hashcat 7.0.0 - Release Notes

The hashcat development team is excited to announce the release of hashcat 7.0.0.

This is a major update that brings together over two years of work, with contributions from 105 community contributors, 74 of whom are first-time contributors, and approximately 900,000 lines of code changes. It also includes the culmination of all the unannounced changes from the 6.2.x minor releases which have quietly evolved under the radar.

---

1. New Features

There are many new features in Hashcat v7.0.0. In this section, we highlight what we believe are the most relevant to the majority of users. Additional features can be found in the "Other Notable Features and Changes" section.

1.1. Assimilation Bridge

hashcat has traditionally focused on GPU and CPU compute backends for password cracking. With the release of hashcat v7, this changes. The Assimilation Bridge extends hashcat's compute pipeline beyond traditional backends like CUDA and OpenCL. It enables the integration of alternative compute resources

such as FPGAs, remote TPMs, CPU reference implementations, and even scripting environments. These resources can be used independently or together within the same algorithm pipeline, allowing for flexible and hybrid execution models.

All existing hash-mode plugins continue to work as before. The use of Bridges is optional and only takes effect when explicitly declared in a plugin's configuration. This ensures full backward compatibility with existing setups.

For example, with an algorithm like scrypt, GPU acceleration can be used for the PBKDF2 part, while SMix can be offloaded to an FPGA. This allows each part of the algorithm to run on the most suitable hardware, improving both performance and flexibility. While the original goal was to enable FPGAs and GPUs to work together, the Assimilation Bridge in its final implementation supports much more functionality.

Below are some use cases what we can do with the bridge:

1.1.1. Embedded Language Runtimes

hashcat v7.0.0 introduces the **Python bridge** as the primary demonstration example for the Assimilation Bridge. This plugin allows users to write and integrate custom hash-matching algorithms directly in Python, without requiring recompilation. It makes it easy to experiment with new or obscure algorithms without modifying core C code or writing OpenCL or CUDA kernels.

Just write your logic in Python and you're ready to start cracking. Here's a sample how a user can add yescrypt (`$y$...`) support with just one line of code:

```
from pyescrypt import Yescrypt,Mode

def calc_hash(password: bytes, salt: dict) -> str:
  return Yescrypt(n=4096, r=32, p=1,
mode=Mode.MCF).digest(password=password,
settings=hcshared.get_salt_buf(salt)).decode('utf8')
```

See `docs/hashcat-python-plugin-quickstart.md` for details about hashing formats, self-test pairs, or when to use hash-mode 72000 vs. 73000.

The Python Bridge is a fantastic generic hash replacement. If you just need a quick implementation for a CTF or similar use case, this is your friend. You get all the multithreading, the rule engine, chunked wordlist loading, and all the hashcat features for free. The best part is, you don't even need to write a decoder for your hashes!

Here's some quickstart documentation:

See the Python Plugin Quickstart Guide for setup instructions.

Refer to the Plugin Requirements for dependencies.

1.1.2. Hybrid Architecture

Note that in the Python bridge example, only CPU resources are used. hashcat does not convert Python code into GPU kernels. For anti-GPU based algorithms, this approach is often ideal. However, the Assimilation Bridge also supports hybrid setups, where part of the workload runs on a traditional backend like a GPU and another part on the bridge. This allows performance-critical components to run on the most suitable type of compute unit.

For example, in hash mode 70100, a demonstration of scrypt, the PBKDF2 stage runs on a GPU using any of the compute backends OpenCL/CUDA/HIP/Metal, while the memory-intensive SMix runs on the CPU through a bridge using the scrypt-jane implementation.

Our main idea here would be that the same task could just as easily be offloaded to an FPGA instead of a CPU, which would benefit not because PBKDF2 runs well on a GPU, but because removing PBKDF2 code from the FPGA simplifies its logic. The reduced code complexity allows for greater parallelization, making room for more efficient use of FPGA resources.

A mix of traditional backend compute on GPU and embedded Python is also possible.

1.1.3. CPU-Based Reference Code

Bridges can also be used to quickly integrate reference implementations of new algorithms. We will provide initial examples for Argon2 and scrypt. These can run entirely on CPU or form part of a hybrid setup.

- Hash-Mode 70000 uses the official Argon2 implementation from the Password Hashing Competition (PHC) and supports cracking Argon2id hashes. While it could be extended to support Argon2d and Argon2i as well, its primary purpose is to demonstrate the Assimilation Bridge in action. For real-world Argon2 cracking, regardless of type, use the new hash-mode 34000
- Hash-Mode 70200 demonstrates Yescrypt in its scrypt-emulation mode and benefits from AVX512 acceleration on capable CPUs

1.1.4. Other ideas, not yet implemented

- A bridge could be built to interact with TPMs on mobile devices or laptops, accessed through networked agents. This enables challenge/response flows with hardware-backed key storage which is later returned to hashcat for final processing on a GPU (or pre-processed)
- A remote agent running on a cloud server can handle heavy computation. For example, consider VeraCrypt, which performs a high number of iterations, but only involves the password in the initial PBKDF2 step. Once this first iteration is computed, the bridge can delegate the remaining workload to a remote agent. This allows the subsequent iterations to be processed on fast but less secure hardware. Because only an intermediate hash is transferred rather than the actual password, sensitive information such as the password candidate remains local. This approach helps meet data protection requirements.
- Depending on interface compatibility, code from other password cracking tools (e.g., JtR) could be wrapped in bridges, allowing functionality reuse and deeper collaboration

1.1.5. Conclusion

The Assimilation Bridge introduces a highly extensible mechanism to integrate custom compute resources and logic into hashcat. We are both excited and slightly terrified to see what kinds of use cases the community will come up with next.

1.2. Virtual Devices

Another major feature in hashcat v7.0.0 is the introduction of virtual backend devices. This feature was originally added to support the Assimilation Bridge, but it can also be used independently to increase workload parallelism on a physical compute device.

The need for this feature arises from the asymmetric nature of bridging. A bridge can return multiple units that may operate at different speeds or have different optimal batch sizes. When devices with mixed performance are used together, an asynchronous execution strategy is needed to avoid idle time. While hashcat already supports mixed-speed backends, this typically applies to entire backend devices.

Supporting varying speeds within a single backend device, as required by the Assimilation Bridge, called for a new approach. To solve this, hashcat internally partitions a **single** physical device **into multiple** virtual devices similar to how a virtual machine works on a desktop, but applied to a GPU.

This is handled entirely by hashcat and users do not need to configure anything.

Each virtual device is automatically linked to a separate bridge unit, ensuring that each bridge unit can process data at its own pace without being held back by slower units. However, users can optionally control this behavior using two new command-line options:

- Use `-Y` to define how many virtual backend devices to create. This allows dynamic load balancing even outside of bridge modes
- Use `-R` to bind these virtual devices to a specific physical backend device that will act as the virtualization host, or to be more exact the physical compute unit unit, most likely your GPU

When a bridge is in use, the `-Y` parameter is automatically set to match the number of units reported by the bridge. If no bridge is used, `-Y` can be set manually. In both cases, `-R` lets you specify which device should be used as the base for virtualization. By default, this is device 1.

For example, if a bridge initializes and reports two FPGA units, hashcat will automatically create two virtual devices and bind them to the physical device specified by `-R`.

1.3. New Backends

1.3.1. HIP

For those who are unfamiliar: AMD HIP is to AMD GPUs what CUDA is to NVIDIA GPUs. HIP support has been available in hashcat for some time, but it was introduced shortly after the last official release. As a result, some users may not be aware of it yet, so we are highlighting it here. The initial HIP skeleton was contributed by an AMD employee, and we took over development afterwards.

HIP support in hashcat is considered solid and is handled over OpenCL in the same way that CUDA is. When hashcat detects an AMD GPU along with the HIP runtime, it maps two separate devices to the same physical GPU: one using HIP and one using OpenCL. In this situation, hashcat will default to using the HIP device and skip the OpenCL device, but we do it that way so that users can override this by using the `-d` option to select the OpenCL device instead.

In the benchmark section, you will find a benchmark sheet that lets you compare the performance of the same GPU when using OpenCL versus HIP. In most cases, HIP performs better. However, there are corner

cases where OpenCL is still faster.

It is a constant race between the two runtimes. When we got HIP first started working, it was much faster than OpenCL. But since then, especially with OpenCL 3.0, OpenCL has improved significantly and narrowed the gap. That is why we plan to maintain support for both. In some cases OpenCL is faster, and we also need to be prepared in case AMD (or NVIDIA for CUDA) decides to drop support for HIP, CUDA, or OpenCL. If that happens, we will still have an alternative to fall back on.

We also had some issues getting HIP to work on Windows, especially with `hiprtc.dll`. Initially, HIP runtime libraries were bundled with the driver. Later, `hiprtc.dll` was removed, just like `nvrtc.dll` was removed from the NVIDIA driver. We do not fully understand the vendor's reasoning here. The result is simply that users must now install an additional package.

- If you're on Windows, this additional package is the AMD HIP SDK. It is easy to find and install, don't worry
- If you're on Linux, use the ROCm repositories. They also provide `hiprtc.so` in one of their packages

1.3.2. Metal

Apple Metal is Apple's low-level, high-performance graphics and compute API, designed to replace OpenCL on macOS. It provides direct access to GPU hardware and is optimized for Apple Silicon chips (M1, M2, etc.), improving compatibility and performance on modern devices, especially since OpenCL has been deprecated on the platform.

Given Apple's deprecation of OpenCL in macOS 10.14 and their recommendation to migrate toward Metal, hashcat has added Metal support as a forward-looking measure to maintain GPU compute capabilities on macOS platforms. The integration was non-trivial, as it required carefully modifying hashcat's internal data structures without disrupting the existing runtime implementations. Key changes include:

- Updating the Makefile to support the Apple Metal backend and generate a Universal macOS binary, ensuring compatibility with both Intel and Apple Silicon architectures (arm64 and x86_64). In particular, we use MACOSX_DEPLOYMENT_TARGET=15.0 to ensure compatibility with modern macOS versions
- Updating kernel argument definition macros to support both address spaces (private, constant, global, local, etc.) and attribute qualifiers (thread_position_in_grid, thread_position_in_threadgroup, etc.)
- Introducing the M2S macro to handle #include directives in a way that is compatible across Metal and other runtimes, to workaround the Apple way to use the "offline compiler"
- Updating all pointer definitions in the existing hashcat kernels by prefixing them with PRIVATE_AS, since Metal requires the address space to be explicitly defined
- Modifying various hashing algorithms and core components of hashcat, including the Hardware Monitor which required reverse engineering of IOKit behavior, to support the new Metal runtime
- Writing Objective-C code to manage Metal device interactions, including a custom function to bypass Objective-C's type restrictions based on the macOS or Metal version.

The Metal backend was developed and tested across a broad range of Apple devices, including legacy Intel Macs and the latest Apple Silicon (specifically M1 and M4 Pro). Over the course of development, particularly throughout 2025, several optimizations were introduced that significantly boosted performance on both Apple Silicon and discrete AMD GPUs.

One of the key performance improvements came from reworking the Metal memory management system. A configurable structure was introduced at a single point in the codebase: an array that maps each buffer allocated by hashcat to a corresponding Metal Resource Storage Mode. By default, the system uses MTLStorageModeShared, optimal for Apple Silicon, but dynamically switches to MTLStorageModeManaged at runtime when a discrete GPU (e.g., AMD) is detected.

This seemingly simple change led to dramatic performance improvements, especially on AMD discrete GPUs. For example, the Argon2 algorithm improved from ~10 H/s (1330 ms) to 465 H/s (57 ms)- a 4550% increase in hash rate and a 2223% reduction in execution time on the GPU.

1.4. Hash-mode autodetection

To improve usability, hashcat now supports automatic detection of the hash-mode when the -m parameter is omitted. This new feature simplifies the workflow, particularly for users who are unsure which hash-mode corresponds to their input.

When enabled:

- If the input hash matches only one supported format, hashcat will automatically proceed with the corresponding hash-mode
- If the hash is compatible with multiple formats, a list of matching hash-modes will be displayed, allowing the user to choose the correct one

This feature can be triggered either by omitting the `-m` option or by explicitly using the `--identify` option.

- When the `-m` option is omitted and hashcat identifies a single compatible hash-mode, it will automatically proceed with the attack, following the behavior configured by other command-line options.
- When the `--identify` option is specified, hashcat will instead output a list of all hash-modes compatible with the input hash, without executing an attack.

Following an example:

```
$ ./hashcat --identify
'$argon2id$v=19$m=65536,t=3,p=1$FBMjI4RJBhIykCgol1KEJA$2ky5GAdhT1kH4kIgPN/o
ERE3Taiy43vNN70a3HpiKQU'
```

The user will see:

```
The following 2 hash-modes match the structure of your input hash:
      # | Name                                           | Category
========+================================================+============
  70000 | argon2id [Bridged: reference implementation] | Raw Hash
  34000 | Argon2                                         | Generic KDF
```

NOTE: Auto-detect is best effort. The correct hash-mode is NOT guaranteed! Some algorithms are impossible to differentiate based purely on the output hash format. It is important to also use context clues

such as identifying what software created the hash or how it was stored/extracted to determine what algorithm it may be. As always, the gold standard for confirming an algorithm is cracking a hash.

1.5. Docker

The Docker support in this project currently serves primarily as a build environment. We plan to extend this in the future to support running hashcat directly inside the container. For now, the focus has been on building a clean image that can compile binaries inside Docker, including support for cross-compiling from scratch.

If you've downloaded a binary from hashcat.net or its beta site, those binaries were compiled on an old Ubuntu 18.04 system that we have kept alive specifically for this purpose. The reason is simple: compiling on a system with a low GLIBC version ensures the resulting binaries will remain compatible with older Linux distributions.

However, maintaining an old physical or virtual box just for this reason is not ideal. So, we have refactored the build process we previously used into a Docker-based build system. This allows users to reproduce the binaries in a consistent, containerized environment. Currently, we've included builds for both Ubuntu and Arch Linux, just to cover at least two different package management ecosystems and see how platform compatibility behaves across them.

Cross-Compiling for Windows: This Docker setup also supports cross-compilation to Windows. If you want to produce Windows binaries, Docker makes the process straightforward.. The cross-compilation toolchain, along with the necessary third-party dependencies, is already integrated into the image.

Running hashcat Inside Docker: While the main use case is compilation, with a few small tweaks this Docker setup can also be turned into a runtime environment for hashcat itself. In other words, you can run hashcat directly inside the container, not just use it to compile binaries. We did a quick test on an AMD system. By forwarding `/dev/kfd` and `/dev/dri/*` into the container, we managed to get it working. We did not take it further, especially regarding NVIDIA support. But if you are up for the challenge, feel free to submit a PR. We are sure the community would appreciate it.

Custom Patches and Plugins: We also added an optional `docker/patches/` folder. You can drop any number of git diff patch files into this directory. The build process will automatically apply them before compilation starts. This makes it easy to integrate your custom modifications or plugins, even if you have not shared them publicly yet.

2. New algorithms

2.1. Summary

This section lists a summary of all the newly added hash modes in hashcat v7. These include primitives that can be used by other plugin kernels, generic constructions commonly found in web applications, and protocol, application, or operating system-specific hash types which we refer to as "applications" here.

In total, hashcat v7.0.0 adds:

- 58 new applications
- 17 new constructions
- 11 new primitives to the crypto library, ready for reuse

- 20 new extraction tools

2.1.1. Applications

- 1Password, mobilekeychain (8)
- Adobe AEM (SSPR)
- Anope IRC Services
- Apache Shiro 1
- BestCrypt v4 Volume Encryption
- Bisq .wallet
- Bitcoin raw private key (P2PKH/P2WPKH/P2SH(P2WPKH), compressed/uncompressed)
- Citrix NetScaler
- CubeCart
- Dahua NVR/DVR
- DANE RFC7929/RFC8162 SHA2-256
- Dogechain.info Wallet
- Domain Cached Credentials (DCC), MS Cache (NT)
- Domain Cached Credentials 2 (DCC2), MS Cache 2, (NT)
- Empire CMS
- ENCsecurity Datavault
- GPG
- IPMI2 RAKP
- Kerberos 5, etype 17/18, AS-REP
- Kremlin Encrypt 3.0
- LUKS2
- mega.nz password-protected link
- MetaMask Wallet and Mobile Wallet
- Microsoft Online Account
- Microsoft SNTP
- NetIQ SSPR
- PDF 1.3 - 1.6 (Acrobat 4 - 8)
- Perl Mojolicious session cookie
- QNX 7 /etc/shadow
- RACF KDFAES
- RC4 40/72/104-bit DropN
- RSA Security Analytics / NetWitness
- SecureCRT MasterPassphrase v2
- Simpla CMS
- SNMPv3
- Stargazer Stellar Wallet XLM
- Veeam VBK
- WBB4 (Woltlab Burning Board)

2.1.2. Generic constructions

- bcrypt(HMAC-SHA256($pass))
- bcrypt(sha256($pass))

- md5($salt1.$pass.$salt2)
- md5($salt1.sha1($salt2.$pass))
- md5($salt1.strtoupper(md5($salt2.$pass)))
- md5($salt.md5($pass).$salt)
- md5(md5($pass.$salt))
- md5(md5($salt).md5(md5($pass)))
- md5(md5(md5($pass)).$salt)
- md5(md5(md5($pass).$salt1).$salt2)
- md5(md5(md5($pass.$salt1)).$salt2)
- md5(sha1($pass.$salt))
- md5(sha1($salt.$pass))
- md5(sha1(md5($pass)))
- sha256(sha256($pass.$salt))
- sha512(sha512($pass).$salt)
- sha512(sha512_bin($pass).$salt)

2.1.3. Primitives

- AES-GCM
- Argon2 (Argon2i/d/id)
- BLAKE2s
- CAST
- HMAC-BLAKE2S
- HMAC-RIPEMD160
- HMAC-RIPEMD320
- PBKDF1-SHA1
- RC4-40/72/104/128
- RIPEMD-320
- ShangMi 3 (SM3)

2.1.4. Extractions tools

- apfs2hashcat.py
- bisq2hashcat.py
- bitlocker2hashcat.py
- bitwarden2hashcat.py
- cache2data2hashcat.py
- cryptoloop2hashcat.py
- exodus2hashcat.py
- gitea2hashcat.py
- keybag2hashcat.py
- kremlin2hashcat.py
- lastpass2hashcat.py
- luks2hashcat.py
- metamask2hashcat.py
- radmin3_to_hashcat.pl
- shiro1-to-hashcat.py

- truecrypt2hashcat.py
- veeamvbk2hashcat.py
- veracrypt2hashcat.py
- virtualbox2hashcat.py
- vmwarevmx2hashcat.py

2.2. Argon2

Argon2 is an increasingly popular "GPU resistant" key derivation function. The Netherlands Forensic Institute internally began adding support for it to hashcat. It started out as a simple plugin that used the CPU hook functionality of hashcat 6.x to call the Open Source reference implementation for Argon2.

In early 2025 the idea emerged that the 'warp shuffle instructions' offered by most modern GPUs might allow for an efficient GPU-only implementation after all. These instructions are used to exchange GPU register data between up to 32 threads (one warp or wavefront), allowing us to combine all these registers for computing a single hash. This way, an entire 1024 bytes block of the Argon2 algorithm can be stored in registers, enabling much faster access and computations when compared to storing the block in local or global GPU memory. During development of the warp-based Argon2 primitive, we discovered there was already an excellent warp-based implementation for Argon2 developed by Ondrej Mosnáček in 2017 (https://gitlab.com/omos/argon2-gpu), which was also the basis for Argon2 GPU support in John the Ripper. Development of the final Argon2 primitive was thereafter based on this implementation, adapting and engineering it to fit nicely into the hashcat framework.

Note that to maximize performance of Argon2, all hashes in a single hashcat session should have the same options, in particular the memory requirement (m) and parallelization (p) options. It may also be necessary to disable the self-test hash in case it has different options. The reason is that if all hashes have the same options, these options can be resolved by the JIT (at compile time), allowing for a faster runtime performance. This can increase performance by about 5% in some cases, depending on options and hardware.

Development was initially focused exclusively on NVIDIA hardware. The final phases of development saw many contributions from the hashcat community, resulting in an implementation that runs efficiently on NVIDIA and AMD GPUs, Apple Silicon and even CPUs. These combined efforts enable the hashcat community to create plugins for LUKS2, KeePass and many other applications that use Argon2.

Argon2 support was not available in previous versions of hashcat, so there are no historical results listed in the benchmark sheet for comparison.

However, we believe the current implementation demonstrates strong performance. The results below offer a general idea of what to expect. For detailed information, including the operating system, driver versions, and runtime environment, please refer to the benchmark sheet.

Argon2id with RFC 9106 recommended settings (m=65536, t=3, p=1):

- GPU, NVIDIA GeForce RTX 4090: 1703 H/s
- GPU, AMD Radeon RX 7900 XTX: 1367 H/s
- CPU, Intel i7-14700K: 96 H/s
- CPU, AMD Ryzen 9 9900X: 92 H/s

While NVIDIA delivers the highest absolute performance, AMD provides better value in terms of price-to-performance.

A quick note on CPU performance: While the Argon2 implementation in hash-mode 34000 is highly optimized, the OpenCL runtime can't fully take advantage of modern SIMD extensions like AVX512, since it's designed with GPUs in mind. This becomes clear when compared to the Assimilation Bridge Argon2id in hash-mode 70000, which uses the reference implementation that includes built-in SIMD optimizations.

- CPU, Intel i7-14700K: 110 H/s
- CPU, AMD Ryzen 9 9900X: 109 H/s

3. Performance improvements

This section is always our favorite to write in the release notes. Fine-tuning hashcat is a lot of fun, but it's also one of the most important aspects of a password cracker, right? It's one of the reasons why we code hashcat, just to have fun tuning it.

In hashcat v7, there were several relevant changes, all of which impact speed:

- Autotuner refactorization (see below for details)
- Memory management refactorization (see below for details)
- New low-level instruction support (funnelshift and others)
- New backend support (HIP, Metal)
- Individual hash-mode optimizations
- Updated tuning-database entries

As always, we've prepared a benchmark sheet to compare old and new hashcat speeds (see below for the link).

The benchmark lists results for:

- NVIDIA GeForce RTX 4090
- AMD Radeon RX 7900 XTX
- Intel Arc A770
- Apple M4 Pro

We want to emphasize that this benchmark is not meant to compare the speed of the GPUs, but to show the impact that hashcat code changes have on performance. For that reason, the sheet lists the change from the old hashcat version to the new hashcat version, always on the exact same GPU, on the same host system, and using the same driver and runtimes. We made sure to have the latest drivers and runtimes installed. We also downclocked these GPUs slightly to reduce the chances that the firmware changes the clock speed due to heating during measurement.

Benchmarks are generated using our own utility which you can find in tools/benchmark_deep.pl. This tool automates the process of extracting the self-test for each hash-mode, which is then used as the target hash, and actually runs a single hash cracking session in optimized mode (-O) using brute-force attack (-a 3).

We want to highlight some of the patterns you'll see in the benchmark sheet. Listing all hash modes would be too long, so we're only naming a few. Check the link below for the full list.

3.1. Autotuner and memory management

These results are mainly due to autotuner and memory management refactorization. For instance, the 4090 was capping over-subscription as the password candidate buffer was hitting the artificial 4GiB limit (you will read about this later). The autotuner, with increased thread sizes and improved cache hits, also contributes along with the combination of both changes.

- MD5: +8.02% (NV)
- sha512crypt 6: +7.68% (NV)
- PKCS#8 Private Keys: +5.08% (AMD)
- MySQL323: +25.52% (AMD)
- BestCrypt v3 Volume Encryption: +30.99% (NV)
- NTLM: +12.15% (AMD)
- Whirlpool: +113.76% (AMD)
- SAP (BCODE): +57.56% (AMD)
- RSA/DSA/EC/OpenSSH Private Keys (6): +20.52% (AMD)

3.2. Funnelshift

When you see these 16% figures on NV, these are thanks to the new low-level instructions support for funnelshift, which is also why you don't see these gains on the AMD side.

- SHA1: +16.48% (NV)
- JKS Java Key Store Private Keys: +16.20% (NV)

3.3. scrypt

The following are individual optimizations. It took several weeks to get them right. There's a whole section on the scrypt refactorization in this writeup, even though it's probably too complex to explain with words. If you really want to understand all the details, you'll need to check the sources. Also for custom plugins that may exist on scrypt, read the developer notes section below.

- scrypt (16k:8:1) (mode 8900): +16.27% (NV) and 21.52% (AMD)
- scrypt (16k:1:1) (mode 9300): +320.14% (NV) and 60.87% (AMD)

Of course, that applies to all hash-modes that use scrypt (Ethereum, Multibit, BestCrypt v4, ...)

3.4. RAR3

Next is an individual hash-mode optimization result. The old autotuner ran this with too many threads, causing register spilling. Capped it -> profit.

- RAR3-p (Uncompressed): +54.42% (NV)
- RAR3-p (Compressed): +27.35% (NV)

3.5. bcrypt

We discussed the following in an earlier but unofficial release, so we're showing it again here. We achieved this by avoiding bank conflicts on shared memory. Thanks again, Sc00bz!

- bcrypt $2*$: +11.13% (NV)

3.6. Some record

We actually didn't want to mention this, because it's so irrelevant, but then we saw that this is our first algorithm to break the 1 TH/s mark on a single GPU, so it's cool:

- Java Object hashCode(): +83.12% (NV) - 1,348,241,663,329 H/s

3.7. Trade-offs

These ones are interesting because there were two different changes here. One positive and one negative. In DNSSEC we increased the maximum hostname size to 64, which will actually slow down the algorithm because it requires additional transforms. Still, it's faster, thanks to the positive side effects from autotune, ending up with a net positive. The same effect applies to TOTP, where we added support to run multiple TOTP values, which should have reduced the speed, but again we see a net positive thanks to improvements from autotune:

- DNSSEC (NSEC3): +8.40% (NV)
- TOTP (Time-Based One-Time Password): +25.00% (AMD)

3.8. Tuning Database

Here are some entries that mostly increased due to the updated tuning database. Even these days, when there's no longer a strict need for vector datatypes since all GPUs are scalar 1-element, packing data into 2-, 4-, or 8-element types can still be beneficial. It allows the compiler to better optimize register utilization, especially if there are nested loops involved. However, it's a trade-off, because this also typically increases register pressure, so these optimizations usually only work for simple algorithms like raw hashes or some construct types

- md5($salt.$pass.$salt): 15.24% (AMD)
- md5($salt.$pass): +4.05% (NV)

3.9. Intel discrete GPUs

In hashcat v7, we added results from an Intel GPU to the benchmark sheet for the first time. You will notice some surprising numbers, such as:

- NetNTLMv2: 223.92%
- Whirlpool: 536.88%
- MS Office <= 2003 $3, SHA1 + RC4, collider #2: 1039.89%
- SIP digest authentication (MD5): 2076.51%

This is not fake, and also not a bug. These results come from improvements made during the autotuner refactor. Specifically, we added logic that better accounts for how Intel GPUs respond to tuning. But no changes were made to the kernel code. The performance increase is entirely due to better tuning, or more accurately, fixing the missing tuning in earlier versions.

You could say it was simply not tuned for Intel before and that is correct.

It is important to mention that we are not sponsored by any GPU vendor. We buy all GPUs with our own private money because we enjoy this work. However, this also means we don't have access to rare professional or niche hardware, which is why those models are often not properly tuned.

If you want to know more about the autotuner and memory management refactorizations, we'll explain the new strategies in detail in the following section of this writeup.

3.10. CPU Speed

Cracking NTLM on a CPU may be considered outdated these days, but this achievement is still noteworthy. For the first time, we have broken the 10 GH/s mark for NTLM using a CPU rather than a GPU. This was achieved on an AMD Ryzen 9 9900X with 12 cores, although it required the use of the Intel OpenCL runtime.

What makes this so impressive is that we didn't have to do any extra work to optimize for this. The runtime automatically detects AVX512 support, and since hashcat already supports vector datatypes with 16 elements, everything just comes together seamlessly and fully automatically. The refactored autotuner does the rest.

3.11. Conclusion

Here's the benchmark sheet:

Benchmark Spreadsheet

Yes, we know, there are some yellows. We'll try to fix that in v7.1 😃

4. Improvements on existing features

In the following section, we will analyze in detail the most significant enhancements to the existing functionalities.

4.1. Autotune Refactorization

The autotune engine in hashcat has been completely refactored. The previous implementation had become overly complex and difficult to maintain, resulting in degraded tuning quality compared to earlier versions. Users often reported better results with older releases like v6.0.0, and we listened.

The new design takes inspiration from that simpler era but introduces modern enhancements to make autotuning smarter, and more accurate.

4.1.1. Key Improvements: Three-Stage Tuning Process

Autotuning is now split into two parts, theoretical and measured, and runs in three steps in each to find ideal value for Threads (-T), Loops (-u), and Acceleration (-n).

Theoretical Phase:

- Threads: Start with the preferred warp size (e.g., 32 warps on NVIDIA)
- Loops: Start with the smallest base value which, when doubled repeatedly, hits the iteration count exactly
- Accel: Not theoretical. Run quick test cycles targeting just 1/16 of the final runtime goal

Measured Phase:

- Threads: Doubled until reaching 1/8 of the target runtime. The most efficient setting (candidates/sec) is selected
- Loops: Doubled while staying below 1/4 of the target
- Accel: Doubled again, stopping just before overshooting. The final accel is computed as a fractional value if needed. That's why you may see non-power-of-two values

The core idea: early stages keep runtimes low to allow for refinement later. Each phase gets closer to the runtime target without overshooting it.

4.1.2. Smarter Loop Tuning Strategy

Previously, loop counts were tuned using powers of two. That didn't always align well with iteration counts in slow hashes. Some of them are using iteration counts that are not powers of two. New strategy:

- Finds the smallest base that multiplies cleanly to the iteration count (e.g., 125 for 1000)
- Runs the logic twice: once with the original plugin's value, once with that value + 1, and picks the best fit. That's because some plugins (typically PBKDF2 based) must specify an iteration count -1 in the module

The whole point is that this ensures equal kernel runtimes, regardless of iteration offsets.

4.1.3. Dynamic Thread Count (Per Kernel)

Instead of fixed values, thread counts per block are now set based on occupancy hints provided by vendor APIs (CUDA, HIP, OpenCL, Metal). Benefits:

- Smarter per-kernel configuration using find_tuning_function()
- Better GPU utilization and performance
- Improved cross-platform behavior

In addition, if the final thread count is an odd or uncommon value (like 92), hashcat performs two test runs on the neighboring power-of-two values, in this case with 64 and 128. If either shows better performance, it adjusts the thread count and adapts kernel accel to maintain the same overall kernel runtime.

4.1.4. Accel Tuning Enhancements

- Switched from doubling to a capped floating-point multiplier, allowing finer control.
- Leverages new memory management logic (the hardcoded 4 GiB host limit is gone)
- Adjusts based on actual device and host memory availability, removing the need for static caps

4.1.5. Runtime Behavior Adjustments

To simulate real cracking conditions more accurately:

- Invisible warm-up kernels are run during autotune to populate caches and trigger runtime optimizations
- Early tuning stages use small workloads and no over-subscription to keep autotuning fast

4.1.6. Benchmark & Speed Fixes

- Benchmarks now enforce a minimum duration of 4 seconds, regardless of kernel speed
- Fixes issues where fast hashes (like NTLM) are completed too quickly, producing unreliable results
- Helps avoid misleading conclusions when adjusting -n

4.1.7. Default Mask Fix

The benchmark mask has changed from `?b?b?b?b?b?b?b` to `?a?a?a?a?a?a?a`. Why this matters:

- The old mask often generated invalid UTF-8, relevant especially for pure kernels doing UTF-16 conversion
- That caused the kernel to exit early, leading to very short runtimes and incorrect measurements
- In turn, this misled autotune into selecting too high values for accel, especially for slower modes, and invalid benchmark results
- The new mask avoids this by providing a more realistic and consistent character mix

4.1.8. Plugin Developer Guidance

While the new autotuner is robust, some corner cases still exist. Plugin developers can improve tuning behavior by setting sensible minimum and maximum values for `-u`, `-T`, and `-n` in the appropriate plugin callback functions.

We have intentionally left printf() debug statements in autotune.c to help plugin developers understand why hashcat selects certain values. These insights may help you identify strategic points where setting a minimum or maximum value can improve performance of your plugin.

4.2. Memory Management Refactorization

`Memory Management Refactorization` might sound unusual for a tool like hashcat, which allocates most of its buffers on the compute device. But it's actually a complex topic, because hashcat operates in two separate memory worlds: the host and the device. Each has its own limits, constraints, and rules, and they need to work together seamlessly. But in general, the goal of this change is to improve performance, and when you will look at the benchmark sheet, you will see its effect.

It all started with a small change: enabling support for larger memory buffers for the password candidate array buffer, with the simple goal to improve performance by allowing for larger over-subscriptions. This array buffer (for instance) holds the words from the wordlist and must be allocated on both the host and device. Each word takes 260 bytes (256 + 4), so memory usage grows fast.

Example: rockyou.txt

- Size of wordlist: ~139 MB
- Required buffer size on device: ~3700 MB
- Additional compressed buffer on host: +3700 MB
- Total usage: ~8 GB, just for password buffers

And that's only one part of the system, hashcat uses around 30 different buffers overall. As a result, host memory usage scales up quickly.

```
Compressor Kernels:

In case you are wondering about the purpose of the additional compressed
buffer, it is a scratch buffer that hashcat allocates on both the host and
the device. The term "decompress" is a bit misleading, but we will get to
that. This buffer is used to efficiently transfer the wordlist over PCIe.
Hashcat does not actually copy 260 bytes for each password candidate. That
would be a major waste of bandwidth.

Instead, when hashcat copies a new chunk of words to the device, it creates
a long stream of data where each element looks like this: length of next
word followed by the word itself. This pattern repeats for each word. On
the device, before the cracking kernels access the data, hashcat runs a
"decompress" kernel. Again, there is no actual compression involved. This
kernel simply unpacks the password candidates stream into a structure that
compute devices can access directly and efficiently.

This gets even more demanding when using multiple GPUs (e.g., 4 devices),
because each device operates in its own thread and maintains its own copy
of all necessary buffers. Sharing buffers across threads would require
mutexes, which would severely impact performance.
```

You might run into some "out of host memory" error with the new version

If so, try lowering the `-w` setting. With the new memory management, using **-w 4 is not recommended**, as it can quickly exhaust available memory. You don't need it anymore and you'll already achieve higher speeds than the old `-w 4` just by using `-w 3`. Plus, it helps reduce energy consumption.

4.2.1. Thread Scaling

Most algorithms perform better with more than 32 threads per warp. In older versions like v6.2.6, this was limited. Now, hashcat allows up to 1024 threads per warp. However, if the compute runtime recommends a lower value (per kernel), hashcat respects that and adjusts accordingly. That's relevant here, because the thread count has a large impact on the memory requirements.

4.2.2. The Downtuner

So what changed to the old version, and why is that relevant? Previously, hashcat enforced a hard 4 GiB per-device memory limit. The new refactor lifts that restriction and replaces it with a smarter, dynamic system.

The downtuner is a component in hashcat that runs before the autotuner. Its job is to compute the upper limits for the minimum and maximum values of the thread count and the over-subscription count. This is important because these values have a major impact on memory allocations starting from that point in the code. The lower limits are typically set by the plugin module, while the upper limits are based purely on theoretical hardware specifications, such as:

- Number of compute units (e.g., 128 for an RTX 4090)
- Warp size (typically 32 threads per unit)
- Device memory size

- Host memory availability

The ideal thread count for full utilization is `compute_units` * `warp_size`, but PCIe bottlenecks and other factors complicate this. To compensate, hashcat uses:

- In-kernel amplification (rules, iterations)
- Over-subscription (queueing more threads than physically fit to hide latency)

Over-subscription is important in this aspect. It helps keep the GPU busy while one warp is waiting, a concept known as latency hiding, which is controlled by the -n command-line parameter. But it also is another multiplier on the memory buffer size.

4.2.2. Parameter Selection Loop

So all in all, the downtuner evaluates combinations of:

- -n (accel)
- -T (threads)

Each combination is checked against:

- Available memory on the compute device
- Available memory on the host, especially when using multiple devices

Some buffers, like temporary buffers, are device-only and excluded from host calculations (unless hooks or bridges are used). Others, like password candidate buffers, are counted on both sides.

The loop stops when all estimated buffer sizes fit. If you manually start hardcoding values on -T and -n, the loop has no room to adjust. If you set them too high you will most likely run into out of memory errors.

4.3. Testing Framework

Over the years, the framework used by hashcat to test its functionalities has significantly evolved. Changes have been made both to existing scripts (e.g., bug fixes on test.sh and many hash-mode-specific perl test modules) and through integrations prompted by the introduction of new hash-modes (with new perl test modules added for these new hash-modes).

The `install_modules.sh` script has been updated to use `pyenv`, which allows Python dependencies to be installed without modifying the system-wide environment. Unit tests that previously required a local PHP installation have been rewritten, so there are no longer any PHP dependencies. Additionally, the script now uses specific GitHub repositories to install updated versions of certain Perl and Python modules that are no longer actively maintained.

Very recently, a new tool, `test_edge.sh`, was introduced, which also led to modifications in `test.pl`. The purpose of this tool is to generate, via test.pl, edge cases for each hash-mode that has a specific test module, using the constraints defined within it, and to verify hashcat's behavior.

Thanks to this new tool, numerous bugs have been identified and resolved, both in the constraints of the hash-mode-specific test modules and in the kernel code. Finally, the tool benchmark_deep.pl was also modified, primarily by introducing a method to clear the kernel cache on Apple systems, as well as updating the list of hash-modes to be tested.

4.4. Scrypt Refactorization

The handling of scrypt-based algorithms in hashcat has undergone a significant refactor. The core idea was to replace static tuning logic with dynamic, device-aware tuning, making the engine more adaptive, consistent, and reliable.

4.4.1. Dynamic Tuning Strategy

With scrypt, we now bypass the normal autotune path and use a specialized one instead. Previously, scrypt tuning relied on fixed values from a static tuning database. The new design moves the tuning logic into the code path itself, specifically inside module_extra_tuningdb_block(). This function now:

- Computes kernel_accel and TMTO dynamically at runtime
- Simulates different memory usage scenarios (e.g., TMTO = 1 or 2) and checks what fits in device memory
- Leaves space for other buffers (e.g., pws[], tmps[], etc.)
- Injects the calculated kernel_accel into the tuning database as if it came from a file

After returning from module_extra_tuningdb_block(), hashcat proceeds normally and looks up the tuning database. It will find the dynamically added entry. Note: if you manually add entries to the tuning database, they take priority. This can be useful, since the automatic calculation is intentionally conservative. Just keep in mind that due to the memory management refactorization, old tuning-database entries may no longer be valid and need adjustment.

4.4.2. Modularized Design

While performance was the primary goal of this refactor, there was another important motivation: reusability. Many plugins (especially private ones) are based on scrypt. To simplify plugin development, a shared library implementation of scrypt logic was added in `src/modules/`. This can be reused directly from any plugin, avoiding duplicated code.

Additionally, an OpenCL include file was added so that kernel-side logic for scrypt is also reusable and consistent. If you're developing your own plugins, refer to the official scrypt hash-mode implementation as a guide.

4.4.3. Flexible Overrides

The tuning system now allows full control through command-line options: `-n`, `-T`, and `--scrypt-tmto`. If these options are not set, the logic falls back to auto-calculated, device-optimized values. This gives you the flexibility to find the ideal settings for these three parameters, which are critical for maximizing scrypt performance. Once you have found the right values, you can add them to the tuning database.

However, keep in mind that if you do this on a non-headless system, your desktop environment may already be using a significant portion of device memory. If you then force hashcat to allocate memory in that area, it can lead to major performance drops due to swapping. For this reason, we strongly recommend avoiding static tuning values on desktop systems or on systems where other applications might use the same device memory in parallel.

4.4.4. Register Spilling and Memory Safety

What is register spilling or why is it relevant here? When a kernel uses more registers than physically available, it spills values to global memory. If hashcat allocates too much memory and leaves no room for spilling, the runtime may fall back to slow PCIe transfers or, worse, silently overwrite other GPU buffers.

Since scrypt can consume up to 99% of the GPU memory for its B[] buffer, this makes it very likely that such overwrites will hit critical data like candidate buffers, leading to false negatives. These are silent and extremely hard to detect.

To prevent this, hashcat now estimates register spill size per kernel using runtime-provided hints. For example, scrypt typically requires around 7 KiB of spill memory per thread. A configuration with 32 threads and 180 blocks (e.g., default tuning on a 16 MiB setup for a headless 4090) could require over 40 MiB just for spills.

Spill size is now a core part of the downtuner logic, a feature introduced as part of the memory management refactor. This has significantly improved stability on memory-constrained devices also on other hash-modes.

4.4.5. Device and Host Memory Handling

hashcat now queries available memory through low-level APIs:

- NVML (NVIDIA)
- SYSFS (Linux)
- CUDA and HIP runtimes (when available)
- OpenCL runtimes do not support accurate memory reporting

If no accurate low-level API is available, hashcat assumes that only two thirds of the reported memory is usable and shows a warning to the user.

A new option, `--backend-devices-keepfree`, allows users to manually override this two-thirds limit and specify how much memory to reserve instead. By default, one third of the memory is reserved. This value can be lowered, for example to 5, but it is your responsibility to make sure that the memory is actually free.

4.5. Rule Engine Enhancements

The rule engine has recently been improved, both in terms of the kernel code and its underlying logic, thanks to the introduction of support for Character Class Commands. On one hand, the code has been refactored to eliminate duplicate functions and fix minor bugs; on the other hand, Character Class Commands have expanded the logic and brought hashcat's behavior more in line with John the Ripper.

4.5.1. Supported Character Classes

A limited set of character classes are used, compared to those already supported by hashcat:

Pattern	Characters	Description
?l	abcdefghijklmnopqrstuvwxyz	[a-z]
?u	ABCDEFGHIJKLMNOPQRSTUVWXYZ	[A-Z]
?d	0123456789	[0-9]

Pattern	Characters	Description
?h	0123456789abcdef	[0-9a-f]
?H	0123456789ABCDEF	[0-9A-F]
?s	!"#$%&'()*+,-./:;<=>?@[]^_`{	}~

4.5.2. Supported Character Class Commands

These new commands follow a syntax similar to John the Ripper, using the ~ symbol at the beginning of each command.

Supported on both CPU and GPU:

Rule	Description
~s?CY	Replace all characters of class C in the word with Y
~@?C	Remove all characters of class C from the word
~e?C	Capitalize the first letter and every letter following a character from class C

Supported only on CPU:

Rule	Description
~!?C	Reject the word if it contains any character from class C
~/?C	Reject the word unless it contains at least one character from class C
~(?C	Reject the word unless its first character is in class C
~)?C	Reject the word unless its last character is in class C
~=N?C	Reject the word unless the character at position N is in class C
~%N?C	Reject the word unless the word contains at least N characters from class C

4.5.3. Optimized rules

Optimized several rule files, mostly the automatically generated ones, by removing many duplicate rule entries in the following files:

- generated.rule
- generated2.rule
- dive.rule
- d3ad0ne.rule

4.5.4. New Rules

And we added some new rule files:

- T0XlC_3_rule.rule

- T0XlC_insert_HTML_entities_0_Z.rule
- top10_2025.rule

5. Developer and Maintainer Notes

5.1. Updated module interface

Two new callbacks were added to the hashcat module, used to configure the assimilation bridge if needed. This means that if you wrote a private plugin, you must update it for compatibility with hashcat v7. Fortunately, it's a simple change. Just add the following two lines to `module_init()`:

```
  module_ctx->module_benchmark_salt           = MODULE_DEFAULT;
+ module_ctx->module_bridge_name              = MODULE_DEFAULT;
+ module_ctx->module_bridge_type              = MODULE_DEFAULT;
  module_ctx->module_build_plain_postprocess  = MODULE_DEFAULT;
```

And that's it. Sorry for the interruption!

5.2. scrypt

If you are using or writing a private plugin that implements scrypt, there are two important updates to be aware of:

- hashcat now includes a library that handles both the module and kernel parts. While updating your plugin is not strictly required, it is highly recommended and otherwise, you will not benefit from the performance improvements (on average around 16% faster on high memory configurations, a lot more on low memory configurations). Thanks to the new library, upgrading is very straightforward. In fact, you will likely end up removing most of the code you originally copied from the old scrypt plugin template. No new code needed, only removing code. Check out hash-mode 8900 both the module and kernel you are going to love it
- The other thing, regardless of whether you switch to the library, is that the tuning-db entries are no longer valid. If you made optimized tuning-db entries for your hash-mode (which is likely because we needed them in the old scrypt version), then you need to remove them from the tuning database. If you leave them there, you will not benefit from the new runtime tuning logic as manual tuning database entries still have priority. Additionally, from the given perspective, your tuning was made for the old scrypt base, which has changed in the new version, so the values are now outdated

You can still have optimized tuning database entries, but you need to redo them. However, the automated runtime tuning logic is already quite good, and it is likely you do not need to do that additional step.

Also some experiment results:

We experimented with porting warp shuffle logic from Argon2 to scrypt. Two strategies were tested: 8 passwords per warp, and 1 password per warp, both aiming to optimize Salsa's native 4-thread operations. Unfortunately, neither approach matched the performance of the current implementation, where every warp thread processes its own candidate. Performance was roughly cut in half. Too bad, we would have loved to do this as it reduced the minimum password candidates required to achieve full utilization, but not for the price of half the performance.

5.3. Argon2

Adding Argon2 should be as simple as possible, thanks to the Argon2 library available for both modules and kernels. However, there is one important difference compared to the scrypt library that you need to be aware of if you want to use Argon2 in your plugin.

The scrypt library does not require an esalt to function, but Argon2 does. This means you need to copy the esalt structure from the generic Argon2 module (34000) and preserve at least the beginning of that structure exactly as it is. You can then add any additional attributes you need at the end of the esalt structure.

5.4. Assimilation Bridge Documentation

Development documentation on the Assimilation Bridge itself:

Assimilation Bridge Development Guide

There is also developer documentation for the Python Bridge:

Python Plugin Development Guide

5.5. Debugging

The -g flag was introduced under "build options" when hashcat is compiled with DEBUG=1 or higher. This change enables a better debugging experience for kernel code when the selected device is managed by PoCL.

5.6. Tokenizer

You can now define multiple signatures for a single hash-mode in your plugins. This was mainly added for Argon2, so that a single compact plugin can support Argon2i, Argon2d, and Argon2id in one hash-mode, without requiring users to choose the correct one. The signature in the hash itself provides the necessary hint for which Argon2 type is being used.

5.7. Plugin callback workflow:

Introduced `hashes_init_stage5()`, which now calls `module_extra_tmp_size()` instead of `hashes_init_stage4()`. At this stage, the self-test hash is now in fully parsed state, so you can access self-test attributes from `module_extra_tmp_size()`.

This is especially useful for kernels that use the JIT compiler to optimize the kernel with command line macros, which become hardcoded values inside the kernel. You can now detect if the self-test configuration is different from the user-specified hash configuration and reject accordingly because that would lead to false negatives otherwise. See scrypt as an example.

5.8. Large memory allocations

The "4-buffer" strategy has been redesigned to prevent excessive GPU memory usage, and implemented generally also for upcoming anti-gpu algorithms. Previously, the buffer was naively split into four equal parts, but this could cause buffer overflows or waste memory, since the number of work-items is not always a multiple of four. Older versions worked around this by over-allocating an entire extra buffer, which led to

significant memory waste especially in high scrypt configurations (such as 256k:8:1) which would result in 1GiB memory wasted.

The improved logic now calculates exactly how many work-items map to each of the four sub-buffers using: buffer index = workitem_id % 4. Each chunk is sized to fit the exact number of assigned work-items, with any remainder distributed so the first 'leftover' buffers get an extra work-item. This ensures every chunk is large enough, eliminates over-allocation totally, and prevents buffer overflows.

This section of the code is especially relevant to your callback module_extra_buffer_size(), which should be considered whenever very large buffer allocations are needed, such as for modern anti-GPU algorithms. By sizing these buffers precisely, hashcat can efficiently work around OpenCL memory limits, avoiding both misinterpretation and wasted resources.

5.9. Plugin OPTS_TYPE flags

- The new OPTS_TYPE_THREAD_MULTI_DISABLE flag allows plugin developers to disable scaling the candidate batch size based on device thread count. This is useful for very slow algorithms that parallelize differently, such as Argon2 or scrypt. Previously, you could emulate this behavior by setting threads_min and threads_max to 1 in your plugin, but there is an important difference: with this flag, you can still run 32 threads, but hashcat assigns only one password candidate for all 32 threads to process together. This way, if your algorithm allows it, the threads can be used to compute a single hash in parallel
- The new OPTI_TYPE_SLOW_HASH_DIMY_INIT/LOOP/COMP flags allow plugin developers to enqueue kernels in a two-dimensional way. Previously, all hashcat kernels ran in a single dimension, but some situations, such as Argon2 with P>1, allow parallelization of the computation for a single candidate. These new flags make that possible, and are likely to be used in combination with OPTS_TYPE_THREAD_MULTI_DISABLE, though this is not required
- The OPTS_TYPE_MAXIMUM_ACCEL option has been removed. This is just a heads-up in case you use it in private plugins. All existing hash-modes have been updated. The function is no longer needed due to improved autotune support

5.10. Build and Install process:

- Fixed build failure on aarch64 platforms (e.g., Raspberry Pi)
- Updated BUILD instructions for CYGWIN and MSYS2
- Updated build flags on macOS (version placeholder: XX)
- Updated dependencies including UNRAR and OpenCL headers
- Fixed all compiler warnings for both GCC and Clang (latest versions) in hashcat source and suppressed warnings in third-party libraries
- Fixed compilation error in MSYS2 native shell
- Fixed Clang version detection in `src/Makefile`
- Changed package script source directory from `$HOME/hashcat/` to current directory
- Removed old iconv patches (now handled by CMake)
- Fixed bash completion install script
- Updated WSL documentation to use CMake for win-iconv setup
- Fixed MinGW printf format issues
- Fixed compilation on newer FreeBSD versions
- Included winsock2.h instead of winsock.h

- Silenced Clang32/64 warnings related to signed/unsigned comparisons
- Fixed stdcall-related warnings under Clang32/64

5.11. Updated 3rd party dependencies:

- Updated zlib to 1.3.1
- Updated xxHash to 0.8.3
- Updated LZMA SDK to 24.09
- Updated OpenCL-Headers to v2024.10.24

5.12. Auto-detection of hash-modes

To disable automatic hash-mode detection for a specific mode, the flag OPTS_TYPE_AUTODETECT_DISABLE must be added to its OPTS_TYPE definition. This allows fine-grained control over which hash-modes are eligible for autodetection and ensures correct handling of edge cases or ambiguous formats. If you are a plugin developer, read more about this in the developer section below.

Automatic hash-mode detection is currently disabled for the following modes due to ambiguity, special handling requirements, or lack of reliable identification patterns:

- 2000: STDOUT
- 2501: WPA-EAPOL-PMK
- 9000: Password Safe v2
- 9710: MS Office <= 2003 $0/$1, MD5 + RC4, collider #1
- 9720: MS Office <= 2003 $0/$1, MD5 + RC4, collider #2
- 9810: MS Office <= 2003 $3, SHA1 + RC4, collider #1
- 9820: MS Office <= 2003 $3, SHA1 + RC4, collider #2
- 10410: PDF 1.1 - 1.3 (Acrobat 2 - 4), collider #1
- 10420: PDF 1.1 - 1.3 (Acrobat 2 - 4), collider #2
- 16800: WPA-PMKID-PBKDF2
- 16801: WPA-PMKID-PMK
- 20510: PKZIP Master Key (6 byte optimization)
- 22001: WPA-PMK-PMKID+EAPOL
- 25400: PDF 1.4 - 1.6 (Acrobat 5 - 8) - user and owner pass
- 27000: NetNTLMv1 / NetNTLMv1+ESS (NT)
- 27100: NetNTLMv2 (NT)
- 31500: Domain Cached Credentials (DCC), MS Cache (NT)
- 31600: Domain Cached Credentials 2 (DCC2), MS Cache 2, (NT)
- 72000: Generic Hash [Bridged: Python Interpreter free-threading]
- 73000: Generic Hash [Bridged: Python Interpreter with GIL]
- 99999: Plaintext

6. Other notable features and changes

- Masks (and flags): The number of custom charsets has been increased from 4 to 8 (using `-5` / `-6` / `-7` / `-8`, and in mask files)
- Status: The status output "Kernel.Feature" line now includes information about minimum and maximum supported password length when using pure vs optimized kernels

- Status: Keybinds [b]ypass, [c]heckpoint, and [f]inish now work when attack is paused
- Added JSON format support for backend-info, hash-info, status-screen, progress-info, speed-info, and outfiles
- Flags: New option `--benchmark-min` and `--benchmark-max` can now be used to set the start and stop hash-mode number for running multiple benchmarks
- Flags: Using `--benchmark-all` now implies `--benchmark`
- Flags: New option `--total-candidates` flag now shows the full keyspace for an attack (different from the workload-focused `--keyspace`)
- Flags: Due to Markov chain optimization and the use of ordered wordlists, the efficiency of attacks ideally drops off over time. The new flags (`--bypass-delay` and `--bypass-threshold`) let users set thresholds for crack rates, allowing hashcat to automatically move on to the next attack when the results slow down
- Flags: The `--bitmap-max` value is now limited to a maximum of 31
- Flags: When `--outfile-check-timer` is used, the check now runs immediately on startup, instead of waiting until the end of the first specified interval
- Flags: New `-ii` / `--increment-inverse` option reverses the increment direction, processing from right to left rather than left to right
- Flags: New option `--color-cracked` enables ANSI colorizing of cracked hashes
- Flags: The new `--backend-devices-keepfree` option lets users override the default (33%) and specify how much device memory to keep free, which is particularly useful for memory-hungry algorithms
- GitHub: `docs/hashcat-help.md` and `docs/hashcat-example_hashes.md` are now automatically updated via Github actions
- Information/warnings: If the base wordlist does not contain enough entries, hashcat now informs the user with an estimate of how many basewords are needed
- Information/warnings: More informational output is now properly suppressed when using `--quiet`
- Workaround: Microsoft OpenCL D3D12 platform is now automatically detected and ignored, as it often crashes during demanding sessions
- Revisited the Apple OpenCL 2 GiB bug (still present)
- Revisited all module_unstable_warning entries across all plugins and synchronized them with the current driver versions at release time; most warnings have been removed
- KeePass: Added XML support for keyfiles
- RAR3: Added true UTF-8 password conversion support
- Rule Debugging: Added support for using `--debug-mode` in Association Attacks
- Reduced JIT compile time by lowering the required C++ standard of the JIT compiler (C++ is not used), and by enabling multithreaded compilation
- Updated the list of consumer, mobile, and professional NVIDIA, AMD, and Intel GPUs in the tuning database
- Improved benchmark stability: hashcat no longer creates and destroys context and command queues for each device when switching hash-modes, preventing GPU memory leaks with clCreateContext, especially on NVIDIA OpenCL
- Gracefully handle corrupted .gz archives, for example when using compressed wordlists, which is supported by hashcat
- Implemented missing XZ file seeking functionality
- Improved ASN.1 checks for RSA/DSA/EC/OpenSSH private key modules (22911, 22921, 22931, 22941, 22951)

- AuthMe: Updated token length limit
- VeraCrypt and TrueCrypt: General code cleanup with backported fixes and improvements across both legacy and new modes
- Bitwarden: Increased iteration limit
- Electrum: Added support to detect more private key prefixes and reduced false positive reported
- Metamask: Added support for dynamic iteration counts
- RAR: Check UnpackSize to reduce false positives
- md5crypt/sha256crypt: Optimized same-salt cracking by marking them as compatible with OPTS_TYPE_DEEP_COMP_KERNEL
- WPA: allow users to override nonce_error_corrections even if message_pair suggests otherwise
- Flask: Updated session payload max length to 2047
- LastPass: Added IV support
- VeraCrypt: added support for keyfile cascades

7. Bug fixes

- Improved OpenCL function declaration bindings to prevent crashes
- Fixed XZ file seek operation in hc_fseek()
- Return an error for unsupported arbitrary seek operations
- Solved TODOs in hc_fstat() and file handling code
- Fixed syntax error in vmwarevmx2hashcat
- Fixed bug in Hardware Monitor: prevent disabling if ADL fails
- Fixed bug in `--stdout` when multiple computing devices are active
- Prevented NULL dereference in read_restore() via hcmalloc
- Fixed race condition in selftest_init on OpenCL with non-blocking write
- Fixed division by zero bug in fast hashes caused by hashes->st_salts_buf->salt_iter not being used
- Fixed socket leak in brain_server()
- Fixed incorrect comparison result in sort_pot_orig_line()
- Fixed incorrect comparison result in sort_by_src_len()
- Fixed memory leaks in tuning_db_init in tuningdb.c
- Fixed host buffer overflow when copying rules from host to device
- Fixed missing entries in switch_buffer_by_offset_8x4_le_S()
- Properly parse maskfiles and escaped question marks
- DiskCryptor: Fixed unit tests
- PKZIP: Fixed stack buffer overflow in modules
- GOST R 34.11-94: fixed false negative in optimized mode in attack mode 3 for passwords of length 16 or 32
- WinZIP: Fixed hash encoding issues
- Blake2: Removed redundant casts and corrected parameter types for FINAL value
- Electrum (Salt-Type 5): Fixed false positives and false negatives in multihash mode, where only the first hash was marked as cracked regardless of which was actually cracked (affected only beta versions)
- PDF 1.7 Level 8 (Acrobat 10 - 11): Fixed thread count issue on NVIDIA OpenCL (4 bytes were lost per thread for unknown reasons)
- Added recovery from rare non-fatal file locking problems
- RSA/DSA/EC/OpenSSH Private Keys ($1, 3): Added workaround for false positives

8. Plans for future and known Issues

- Migrate the test framework from Perl to Python to improve cross-platform compatibility (long-time project)
- Improve support for hardware with unified memory for more efficient memory access and kernel execution
- Add an FPGA example plugin to the Assimilation Bridge
- Improve performance of *crypt hash modes on Intel discrete GPUs
- Optimize Argon2 performance on Apple devices by using the hardcoded parallelism setting
- Add support for LUKS2 ciphers beyond AES
- Add support for running hashcat inside Docker

9. List of contributors

We would like to thank the following people who contributed to hashcat 7.0.0:

Name	Name	Name	Name	Name
0x588	0xVavaldi	ANeilan	b0lek	b8vr
Banaanhangwagen	banderlog	beschio	brandoncasaba	Brets0150
c0rs	cablethief	Chick3nman	codecuriously	danielnachun
davidbolvansky	davidrozen76	DCNick3	dcorpron-kl	DhruvTheDev1
dloveall	Dook97	dunghm19	e-lliot	enwony
Eomtaeyong820	erasmusc	Fil0sOFF	fin3ss3g0d	flaggx1
Freax13	fse-a	gemesa	gogolovefish	gorgiaxx
Greexter	half-duplex	hans-vh	hashrepublic	hlein
holly-o	hops	its5Q	jarlethorsen	jkramarz
JoeMcGeever	jsteube	jtojanen	ketlerd	khorben
lakiw	Lars-Saetaberget	lhywk	llamasoft	lorenzobilli
loukabvn	magnumripper	MathiasDeWeerdt	matrix	mdawsonuk
mephesto1337	Miezhiko	mohemiv	mostwanted002	neheb
NripeshN	nycex	nyxgeek	Oblivionsage	OutWrest
Ozozuz	PenguinKeeper7	philsmd	piwvvo	qasikfwn
rarecoil	raulperdomo	realSnoopy	redongh	reger-men
rixvet	rjancewicz	roycewilliams	s3inlc	Sc00bz
Slattz	stigtsp	stumblebot	technion	thatux
TheToddLuci0	therealartifex	TomStokes14	tweqx	unix-ninja
ventaquil	vikerup	ViperelB	visitorckw	willcrozi

Name	Name	Name	Name	Name
Xanadrel	Xeonacid	YSaxon	Zgzorx	zyronix

These are the GitHub account names, you can find their profile page on:
https://github.com/$ACCOUNTNAME

10. Final words

The release of v7.0.0 took much longer than we had hoped, and we're fully aware of that. Even writing this document took nearly two weeks, with multiple people contributing to it. But the real challenge was diving into two-year-old git commits and trying to understand the decisions that were made at the time. The details matter a lot, because we had to determine whether certain changes were still relevant or had since been overwritten or replaced. That's a lesson learned for us, and we'll strive to streamline this process for future releases.

hashcat v7.0.0 is packed with new features, performance improvements, and complex, workflow-specific enhancements. With additions like the Assimilation Bridge and new interfaces that depend heavily on third-party applications, such as the Python Bridge, we expect there may be some growing pains. If something doesn't work perfectly out of the box, please don't be discouraged. Instead, open an issue on GitHub, or even better, send in a fix.

We want to sincerely thank all contributors, testers, and supporters who helped bring this release to life. Your work, feedback, and patience over the past months have been invaluable. You know who you are. Thank you.

We hope you enjoy using hashcat v7.0.0 as much as we enjoyed building it.