, a modular and scalable FPGA–CPU framework that accelerates multi-target password recovery. BF^2 combines a password-candidate generator, a fully-pipelined NT hash core, a Bloom filter stage to filter non-matching candidates, and a multi-threaded host-side component that performs an exact membership check using a perfect hash function. We implement BF^2 on the low-cost, $199 NiteFury II board. With 16 parallel pipelines running at a 100 MHz clock frequency, our FPGA implementation generates 1.6×10^9 hashes/s. In our experiments, BF^2 demonstrates up to 7.5× higher throughput than John the Ripper, and reduces power consumption by as much as 90% compared to Hashcat on an RTX 5000.

Keywords: Brute-force attacks · FPGA–CPU co-design · Bloom filter · NT hash

1 Introduction

Data breaches expose large-scale sets of password hashes, enabling offline attacks where adversaries test candidates against the database. Unlike single-target attacks that compare each digest against a fixed value, multi-target attacks require set membership testing. In large target sets, set membership testing can become a bottleneck due to limited on-chip memory, irregular off-chip accesses, and bandwidth constraints. As a result, efficient membership testing becomes as critical as hash-computation throughput for multi-target password recovery.

Over the years, advances in specialized hardware such as GPUs, FPGAs, and ASICs have increased brute-force password recovery throughput by exploiting massive parallelism [VHN+11, MDK14, AAA+23], as many password hashing algorithms involve repetitive mathematical operations. GPUs have a very high throughput in password recovery because they are specifically designed for parallel processing. However, they do so at high power and cost, which can be limiting in energy and cost constrained settings. CPU-only tools are accessible and flexible when handling large target sets in main memory, but they provide substantially lower throughput compared to GPU tools. FPGAs offer reprogrammability and are available in a wide range of low-cost, low-power devices. However, for multi-target workloads, large target sets rarely fit in the FPGA's on-chip memory. Therefore, exact matching must be performed off-chip, which quickly becomes bandwidth and latency limited. This creates a gap between hash-computation throughput and the effective end-to-end throughput.

E-mail: cansu.karakuzuaslan@hpi.de (Cansu Karakuzu Aslan), wenzel.puenter@hpi.de (Wenzel Pünter), christian.doerr@hpi.de (Christian Dörr)

This work is licensed under a "CC BY 4.0" license.
Received: 2026-05-04 Accepted: 2026-06-30

To address this gap, we propose BF^2, a hybrid FPGA–CPU brute-force framework for multi-target password recovery. We consider an offline attacker who is given a set of leaked hashes and tests password candidates by hashing them and checking membership against the target set. In this setting, recovering any password from a large target set is often sufficient for an attacker to obtain value from the leak, making multi-target throughput an important security metric. We evaluate this with our hybrid FPGA–CPU architecture. The FPGA utilizes pipelining and parallelization to generate and hash candidates at high rates, while the CPU handles the memory-intensive lookups over the full target set. To minimize off-chip traffic and host-side verification pressure, BF^2 integrates a runtime-programmable Bloom filter directly into the FPGA pipeline as an early-rejection stage and transfers only filter hits to the host for exact verification. By keeping only the compact filter on-chip, BF^2 avoids storing large target sets in FPGA memory and reduces the host transfer and verification workload. Although the false-positive rate of the Bloom filter increases as the target set grows, BF^2 is designed to scale both on the FPGA and CPU sides to mitigate this limitation.

BF^2 targets NT hashes, computed with a single unsalted MD4, due to their continued prevalence in Windows Active Directory deployments. The same approach can be applied to large sets of unsalted, fast-to-compute hashes, such as those based on MD5 or SHA-1. To show that multi-target recovery is practical even on budget hardware, we implement BF^2 on a low-cost FPGA. Both our FPGA implementation and the CPU user application are released as open-source[1].

The main contributions of this paper are as follows:

- We present BF^2, a hybrid FPGA–CPU framework for multi-target NT hash recovery that uses a runtime-programmable Bloom filter on the FPGA for early rejection, while offloading the exact verification to the host CPU.
- We design and implement a modular and scalable FPGA pipeline consisting of a password-candidate generator, a fully-pipelined NT hash core, and a Bloom filter stage, instantiated across multiple parallel pipelines.
- We implement a host runtime that constructs and programs the Bloom filter and performs efficient exact verification of Bloom filter hits using a perfect hash function.
- We demonstrate that this hybrid FPGA–CPU approach provides a low-cost, power-efficient alternative. In our experiments with a $199 NiteFury II board, BF^2 accomplishes as high as 7.5× throughput over John the Ripper (CPU baseline), and consumes up to 90% less power than Hashcat on an RTX 5000 (GPU baseline).

2 Related Work

FPGA-acceleration for high-throughput key search. Early work on hardware-accelerated brute-force search leveraged massively parallel FPGA platforms such as COPACOBANA [KPP+06] and its successors (e.g., enhanced COPACOBANA [GPPS08], RIVYERA [WBB+10]), demonstrating that parallel instantiation of many cryptographic cores can achieve high search rates. In COPACOBANA, authors develop a custom machine that hosts 120 low-cost Spartan-3 1000 FPGAs, capable of processing Data Encryption Standard (DES) keys at an average rate of 48 billion key trials per second, showing that low-cost hardware makes DES vulnerable to brute-force attacks.

A recent work on FPGA-accelerated brute-force search [MBM25] presents a massively parallel exhaustive key-search architecture for A5/3 on an AMD-Xilinx Alveo U250 card. Their architecture operates with 104 cores clocked at 496.7 MHz, and evaluates

[1] https://github.com/cansusu/bf2

approximately 51.72 billion keys per second. Their architecture computes only the first 64 bits of the A5/3 keystream per candidate and compares this partial result against a known value. Keys passing this comparison are sent to host for further evaluation, since partial matches can yield false positives.

FPGA-acceleration for password hashing and recovery. FPGA-based acceleration has also been applied to password hashing schemes, including bcrypt [WZ14], PBKDF2 [AVWS14], WPA2-Personal [KMB+16], MD5 [GPG19], and sha256crypt [ZL20], typically by instantiating many parallel cores and carefully managing data dependencies.

For bcrypt, Wiemer and Zimmermann [WZ14] demonstrate a high-speed search on a low-power Zynq 7020 FPGA, with 40 parallel cores running at 100 MHz. Yet, their architecture is limited to single-target search. The host transfers a target salt and a target hash to the FPGA at startup, and when the hash is found on the FPGA, the attack completes. While this work validates FPGA-based bcrypt acceleration, their single-target focus limits scalability for large-scale multi-target scenarios.

Similarly, for PBKDF2-based attacks, Abbas *et al.* [AVWS14] demonstrate multi-FPGA acceleration in the context of TrueCrypt container recovery using RIVYERA S6-LX150, containing 128 Spartan6-LX150 FPGAs. They show a notable speed-up compared to high-end CPUs and lower energy consumption per password, compared to CPUs and GPUs.

Kammerstetter *et al.* [KMB+16] show an attack on WPA2-Personal with a highly-optimized low-cost FPGA cluster using repurposed cryptomining boards with 36 Spartan-6 FPGAs, where the password is derived through PBKDF2 in combination with SHA-1. They showed that shorter, low-entropy WPA2 passwords can be broken even with low-cost FPGA solutions.

Hybrid CPU–FPGA architectures have also been proposed for password hashing schemes. Zhang and Liu [ZL20] demonstrate a CPU–FPGA hybrid brute-force attack on the sha256crypt algorithm, implemented on a custom-made board. They achieve 2.54× energy efficiency compared with Hashcat running on GTX 1080 Ti GPU. Unlike BF^2, which is designed for multi-target hash recovery, their design focuses on mitigating sha256crypt's data dependencies via CPU-side scheduling while recovering one target hash at a time.

Software baselines and multi-target prefiltering. In practice, password recovery tools that automate the process of testing candidate passwords against captured hash digests are widely used. Two of the most commonly used tools are John the Ripper (JtR) [Ope] and Hashcat [Has]. JtR primarily targets CPUs, though its Jumbo version supports GPU acceleration via OpenCL. Hashcat is designed to utilize GPUs for massively parallel password testing. In multi-target settings, Hashcat uses a bitmap-style prefilter similar to Bloom filters, which generates a bitmap using parts of the hash digests. After the initial filtering, candidates are verified against the hash list to determine a match, using a binary tree.

Tasic *et al.* [TVC24] provide a comparative analysis to demonstrate the performance of Bloom filters in password cracking. Using JtR to generate guess hashes, they compare Bloom filters with other search techniques, such as hash tables, binary search, and Cuckoo filters. Their results show that Bloom filters provide a favorable trade-off between speed, memory consumption, and the probability of false positives. However, their implementation is limited to software-based evaluation and does not address integration into a deeply pipelined FPGA datapath or the resulting trade-offs between host and FPGA at very high candidate rates.

This work. To the best of our knowledge, there is no prior work that integrates a Bloom filter-based early rejection mechanism directly on FPGAs as part of the password recovery

pipeline. BF^2 places this early-rejection stage after a high-throughput NT hash core in the FPGA, while offloading the exact verification to the host. This FPGA–CPU co-design avoids storing large target sets in the FPGA on-chip memory and reduces host transfer and verification workload.

3 Preliminaries

3.1 NT Hash

We focus on the offline recovery of NT hashes [Mic26], a legacy hash scheme that remains prevalent in leaked credential datasets due to its historical use in Windows environments. An NT hash is computed by encoding the password as UTF-16LE and hashing the resulting byte string using MD4.

UTF-16LE. UTF-16LE (UTF-16 Little-Endian) [HY00] encodes characters as 16-bit words in little-endian byte order. For ASCII characters, each code point is represented as two bytes of the form `<byte>00`.

MD4 Message Digest Algorithm. MD4 [Riv91] is a lightweight cryptographic hash function that takes an arbitrary-length input and generates a 128-bit digest. MD4 maintains a 128-bit internal state and consists of three rounds of processing, where each round includes 16 steps. In each step, part of the message is combined with the current state using Boolean functions, modular addition, and a fixed left rotation. This structure makes MD4 very suitable for deep pipelining and parallel replication in hardware.

Although MD4 was considered secure, its security is now severely compromised. Prior work has shown various types of attacks, including collision attacks [Dob96, SNKO07] and preimage attacks [GLRW10, AS09], highlighting the vulnerabilities of the MD4 hash function. These cryptanalytic weaknesses are not used in our attack model, since our focus is on efficient large-scale evaluation and membership testing.

3.2 Bloom Filter

A Bloom filter [Blo70] is a probabilistic data structure to test whether an element is a set member. A standard Bloom filter consists of three main components: an m-bit array (initialized to 0s), k hash functions, and N inserted elements.

To insert an element into the Bloom filter, the element is hashed with the k hash functions, each producing an index into the m-bit array, and the corresponding bits are set to 1. The setting of bits continues until all elements are inserted. An example of a Bloom filter with a set $\{\mathrm{A}, \mathrm{B}\}$ and hash functions $\{h_1, h_2, h_3\}$ is illustrated in Figure 1.

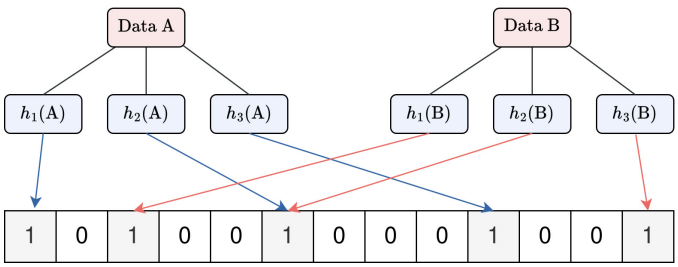


Figure 1: Example Bloom filter with set $\{\mathrm{A}, \mathrm{B}\}$ and hash functions $\{h_1, h_2, h_3\}$.

To test whether an element is in the Bloom filter, the element is hashed with each hash function, and the corresponding bit positions are checked. If all the bit positions are 1, the element might be in the set. Otherwise, the element is guaranteed not to be in the set, meaning there are no false negatives. Algorithm 1 shows the insertion and query operations of the Bloom filter.

Algorithm 1 Bloom Filter Operations

function INSERT(x)
Input: element x, bit array B of length m, hash functions $h_1, \ldots, h_k$
Output: updated bit array B
  for $i = 1 \to k$ **do**
    $p \leftarrow h_i(x) \bmod m$
    $B[p] \leftarrow 1$
  end for
  return B
end function

function QUERY(x)
Input: element x, bit array B of length m, hash functions $h_1, \ldots, h_k$
Output: **True** if x may be in the set, **False** if x is definitely not in the set
  for $i = 1 \to k$ **do**
    $p \leftarrow h_i(x) \bmod m$
    if $B[p] == 0$ **then**
      return False ▷ x not in set
    end if
  end for
  return True ▷ x might be in set
end function

The probability of a false positive in a Bloom filter depends on the size of the bit array m, the number of hash functions k, and the number of inserted elements N. Assuming independent and uniformly distributed hash functions, the false positive probability can be approximated by Equation 1, following the classical analysis of Bloom [Blo70] and Mitzenmacher and Upfal [MU05, §5.5.3].

$$\mathsf{P}_{\text{false}} \approx \left(1 - e^{-kN/m}\right)^k \tag{1}$$

Although Bloom filters are probabilistic and have false positives, they offer a significant space advantage by having a representative array of the elements. A Bloom filter query checks k bit positions, which has a lookup cost of $O(k)$. The lookup cost is independent of the number of inserted elements (and constant for fixed k). These space and time advantages make Bloom filters particularly suitable for applications requiring prefiltering before a more expensive exact verification step.

4 Main Design

4.1 High-Level Architecture of BF²

Brute-force password recovery architectures generate password candidates and compute their hashes, which are then compared against a set of target hashes. Although many accelerators focus on maximizing hash-computation throughput and comparing candidates against a single target, membership testing becomes necessary when there is a large target set. A straightforward method for membership testing is a linear search, which quickly

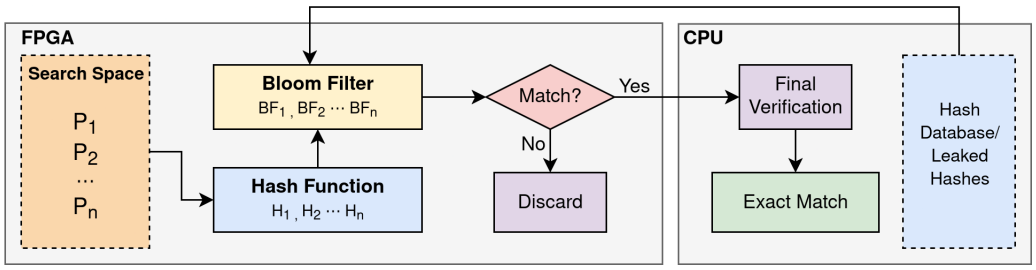


Figure 2: High-level architecture of our proposed design with Bloom filter pre-filtering, targeting multiple hashes.

becomes expensive for a large number of targets. Various other data structures can be used for membership testing, including hash tables and binary search trees, with different memory and performance trade-offs. Probabilistic filters such as Bloom filters and Cuckoo filters can offer space-efficient alternatives at the cost of having false positives. As discussed in Section 2, the evaluation by Tasic *et al.* [TVC24] shows that Bloom filters offer the best trade-off for large hash sets, enabling fast prefiltering with low memory overhead.

To enable high-throughput password recovery, we propose a hybrid FPGA–CPU architecture that performs membership testing against large target sets using a runtime-programmable Bloom filter. Figure 2 shows the high-level architecture of our design. The host generates a Bloom filter from the target set and programs it onto the FPGA. This avoids on-chip storage of the full target set. On the other hand, the FPGA generates password candidates, computes their NT hashes, and performs Bloom filter membership tests. Only the corresponding passwords of Bloom filter hits are buffered in the FPGA's block RAM (BRAM) and transferred to the host, which performs exact verification against the target set.

The remainder of this section describes the FPGA design and its scaling across multiple parallel pipelines, the host-side exact verification, and the PCIe/XDMA integration used to stream Bloom filter hits to the CPU.

4.2 FPGA Design

Our FPGA design is composed of three main modules: a password generator, an NT hash core, and a Bloom filter. We instantiate n parallel pipelines to scale throughput, where n is chosen based on available FPGA resources ($n = 16$ for NiteFury II).

Each parallel pipeline utilizes a password generator module that enumerates candidates in the configured search space. Then, these passwords are passed to our fully pipelined NT hash core, producing 128-bit hash digests. The hash digests are verified against our programmable Bloom filter. The candidates that pass the Bloom filter are queued in the BRAM for transfer to the host for final verification. Figure 3 summarizes the proposed architecture.

Password-Candidate Generation Module. The password-candidate generation module generates a stream of candidate passwords for the NT hash pipeline. Once enabled, it generates one new candidate per clock cycle.

The password generator module is parameterized by the password length (`PWD_SIZE`). Each character is represented as a 7-bit ASCII code, and by default, the character range is set to the printable ASCII range. This can be reconfigured by adjusting the character bounds in the register-transfer level (RTL) code.

The password generation logic utilizes a carry-propagating counter, where the least significant character increments with each cycle and wraps at its upper bound, propagating

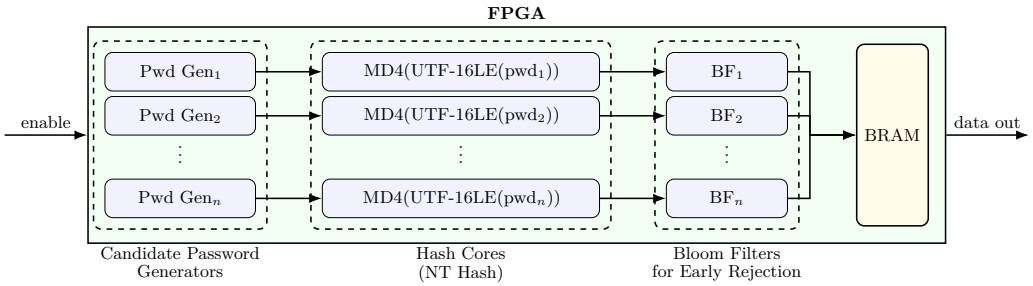


Figure 3: Architecture of our FPGA design with n parallel pipelines. Each pipeline utilizes a password-candidate generator, a hash core, and a Bloom filter (BF).

a carry to the next character. This logic ensures that the entire search space is exhausted. To ensure scalability across n parallel pipelines, each generator core is assigned a unique `core_id` of $\log_2 n$ bits, and uses it to fix a per-core prefix in the generated passwords. In our implementation, this prefix is embedded into the first character's lower $\log_2 n$-bits, while the remaining upper bits continue to increment as part of the carry-propagation counter. This effectively partitions the search space across parallel pipelines with minimal additional control logic. An example of the password scaling strategy when `PWD_SIZE` $= 5$ and $n = 16$ is shown in Figure 4.

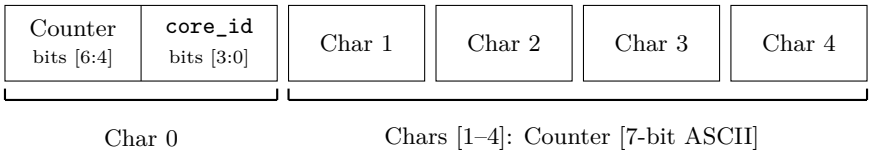


Figure 4: Password structure for `PWD_SIZE = 5`.

The `core_id` partitioning keeps the generator lightweight, but it enumerates Char 0 in n-wide blocks due to the hardcoded `core_id`. For example, for $n = 16$ and a digits-only alphabet `0x30-0x39`, Char 0 corresponds to `0x30-0x3f`, enumerating a superset of the alphabet, while remaining characters enumerate the intended range. Unless otherwise stated, we report throughput in terms of all candidates evaluated by the generator, including any superset values induced by the n-way partitioning of Char 0.

NT Hash Computation Module. As previously defined in Section 3, the NT Hash is computed as $\mathsf{MD4}(\mathsf{UTF\text{-}16LE}(\mathsf{pwd}))$.

To achieve a high-throughput hash generation rate, we implement a fully-pipelined MD4 core. MD4 consists of 48 steps (3 rounds $\times$ 16 steps) operating on four 32-bit state registers (A, B, C, D). We map each MD4 step to a single processing element (PE), allowing the pipeline to accept one input per cycle.

Each PE performs one MD4 step of the form:

$$A \leftarrow \mathsf{ROL}_s\big(A + \Phi(B, C, D) + M_j + K_r\big), \tag{2}$$

followed by the standard MD4 register rotation across (A, B, C, D). Here, M_j denotes the selected 32-bit message word for that step, K_r is the round constant ($K_1 = 0$, $K_2 =$ `0x5A827999`, $K_3 =$ `0x6ED9EBA1`), ROL_s is a fixed left rotation, and Φ is one of the

MD4 Boolean functions f, g, h:

$$f(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge Z), \tag{3}$$
$$g(X,Y,Z) = (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z), \tag{4}$$
$$h(X,Y,Z) = X \oplus Y \oplus Z. \tag{5}$$

We introduce two key optimizations in our design to improve area and timing.

- **Pipelined execution:** Each PE performs the round operation in a single cycle, and generates the updated state registers as the output. These intermediate outputs are registered and passed to the next PE, which allows continuous hashing without any data dependencies, as shown in Figure 5.
- **On-the-fly UTF-16LE and MD4 padding:** Instead of transferring the full 512-bit MD4 message block, we only transfer the $7L$ bits of a length-L password. Each PE computes the required message word M_j (UTF-16LE expansion, `0x80` padding, zeros, and 64-bit length field), on the fly. This reduces pipeline register width and routing pressure, saving $(512 - 7L)$ flip-flops (FF) per PE.

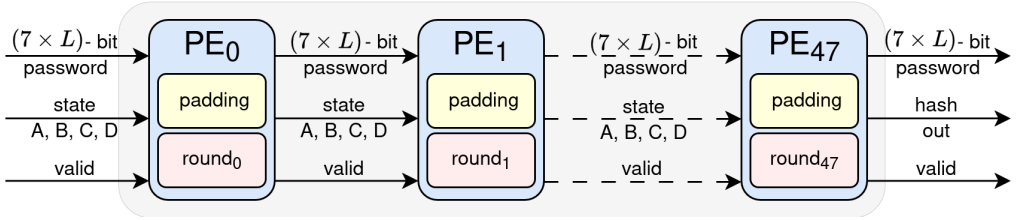


Figure 5: Fully-pipelined MD4 architecture.

Bloom Filter for Early Rejection. Bloom filters are integrated into our pipeline to accelerate the recovery of multi-target workloads. Each pipeline uses its own Bloom filter, which is implemented as a BRAM. A Bloom filter consists of three main components: an m-bit array, k hash functions, and N inserted elements. In standard Bloom filters, the k hash functions are assumed to be independent and uniformly distributed [LGM+19]. However, to reduce logic and lookup latency, we avoid computing k independent hashes in hardware and instead derive k indices from the NT hash, where each index directly addresses one Bloom filter bit. This preserves the Bloom filter mechanism since the NT hash already provides well-distributed bits, which our later evaluation confirms in Subsection 5.5 by varying m and k.

At the beginning of the operation, the host computes the optimal number of indices using Equation 6, where m is the Bloom filter size in bits and N is the number of target hashes.

$$k_{\text{optimal}} = \frac{m}{N} \ln 2 \tag{6}$$

Since our hardware resources are limited, we constrain the number k to the range $[1, 15]$. This value is transferred to the FPGA via AXI GPIO.

To support testing with different target sets, our Bloom filter is designed to be programmable at runtime via the CPU interface. The host generates and streams the filter as 32-bit words. Each word contains a 16-bit address field and a 16-bit data field.

On the FPGA side, the programming logic is as follows:

- **Address change detection:** The programming logic continuously monitors the `bloom_filter_data_i` that is updated through the host CPU. Once the address field (upper 16 bits of the register) changes, the corresponding 16-bit data field (lower 16 bits) is stored in a buffer.

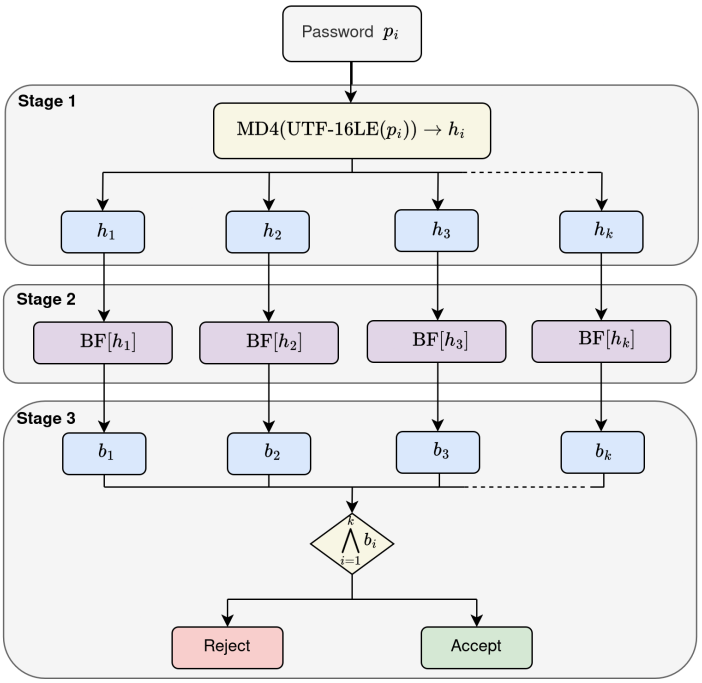


Figure 6: Three-stage Bloom filter pipeline.

- **Data accumulation and broadcast programming:** We use BRAMs with a 64-bit word width to store the Bloom filter. Once a 64-bit word is accumulated from `bloom_filter_data_i`, `prog_active` signal is asserted, and a simultaneous broadcast write happens to each Bloom filter BRAM instance.
- **Programming completion:** When the final address is received (depending on the configured filter size), `prog_done_o` signal is asserted, and the hash pipeline begins its operations.

The entire programming sequence requires `BF_SIZE`/16 GPIO writes. This one-time initialization overhead is negligible compared to the password recovery runtime. Once the Bloom filter is ready, membership testing is performed in three stages, as illustrated in Figure 6 and summarized as follows:

- **Index derivation:** The 128-bit NT hash is used to extract k $\log_2$(`BF_SIZE`)-bit indices.
- **Lookup stage:** Each index is used to read one corresponding bit from the Bloom filter, resulting in a total of k bits.
- **Decision stage:** The k Bloom filter bits are ANDed. If any bit is 0, the candidate is rejected and discarded; otherwise it is considered a potential match.

If a password candidate passes the Bloom filter, it is queued in the BRAM and transferred to the host for further verification. With n parallel pipelines, multiple positives can occur in the same clock cycle. By construction of BF[2], all n pipelines generate identical password characters except for the first character (Char 0), which differs across pipelines with the `core_id`. To sustain throughput while performing only one BRAM write per cycle, we write a single base password containing the trailing characters (Char 1

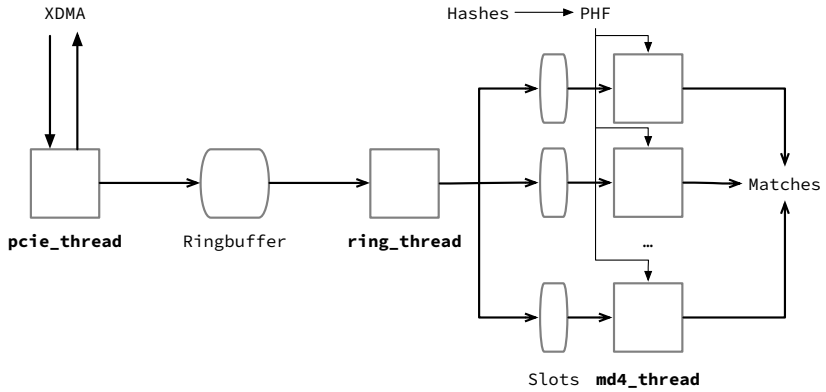


Figure 7: Multi-threaded software verification pipeline overview.

to Char (`PWD_SIZE`−1)), the prefix of Char 0, along with `core_hits`, which is an n-bit mask indicating which pipelines reported a Bloom filter hit. The host reconstructs all hit candidates by inserting the core-specific bits for the first character, according to the set bits in `core_hits`, and then performs exact verification.

4.3 CPU Pipeline

The FPGA platform is controlled by a host computer using its PCIe interface. Hereby, the host system is responsible for configuring the system and verifying the generated hash candidates. This functionality is implemented using a Linux userspace application written in Rust, utilizing APIs provided by the Xilinx XDMA kernel driver.

Configuration. At startup, the program processes the list of target hashes and constructs a Bloom filter with a configurable bit-array size. As hashes already contain sufficient entropy, the Bloom filter is derived from the bits of each hash. Programming k, the Bloom filter, and setting the start bit using memory-mapped registers enable the FPGA pipeline, whose data stream is handled in Runtime mode.

Runtime. During the execution of the brute-force process, the host executes a data processing pipeline to identify true-positive matches in the set of hashes, which is shown in Figure 7. To optimize the throughput of this process, several techniques are used:

- **DMA to ring buffer:** Incoming PCIe beats with the Char 0 prefix, remaining password characters, and `core_hits` are pulled into a large ring buffer using direct memory access, whereby the XDMA driver can map aligned pages to accelerate transfers initiated by the FPGA.
- **Multi-threaded work distribution:** A dedicated thread distributes the received beats to a pool of processing threads, whereby beats are copied to processing slots of each worker using SIMD instructions. Each slot has a depth of 64 elements.
- **SIMD MD4 hash recalculation:** As the stream of FPGA hits contains only probabilistically matching passwords but not the hashes due to the constrained bandwidth of the XDMA protocol, the software recalculates MD4 hashes for all hits. For this purpose, we implement an optimized SIMD version of MD4 supporting 4, 8, or 16 lanes of 32-bit elements. One message block per SIMD lane is processed in parallel. Consequently, MD4 is calculated on a matrix of 16 registers, with each

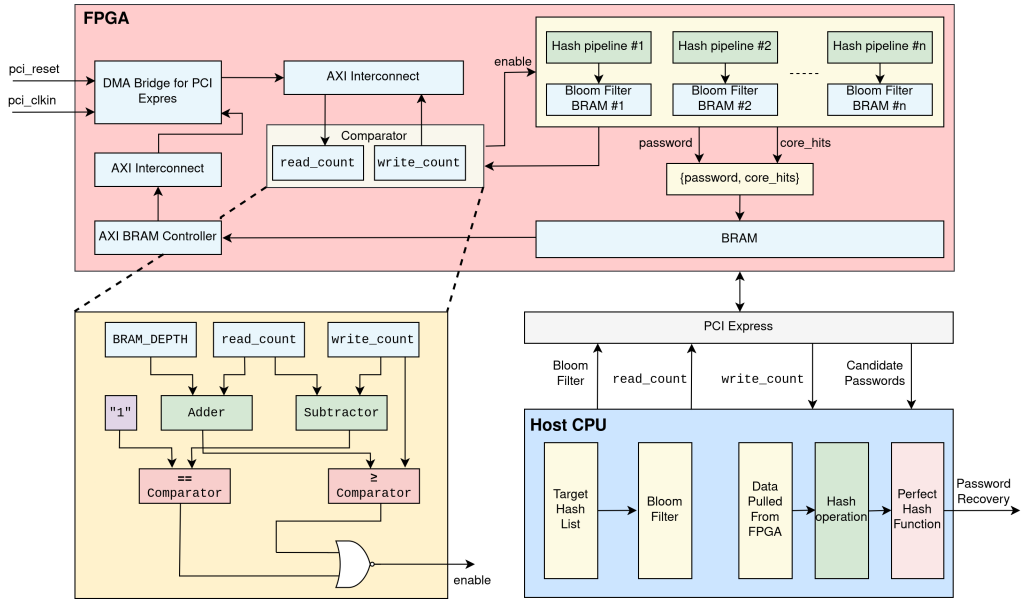


Figure 8: FPGA–CPU Architecture.

register containing 32-bit values of the messages. Each message can have a length of up to 27 characters before padding constraints require additional blocks, which is enforced in software.

Messages are reconstructed from PCIe beats in two steps: First, the SIMD registers are initialized with a template of the padded and UTF-16LE encoded message, which is set through a SIMD broadcasting mechanism. We use the `popcnt` (population count) instruction to efficiently count set bits in `core_hits` to determine the number of required lanes. Afterwards, all the leading bytes are reconstructed from their bit positions and loaded into the first message block register. As soon as a block is full or after all PCIe beats have been received, the MD4 calculation is performed.

- **PHF Lookups.** To verify that an MD4 digest is a true positive, the software implementation uses a perfect hash function (PHF) constructed with the CHD (Compress, Hash, Displace) [BBD09] method to determine its offset in an ordered list of the used hash set. Afterwards, SIMD instructions are used to compare the calculated digest with the indexed candidate. If both elements match, the password processed by the corresponding core has been on the target list and is returned as a true-positive match. For the evaluation scenario, we configure the PHF with 4 keys per bucket, 1 key per bin, and a load factor of 80%. These parameters determine the PHF construction trade-off between space and construction time. The 80% load factor oversizes the target table by 20%, which provides additional space to ease the PHF construction. These parameters affect only the PHF construction and memory usage, not the lookup procedure used during runtime.

4.4 FPGA–CPU Integration

Figure 8 shows in detail how the FPGA and CPU are connected. The host CPU interfaces with XDMA over PCIe and uses two access paths: AXI4-Lite (via AXI GPIO registers) for configuration, and AXI4 memory-mapped transactions to drain the BRAM result queue.

To guarantee the transfer of Bloom filter positives without loss and to prevent overwriting unread BRAM entries, we implement a lightweight backpressure mechanism controlled

by two counters: `write_count` and `read_count`. The FPGA increments `write_count` whenever it appends a new entry to the BRAM queue and exposes its current value to the host via a GPIO register. The host pulls `write_count` to determine the number of available entries, reads them via XDMA, and writes back an updated `read_count` to indicate how many entries have been read. A comparator module in the FPGA then compares the counters and stalls the candidate generation logic when the queue is full.

Bloom filter positives are enqueued into a true dual-port BRAM. The host drains these entries and verifies them through the CPU pipeline, returning only the true positives.

5 Evaluation

We evaluate the end-to-end performance of BF^2 across four aspects: (i) Bloom filter false-positive and queue-pressure trade-offs as the target set scales, (ii) scaling with password length and character set, (iii) comparison against CPU and GPU baselines, and (iv) single-target FPGA version of BF^2 with the membership-testing stages removed.

Threat model. We assume an offline attacker who is given a set of target NT hashes and tests password candidates by computing their hashes and checking membership against the target set. We do not consider online authentication protocols or NTLM challenge–response exchanges.

5.1 Experimental Setup

We implement our design on the NiteFury II [LLC22] board. NiteFury II is a low-cost FPGA development board with a Xilinx FPGA (XC7A200T-2FBG484E), and a PCIe interface. The FPGA runs 16 parallel pipelines at a 100 MHz clock frequency. Synthesis, implementation, and bitstream generation are done on the Xilinx Vivado 2023.1 toolset.

The FPGA board is connected to the host system over a PCIe Gen3 link. The host runs Ubuntu on an Intel Core i7-1255U CPU. On the software side, the host constructs the Bloom filter from the target list, transfers it to the FPGA, reads the resulting candidates, and then recomputes hashes and performs a perfect hash lookup for final matching.

5.2 Baselines and Configurations

We compare BF^2 against Hashcat, a GPU-based brute-force tool, and John the Ripper (JtR, *jumbo* version), a CPU-based brute-force password recovery tool. For the Hashcat experiments, we run Hashcat v6.2.5 on an NVIDIA Quadro RTX 5000 GPU with CUDA 12.2. JtR is evaluated on the host CPU given in Subsection 5.1.

For all baselines, we use mask-based brute-force attacks. A mask specifies both the candidate length and the character alphabet, e.g., printable ASCII characters or lowercase characters. Hashcat is executed in mask mode on a single GPU, while JtR is run in mask mode and parallelized via process forking across 12 CPU threads for higher throughput.

We report password recovery throughput for the software baselines using the candidate testing rate reported by the tools. GPU power for Hashcat is measured at 1 Hz using `nvidia-smi`. CPU power for JtR and for the CPU component of BF^2 is estimated from Intel RAPL by sampling the package energy counter (`energy_uj`) at 1 Hz and converting per-interval energy differences to power. We report the average power over the run.

5.3 Test cases

Varying Bloom filter size and number of target hashes. We evaluate BF^2 with different Bloom filter sizes and with varying numbers of target hashes. As the target set

grows, smaller filters increase the false-positive rate and, therefore, the host verification load and queue pressure. Larger filters reduce this overhead with a lower false-positive rate.

Varying number of Bloom filter indices k. In our design, we do not implement k different hash functions, but use our already generated hash to index the filter. We first show the theoretical false-positive rate of the Bloom filter, and compare the results with BF^2 execution time under $k \in \{1, \ldots, 15\}$.

Single hash verification vs. batch processing. We compare the single-target configuration against the multi-target configuration that uses Bloom filters and host-side exact verification. In the single-target configuration, the host transfers a single target hash to the FPGA via GPIO registers, and the FPGA stops the pipelines when a match is found. This eliminates Bloom filter programming and BRAM buffering overhead.

Password length and character set variations. Lastly, we vary password length (5 to 8 characters) and character sets, including lowercase-only, alphanumeric, and full printable ASCII. Although our experiments use a limited set of examples, both the password length and the character set are configurable in our RTL code.

5.4 Host-side microbenchmarks

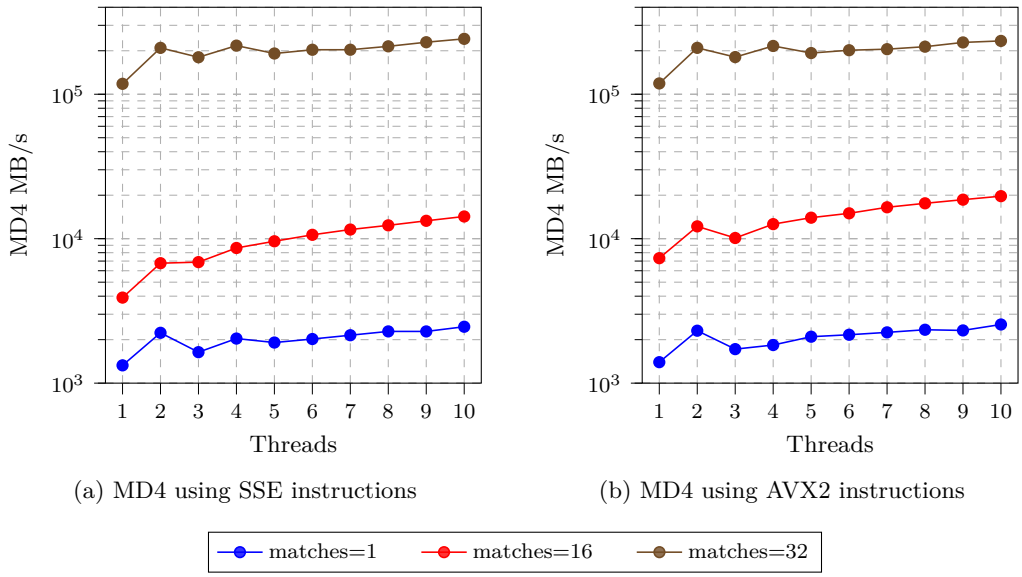
The throughput of the CPU pipeline constrains the throughput of the FPGA pipeline, as backpressure in verifying hashes stalls candidate generation due to limited PCIe queue memory. For this purpose, we evaluate the performance of key software components, including the recomputation of MD4/NT hashes for Bloom filter hits and PHF throughput.

MD4/NT hash throughput. Given that the host system must recalculate the MD4 digest for all password hits of each beat, the majority of the computational effort of the CPU pipeline is spent reconstructing these messages and performing the digest calculation. To estimate the baseline for the experimental set, we measure the median throughput with a 95% confidence interval (CI) depending on the number of computing threads and the number of matches per PCIe beat. In addition, we benchmark the difference between SSE and AVX2 SIMD instructions for the used platform to estimate the influence of 4 vs. 8 message block batches on the actual throughput.

The median throughput of the implementation strongly depends on the number of matches, as shown in Figure 9. With only a single match per beat, we observe a single-threaded throughput of 5.56 GB/s. As the number of threads and matches increases, the throughput also increases significantly, reaching 241.83 GB/s for 10 threads and 32 matches per password, which exceeds the memory bandwidth of the platform. The use of SSE or AVX2 instructions did not have a significant impact on performance.

PHF throughput. We determine throughput baselines for the experimental setup despite the CHD–PHF having an $O(n)$ construction time and $O(1)$ evaluation time, as any modern memory subsystem involves a number of caches and predictors that influence the latency and throughput of DRAM accesses. For this purpose, we measure the median query time (95% CI) for the retrieval of all elements in different set sizes for a PHF with 4 keys per bucket, 1 key per bin, and a load factor of 50%.

The overall query time over all elements is indeed close to linear in the number of elements in the set, as shown in Figure 10. The number of queries per second is, however, strongly affected by the caching hierarchy of the used system and shows degrading performance as soon as data is fetched from main memory. For set sizes up to 10^7 elements

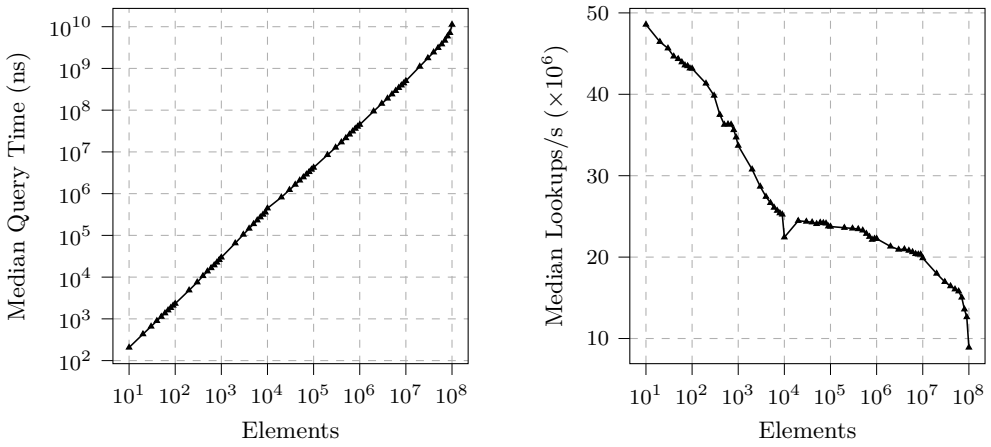


(a) MD4 using SSE instructions

(b) MD4 using AVX2 instructions

Figure 9: Experimental performance of the MD4 SIMD implementation.

on our test system, the single-thread configuration achieves a baseline throughput of approximately 20 million lookups per second.



(a) Median query-All time

(b) Median queries per second

Figure 10: Experimental performance of the CHD–PHF implementation.

5.5 Results

Bloom filter size vs. target-set scaling. We first measure the end-to-end execution time with various Bloom filter sizes and numbers of target hashes. Since the Bloom filter is implemented for each pipeline and uses resources in the FPGA, there is a limit on its size. For our design with NiteFury II as the platform, we are able to implement the Bloom filter up to 32768 bits. For the user password, for testing purposes, we target 5-character passwords with printable ASCII for the character range. The execution time of our system is given in Figure 11 where each curve corresponds to different Bloom filter sizes.

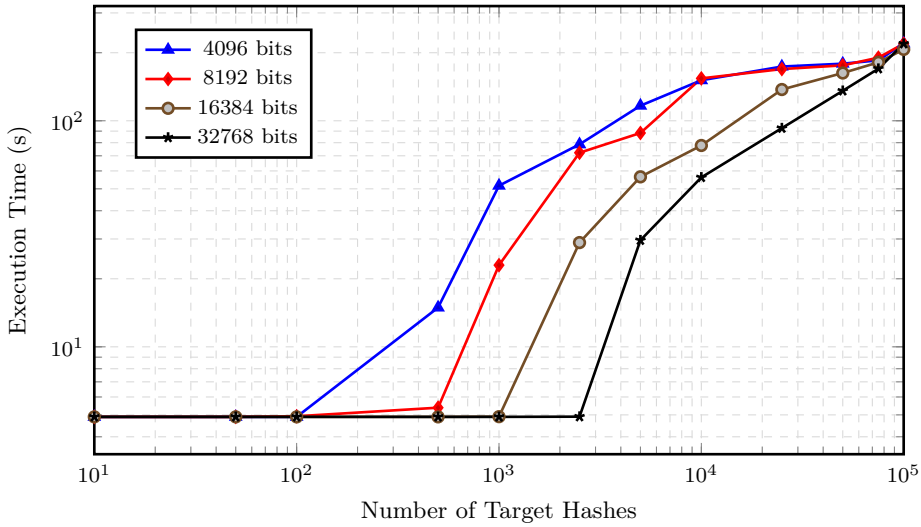


Figure 11: BF^2 execution time as a function of number of target hashes across Bloom filter sizes (`PWD_SIZE = 5`, printable ASCII search space).

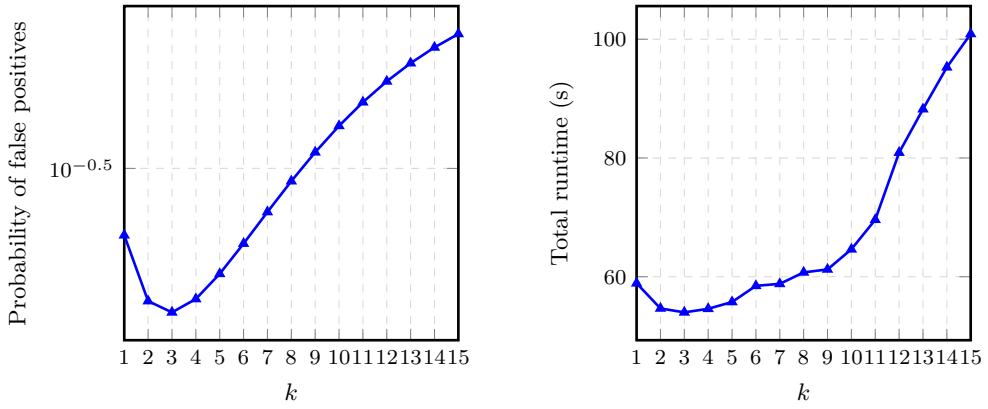
For small datasets, execution times are nearly identical across all filter sizes. As the number of target hashes increases, smaller filters show longer execution times compared to the larger ones. This behaviour matches the expected impact of false positives from the Bloom filter, as there are more hits that fill the queue faster. When there are more elements in the queue, the host CPU has to do more verification, which can stall the FPGA due to the backpressure mechanism. In contrast, larger filters maintain lower backpressure, resulting in faster execution times.

Overall, for a large number of target hashes, larger Bloom filters are more advantageous as they provide faster password recovery with lower false-positive rate.

Number of Bloom filter indices. We next evaluate the effect of the number of Bloom filter indices k. Although BF^2 selects k automatically at runtime, we vary it to validate the expected Bloom filter trade-off and to confirm that our hardware-friendly indexing scheme (deriving k indices from the NT hash instead of k different hash functions) behaves consistently with standard Bloom filter operation. We show the results for $k \in \{1, \ldots, 15\}$, under the configuration: `PWD_SIZE = 5`, printable ASCII, 4096-bit filter, 10^3 targets. Each execution time is the average of five independent runs to minimize the run-to-run variance.

For smaller k, the filter is less selective and produces more false-positives compared to the optimal k (except when $k_{\text{optimal}} = 1$). With a higher number of false-positives, the BRAM queue fills more quickly and introduces additional host-side verification work, resulting in longer execution times. As k increases, the false-positive rate decreases, which also decreases the queue fill rate. This results in fewer stalls on the FPGA side and improves end-to-end execution times. However, beyond a certain point, increasing k gives diminishing results as the filter gets saturated.

We illustrate this behaviour in Figure 12a, where we show the theoretical probability of false positives in the Bloom filter. In Figure 12b, we show the measured execution time of BF^2, under the same varying conditions. The execution time is shortest when $k = 3$, with 54 seconds, matching the expected behaviour of Bloom filters. For smaller values, too few bits are tested, and the filter is not selective enough. For larger k, insertions set many bits per element and saturate the filter, thereby increasing the probability of false positives and resulting in longer execution times.



(a) Theoretical false positive rate vs. k

(b) BF^2 execution time vs. k

Figure 12: Theoretical Bloom filter false-positive rate (a) and measured BF^2 execution time (b) versus k, for 10^3 inserted elements and a 4096-bit filter.

Scaling with password length and character set. We measure the execution time and throughput across varying password lengths and character sets. As the candidate space grows with larger character sets or longer passwords, execution time increases accordingly. To compute end-to-end throughput of BF^2, we added `candidate_counter` to the FPGA, which counts every candidate that is hashed by the NT hash core. Once the program finishes a run, we read the `candidate_counter` over AXI GPIO and divide it by elapsed time to determine the end-to-end throughput.

Given 16 pipelines with a 100 MHz clock, the theoretical peak is 1600 MH/s. This theoretical peak assumes no stalls, initialization, or configuration times. With 10^3 target hashes and a 32768-bit filter, BF^2 achieves an end-to-end throughput very close to the FPGA's theoretical peak. These results are presented in Table 1.

Table 1: Execution time and throughput for different password lengths and character sets (target hashes: 10^3). Throughput counts all evaluated candidates.

Pwd Len.	**Char. Set**	**Time (s)**	**Throughput (MH/s)**
5	printable ASCII	4.90	1594.5
5	alphanumeric	0.79	1593.3
5	lowercase	0.01	1516.3
6	lowercase	0.28	1587.5
7	lowercase	7.62	1595.7
8	lowercase	205.84	1594.9

Comparison against CPU and GPU baselines. We compare BF^2 with the CPU and GPU baselines under the methodology described in Subsection 5.2. We use two different brute-force workloads: printable ASCII with `PWD_SIZE` = 5 and lowercase-only with `PWD_SIZE` = 8. In each workload, we vary the target-set size (10, 100, and 1000 target hashes) while keeping the candidate space fixed. We report throughput, end-to-end execution time, and average power, as summarized in Table 2.

In each workload, BF^2 outperforms JtR (CPU-only baseline). This shows that utilizing the FPGA for hash generation and filtering, while offloading exact verification to the host, increases password-recovery throughput. The average on-chip power of BF^2 is approximately 3 W (Vivado estimate). We report BF^2 average power consumption as the

Table 2: Performance comparison of FPGA, CPU, and GPU solutions.

Char. Set	Tool[a]	Num. of Targets	Time[b] (s)	Throughput (MH/s)	Power[c] (Watt)	Energy Eff. (MH/J)
Printable ASCII (len=5)	BF^2	10	4.9	1595.8	19.7	81.0
		100	4.9	1595.8	19.7	81.0
		1000	4.9	1595.8	19.7	81.0
	JtR	10	25.6	302.0	21.4	14.1
		100	27.1	285.3	20.9	13.7
		1000	36.5	212.0	19.7	10.8
	Hashcat	10	6.0	3020.3	104.6	28.9
		100	7.0	3030.8	104.0	29.1
		1000	7.0	2999.9	102.8	29.2
Lowercase (len=8)	BF^2	10	205.8	1595.4	19.9	80.2
		100	205.6	1596.8	19.8	80.6
		1000	205.8	1594.9	19.9	80.1
	JtR	10	547.3	381.6	20.2	18.9
		100	875.9	238.4	20.1	11.9
		1000	886.2	235.6	20.1	11.7
	Hashcat	10	18.0	15345.6	211.4	72.6
		100	18.0	15217.5	221.0	68.9
		1000	19.0	15109.7	210.4	71.8

[a] BF^2: FPGA+CPU; JtR: CPU-only; Hashcat: GPU-only.
[b] End-to-end execution time.
[c] Average power. For BF^2, power is the total average of FPGA and CPU measurements.

sum of measured average CPU power and the estimated average FPGA power. Compared to JtR, average CPU power decreases by approximately 16–22% (depending on the workload), as the host evaluates only the Bloom filter hit candidates rather than the full candidate space. Even with the FPGA power consumption included, the total average power of BF^2 is comparable to or lower than the CPU-only baseline.

In terms of throughput, BF^2 achieves between 4 and 7.5× higher throughput than JtR across the evaluated experiments. For example, in the lowercase-only workload with 1000 targets, BF^2 reaches 1594.9 MH/s compared to 235.6 MH/s for the CPU-only baseline, corresponding to a 6.8× higher candidate generation rate. Thus, energy efficiency is also improved by a similar factor, achieving around 80 MH/J compared to approximately 12 MH/J for the CPU-only approach.

On the other hand, Hashcat on the RTX 5000 achieves the highest hash-computation throughput, but at a substantially higher power consumption. In the printable ASCII workload, Hashcat reaches 3.0 GH/s while consuming approximately 103 W. BF^2 achieves 1.596 GH/s at 19.7 W, which is about 5× lower power for around half the throughput. The power consumption difference is even bigger in the lowercase-only workload. Hashcat sustains around 215 W, while BF^2 consumes 20 W, which is an approximately 90% decrease in power consumption. In terms of energy efficiency, BF^2 achieves around 80 MH/J, compared to approximately 70 MH/J for Hashcat, making it the most energy-efficient solution among all experiments.

We also report end-to-end execution time (including initialization and configuration overheads). Under this metric, BF^2 achieves the shortest execution time in the printable ASCII workload across all target counts (4.90 s), despite Hashcat having higher hash computation throughput. In the lowercase-only workload, Hashcat remains fastest, while BF^2 significantly reduces power consumption compared to the GPU.

Table 3: Resource utilization, throughput, and power consumption of our FPGA-only implementation, with printable ASCII character set as default.

Pwd	LUT			FF	BRAM	Power	Throughput
Len	Logic LUT	LUTRAM	SRL			(Watt)	(MH/s)
5	66799	2145	1264	121551	28.5	1.8	1600
6	68370	2145	1505	122456	28.5	1.9	1600
7	67731	2145	1809	123246	28.5	2.9	1600
8	69016	2145	2113	124036	28.5	3.8	1600

Overall, we can see that Hashcat on the GPU is optimized for peak hash rate, at the cost of high power consumption. The CPU-only baseline offers lower power consumption but also lower throughput. In contrast, BF^2 targets energy-efficient password recovery for multi-target workloads, and provides higher throughput compared to the CPU-only baseline.

FPGA-only single-target mode. We also implement an FPGA-only password recovery mode that operates without the CPU-side exact verification. To do this, we simply remove the Bloom filter component from the design and add logic to stop the evaluation once the target hash is found. In this mode, the user can directly send a target hash to the FPGA through the AXI GPIO registers. Once a matching password is found, the FPGA writes the result back to the GPIO registers, which can then be read back by the user. Throughput is only determined by the FPGA clock frequency and the number of parallel pipelines. With 16 pipelines operating at 100 MHz, the design sustains a constant throughput of 1600 MH/s.

Since our password generation module generates the candidates deterministically and in a fixed order across n parallel pipelines, the time to recover a given password is directly proportional to its position in the search order.

With this deterministic candidate generation, we can calculate the worst-case recovery time and average recovery time by a simple formula. As each pipeline generates a candidate per clock cycle (after the initial hash pipeline filling stage around 48 cycles, which is negligible compared to the operation), the throughput is $n \times f_{\text{clk}}$, where f_{clk} is the FPGA clock frequency. Let s denote the total number of password candidates in the search space. The worst-case recovery time corresponds to enumerating the entire search space, while the average recovery time (assuming uniformly distributed target hashes) is half of the worst-case, as follows:

$$T_{\text{worst}} = \frac{s}{n\, f_{\text{clk}}}, \qquad T_{\text{avg}} \approx \frac{s}{2\, n\, f_{\text{clk}}}. \tag{7}$$

The number of parallel pipelines, n, can be scaled depending on the available FPGA resources, thereby reducing the password recovery time.

For our FPGA-only configuration, the additional hardware resources to support longer passwords are low, as given in Table 3. LUT and FF utilization increase slightly with longer passwords, while BRAM usage is the same across `PWD_SIZE = [5-8]`. Power consumption increases depending on the password length, but stays under 4 W for the evaluated configurations. For 8-character passwords, the power consumption is reduced by approximately 98% relative to the Hashcat GPU baseline.

Overall, this FPGA-only configuration is beneficial when there is a single target or a few targets, where maintaining a large target set is unnecessary.

Module-level FPGA resource utilization and scalability. As described in Section 4, each parallel pipeline in our design comprises two main blocks: a hash generation pipeline and a Bloom filter stage that enables early filtering of non-matching candidates. We

Table 4: Resource utilization of our FPGA modules. (`PWD_SIZE = 5`, printable ASCII)

Module	LUT		FF	BRAM
	Logic LUT	SRL		
multi_cores_top	56097	1287	112546	240
hash_pipeline (×16)	3152–3233	66–117	6365–6512	0
pwd_gen	83–95	0	67	0
md4_top	3034–3083	66–97	6186–6246	0
md4_PE (×48)	35–76	0–32	40–161	0
bloom_filter (×16)	259–287	0–15	278–299	15

summarize resource utilization per module in Table 4. We can see that most of the LUT and FF utilization is due to our fully-pipelined MD4 calculation. On the other hand, BRAM usage is dominated by the Bloom filter, since each parallel pipeline instantiates its own Bloom filter to support high-throughput membership checks.

Bloom filter BRAM consumption remains constant across the BF sizes that we evaluate, while LUT and FF utilization changes only slightly due to address and control logic differences.

Limitations. Since we use a fixed-size Bloom filter with an increasing number of target hashes, the filter gradually starts to saturate. Beyond a certain threshold, the probability of false-positives approaches 1, causing the host to recompute all generated hashes. In this case, the Bloom filter no longer provides effective filtering, and BF^2 performance starts to drop. This behaviour can also be seen in Figure 11 where the execution time gets longer for a large number of target hashes.

Scalability. To mitigate Bloom filter saturation, BF^2 is specifically designed to scale on both hardware and software sides. On the FPGA, the number of parallel pipelines n and the Bloom filter size m are configurable. Therefore, implementing BF^2 on larger FPGAs with additional resources can sustain a lower false-positive rate by using larger Bloom filters. On the host, the multi-threaded verification pipeline can be scaled with the number of CPU cores and supports wider SIMD instructions (SSE, AVX2). This allows BF^2 to scale from compact, energy-efficient implementations on small, low-cost FPGAs to high-throughput configurations on larger devices.

Ethical considerations. All experiments were conducted on synthetic target sets with passwords generated for evaluation purposes only. We did not use any leaked, stolen, or user-derived credential datasets.

6 Conclusion and Future Work

In this work, we designed and implemented BF^2, a Bloom-filtered brute-force framework for multi-target password recovery. We demonstrated the capabilities of a hybrid architecture that combines FPGA parallelism for high-throughput hash generation with CPU for large-scale target verification.

On the FPGA side, we implemented a parameterized architecture with 16 parallel pipelines, each containing a password-candidate generator, a fully-pipelined NT hash core, and a Bloom filter stage. We used the low-cost NiteFury II board running at a 100 MHz clock frequency for our evaluations, achieving 1.6×10^9 hashes/s. On the host side, we developed a multi-threaded SIMD-accelerated hashing and verification system using perfect hash functions for exact membership testing.

We evaluated BF^2 with different Bloom filter sizes and numbers of hash indices, and showed the trade-off between false-positives and execution time. We also compared BF^2 with JtR (CPU-based tool) and Hashcat (GPU-based tool) across different character sets and target set sizes. For both 5-character printable ASCII and 8-character lowercase password workloads, our design showed up to 7.5× throughput increase compared to JtR. Compared to Hashcat running on an RTX 5000, BF^2 achieved 80% lower power consumption for 5-character ASCII passwords and 90% lower power consumption for 8-character lowercase passwords. These results demonstrate that BF^2 is a power-efficient alternative for energy or cost constrained scenarios. By leveraging FPGA parallelism for computationally intensive operations and CPU for large-scale verification, the hybrid approach addresses limitations inherent to single-platform solutions.

For future work, an interesting direction is scaling BF^2 to multi-FPGA deployments, beyond a single low-cost device. In such a setting, we can increase throughput and mitigate Bloom filter saturation by using a larger effective filter and exploring new system-level trade-offs.

A further optimization is to use both ports of a true dual-port BRAM per pipeline for each Bloom filter. Since the Bloom filter is written once at initialization and read continuously during candidate testing, both ports can serve reads, halving the number of BRAM copies needed per filter. This can improve the end-to-end throughput for large target sets by enabling larger filters and reducing backpressure. The raw throughput will be unchanged when there are no stalls.

Alternatively, the Bloom filter's address space could be split across multiple BRAMs per pipeline, supporting a larger filter on the same device at the cost of higher complexity. In the current design, all lookups for a base password complete in the same cycle, so BF^2 transfers a single compact result: the base password and a 16-bit mask indicating which pipelines produced a hit. With a split filter, the lookups would finish at different times, so extra logic would be needed to gather each password's results before transfer. This is a larger engineering effort, but a promising way to support larger filters even on low-cost FPGAs.

Another direction would be ASIC implementation for the BF^2 datapath. An ASIC implementation could significantly improve energy and area efficiency, enabling a higher throughput per watt than FPGA-based implementations. However, this also raises a new challenge due to limited post-fabrication flexibility, as it would be difficult to adapt to different hash functions.

Since our design is modular, it will be interesting to extend BF^2 to additional hash functions to characterize the performance and energy implications of different hash algorithms.

Finally, improving efficiency across varying target set sizes and candidate generation is a good direction. As the Bloom filter starts to saturate, the system could dynamically retune parameters or move to alternative strategies. For password-candidate generation, adaptive enumeration strategies such as prioritizing high-probability candidates before the exhaustive search can reduce the execution time for practical scenarios while still preserving full candidate space coverage when required.

Acknowledgements

This work was supported by the European Union's Horizon Europe Project "Security Protection Tools For Networked Medical Devices (SEPTON)", funded under grant agreement No. 101094901.

References

[AAA+23] Ibrahim Alkhwaja, Mohammed Albugami, Ali Alkhwaja, Mohammed Alghamdi, Hussam Abahussain, Faisal Alfawaz, Abdullah Almurayh, and Nasro Min-Allah. Password cracking with brute force algorithm and dictionary attack using parallel programming. *Applied Sciences*, 13(10), 2023. doi:10.3390/app13105979.

[AS09] Kazumaro Aoki and Yu Sasaki. Preimage attacks on one-block MD4, 63-step MD5 and more. In Roberto Maria Avanzi, Liam Keliher, and Francesco Sica, editors, *Selected Areas in Cryptography*, pages 103–119, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-04159-4_7.

[AVWS14] Ayman Abbas, Rian Voß, Lars Wienbrandt, and Manfred Schimmler. An efficient implementation of PBKDF2 with RIPEMD-160 on multiple FPGAs. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 454–461, 2014. doi:10.1109/PADSW.2014.7097841.

[BBD09] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In Amos Fiat and Peter Sanders, editors, *Algorithms - ESA 2009*, pages 682–693, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-04128-0_61.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970. doi:10.1145/362686.362692.

[Dob96] Hans Dobbertin. Cryptanalysis of MD4. In Dieter Gollmann, editor, *Fast Software Encryption*, pages 53–69, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. doi:10.1007/3-540-60865-6_43.

[GLRW10] Jian Guo, San Ling, Christian Rechberger, and Huaxiong Wang. Advanced meet-in-the-middle preimage attacks: First results on full Tiger, and improved results on MD4 and SHA-2. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, pages 56–75, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. doi:10.1007/978-3-642-17373-8_4.

[GPG19] Maruthi Gillela, Vaclav Prenosil, and Venkat Reddy Ginjala. Parallelization of brute-force attack on MD5 hash algorithm on FPGA. In *2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID)*, pages 88–93, 2019. doi:10.1109/VLSID.2019.00034.

[GPPS08] Tim Güneysu, Christof Paar, Gerd Pfeiffer, and Manfred Schimmler. Enhancing COPACOBANA for advanced applications in cryptography and cryptanalysis. In *2008 International Conference on Field Programmable Logic and Applications*, pages 675–678, 2008. doi:10.1109/FPL.2008.4630037.

[Has] Hashcat Project. Hashcat: Advanced password recovery. Accessed: 2026-07-14. URL: https://hashcat.net/hashcat/.

[HY00] Paul E. Hoffman and François Yergeau. UTF-16, an encoding of ISO 10646. RFC 2781, 2000. doi:10.17487/RFC2781.

[KMB+16] Markus Kammerstetter, Markus Muellner, Daniel Burian, Christian Kudera, and Wolfgang Kastner. Efficient high-speed WPA2 brute force attacks using scalable low-cost FPGA clustering. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES*

2016, pages 559–577, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. doi:10.1007/978-3-662-53140-2_27.

[KPP+06] Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, and Manfred Schimmler. Breaking ciphers with COPACOBANA – a cost-optimized parallel code breaker. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, pages 101–118, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. doi:10.1007/11894063_9.

[LGM+19] Lailong Luo, Deke Guo, Richard T. B. Ma, Ori Rottenstreich, and Xueshan Luo. Optimizing Bloom filter: Challenges, solutions, and comparisons. *IEEE Communications Surveys & Tutorials*, 21(2):1912–1949, 2019. doi:10.1109/COMST.2018.2889329.

[LLC22] RHS Research LLC. NiteFury and LiteFury: Xilinx FPGA development board kit in M.2 form factor. https://github.com/RHSResearchLLC/NiteFury-and-LiteFury, 2022.

[MBM25] Konstantina Miteloudi, Lejla Batina, and Nele Mentens. A5/3 make or break: A massively parallel FPGA architecture for exhaustive key search. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(3):361–388, 2025. doi:10.46586/tches.v2025.i3.361-388.

[MDK14] Katja Malvoni, Solar Designer, and Josip Knezovic. Are your passwords safe: energy-efficient bcrypt cracking with low-cost parallel hardware. In Sergey Bratus and Felix F. X. Lindner, editors, *Proceedings of the 8th USENIX Conference on Offensive Technologies*, WOOT'14, page 10, USA, 2014. USENIX Association. URL: https://www.usenix.org/conference/woot14/workshop-program/presentation/malvani.

[Mic26] Microsoft. NT LAN Manager (NTLM) authentication protocol. Microsoft Open Specifications, 2026. Protocol Revision 37.0, Accessed: 2026-07-12. URL: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-nlmp/.

[MU05] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, Cambridge, UK, 2005.

[Ope] Openwall. John the Ripper password cracker. Accessed: 2026-07-14. URL: https://www.openwall.com/john/.

[Riv91] Ronald L. Rivest. The MD4 message digest algorithm. In Alfred J. Menezes and Scott A. Vanstone, editors, *Advances in Cryptology-CRYPTO' 90*, pages 303–311, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. doi:10.1007/3-540-38424-3_22.

[SNKO07] Yu Sasaki, Yusuke Naito, Noboru Kunihiro, and Kazuo Ohta. Improved collision attacks on MD4 and MD5. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, E90-A(1):36–47, January 2007. doi:10.1093/ietfec/e90-a.1.36.

[TVC24] Igor Tasic, Antonio Villafranca, and Maria-Dolores Cano. Reinforcing cybersecurity with Bloom filters: a novel approach to password cracking efficiency. *EURASIP Journal on Information Security*, 2024(1):35, 2024. doi:10.1186/s13635-024-00183-2.

[VHN+11] Anh-Duy Vu, Jea-Il Han, Hong-An Nguyen, Young-Man Kim, and Eun-Jin Im. A homogeneous parallel brute force cracking algorithm on the GPU. In *ICTC 2011*, pages 561–564, 2011. doi:10.1109/ICTC.2011.6082661.

[WBB+10] Lars Wienbrandt, Stefan Baumgart, Jost Bissel, Carol May Yen Yeo, and Manfred Schimmler. Using the reconfigurable massively parallel architecture COPACOBANA 5000 for applications in bioinformatics. *Procedia Computer Science*, 1(1):1027–1034, 2010. ICCS 2010. doi:10.1016/j.procs.2010.04.114.

[WZ14] Friedrich Wiemer and Ralf Zimmermann. High-speed implementation of bcrypt password search using special-purpose hardware. In *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, pages 1–6, 2014. doi:10.1109/ReConFig.2014.7032529.

[ZL20] Zhendong Zhang and Peng Liu. A hybrid-CPU-FPGA-based solution to the recovery of sha256crypt-hashed passwords. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):1–23, Aug. 2020. doi:10.13154/tches.v2020.i4.1-23.