Advances in open-source password cracking

Dhiru Kholia (dkholia@cs.ubc.ca)

January 13, 2013

1 Abstract

This paper presents advances in the area of open-source password cracking. In particular, it describes the design and implementation of various plug-ins for John the Ripper (henceforth called JtR) [1], Ettercap [2], Nmap [3] and Metasploit Framework [4] for mounting attacks against various password protected file formats, password managers, authentication protocols and hashed passwords. Section 2 present security analysis of various file formats (called "non-hashes") and password managers. Section 3 present security analysis of various authentication protocols. Section 4 present security analysis of various password hashing algorithms.

One of the motivation behind this work is to build open-source security tools which can compete with offerings from commercial companies like Elcomsoft and Passware, who are well known in the field of password recovery. Our work describes various JtR plug-ins offering new functionality which is not available even in existing commercial password recovery softwares. Some of our plug-ins are even faster and more scalable than the ones available commercially.

2 Analysis of security of various file formats.

This section presents analysis of various file formats and programs which allow encryption of user data from a cracking perspective. A note about benchmarking, AMD FX-8120 is not a true 8-core CPU and use dynamic frequency scaling, hence the practical speedups obtained by using all 8 cores will be less than 8x. The maximum speedup factor of AMD FX-8120 is (3.1 GHz / 4 GHz) * 8 = 6x when using all 8 cores. Almost all the JtR plug-ins described in this paper are multi-core (by using OpenMP) as well as multi-node (by using MPI). Some of the plug-ins are also implemented in OpenCL resulting in speed-ups of over 150x.

2.1 Analysis of Password Safe 3.x

Password Safe [5] is a free and open source software program for storing passwords originally authored by Bruce Schneier. From a developer point of view, this format has been easiest to write cracking code for since the database format is well documented in formatV3.txt file [6] and [7]. The same database format is used by Password Gorilla

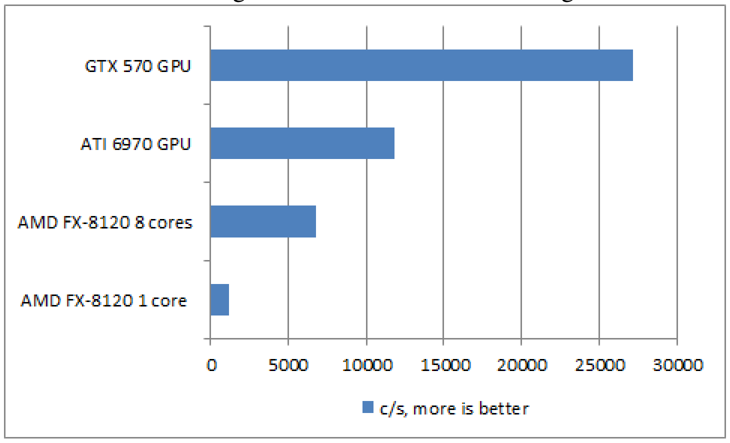
[8] as well as Pasaffe password manager [9], so the analysis here applies to them too. For more details on database format and encryption / decryption process see [7].

Table 1: .psafe 3 database format (header fields, in order)

TAG	4 bytes	The 4 ASCII Characters ‘PWS3’
SALT	32 bytes	256 random bit value generated at file creation
ITER	32 bit LE value	number of rounds in the key stretch algorithm
H(P’) aka HASH	32 bytes	SHA-256 of the user’s “processed” passphrase
B1	16 bytes	Encrypted 128 random value using P’ with Twofish algorithm
B2	16 bytes	Encrypted 128 random value using P’ with Twofish algorithm
B3	16 bytes	Encrypted 128 random value using P’ with Twofish algorithm
B4	16 bytes	Encrypted 128 random value using P’ with Twofish algorithm
Init vector	16 bytes	128 bit random Initialization Vector for the content’s encryption
Header	16 bytes	General information for the database
Records	16 bytes	The records in the database
EOF	16 bytes	unencrypted string “PWS3-EOFPWS3-EOF”
HMAC	32 bytes	256 bit SHA-256 hash of the plaintext contents, starting with the version number in the header and ending with the last field of the last record

From a cracking perspective, only SALT, ITER and H(P’) fields are needed. The Password Safe 3 format uses "variable key stretching" to protect a database against brute-force attacks. The higher the value of "iterations" (ITER) parameter is, the longer it to test a candidate password. The Password Safe 3 format avoids a potential weakness discovered with the old Password Safe 2 ("V2") file format which allowed brute force attacks 1000 times faster than intended. The Password Safe 3 format avoids this issue by depending on the result of the key stretching operation and using it as an input for decryption of data. The key stretching algorithm used in Password Safe is described in following code section. For full implementation details see src/pwsafe2john.c and src/pwsafe_fmt_plug.c.

Figure 1: Password Safe Cracking Benchmarks



Code 1: Password Safe Cracker

```
SHA256_CTX ctx;

SHA256_Init(&ctx);
SHA256_Update(&ctx, password, strlen(password);
SHA256_Update(&ctx, SALT, 32);
SHA256_Final(output, &ctx);

for(int i = 0; i <= ITER; i++) {
  SHA256_Init(&ctx);
  SHA256_Update(&ctx, output, 32);
  SHA256_Final(output, &ctx);
}

if(output == HASH) {
  /* password cracked */
}
```

Our CPU version of the cracking software achieves around 896 c/s on a single core and 7097 c/s on 2 x Xeon E5420 (8 cores total). The GPU version (authored by Lukas Odzioba based on our CPU implementation) achieves a speedup of around 89x over single core CPU result. Currently, the GPU implementation transfers candidate passwords from CPU to GPU which is sub-optimal. Future version of JtR will remove this limitation and higher cracking speeds can be expected.

Password Safe

```
$../run/john -fo:pwsafe -t # AMD FX-8120 (single core)
Benchmarking: Password Safe SHA-256 [32/64 OpenSSL]... DONE
Raw:  1204 c/s real, 1204 c/s virtual

$../run/john -fo:pwsafe -t # AMD FX-8120 (8 cores)
Benchmarking: Password Safe SHA-256 [32/64 OpenSSL]... (8xOMP)
    DONE
Raw:  6826 c/s real, 850 c/s virtual

$ ../run/john -fo=pwsafe -t # Xeon E5420 (1 core)
Benchmarking: Password Safe SHA-256 [32/64 OpenSSL]... DONE
Raw:  896 c/s real, 905 c/s virtual

$ ../run/john -fo=pwsafe -t # 2 x Xeon E5420 (8 cores)
Benchmarking: Password Safe SHA-256 [32/64 OpenSSL]... (8xOMP)
    DONE
Raw:  7097 c/s real, 889 c/s virtual

$ ../run/john -fo:pwsafe-cuda -t # GeForce GTX 570 CUDA
Benchmarking: Password Safe SHA-256 [CUDA]... DONE
Raw:  107185 c/s real, 107185 c/s virtual
```

Password Safe doesn't support the use of "Key Files" described in [10] and [11]. However, for added security, YubiKey hardware [12] which provides 2-Factor Authentication can be used with Password Safe program. So far, no vulnerabilities have been been published for the YubiKey device. Also, it is trivial to increase resistance against brute-force attacks by simply increasing the value of ITER field. This work presents the only known Password Safe cracking software.

2.2 Analysis of Apple Mac OS X Keychain

Apple's keychain is a password management system in Mac OS. Keychain software is an integral part of Mac OS since Mac OS 8.6. In Mac OS X, keychain files are stored in ~/Library/Keychains/, /Library/Keychains/, and /Network/Library/Keychains. The default keychain file is the login keychain (which all users have), typically unlocked on login by the user's login password (blurb borrowed from [13]).

Keychain is an open-source software but compiling modern versions of it on modern Mac OS systems is next to impossible (The whole OpenDarwin project was abandoned in 2006 and the new PureDarwin project has been unable to build security subsystem). Some of the bits required to build Keychain haven't been released as open-source further complicating the compilation process. In addition, Apple's own documentation regarding Keychain's file format [14] is bogus. All these points lead us to doubt the usefulness of Apple's open-source strategy.

Our JtR plug-in and security analysis of Mac OS X Keychain is an extension of the original research done by Matt Johnston (author of extractkeychain program [15]). The following table describes the file format uses by Apple's Keychain software.

Table 2: Keychain file format (DbBlob)

Offset	Length	Name	Purpose
0	4 bytes	Magic Number	Identify Keychain files
4	4 bytes	Version	Identify Keychain version
8	4 bytes	crypto-offset	offset of the encryption and signing key (length 48)
12	4 bytes	total length	total length of the keychain
16	16 bytes	signature	-
32	4 bytes	sequence	-
36	4 bytes	idle timeout	Idle time after which the Keychain is locked automatically
40	4 bytes	lock on sleep flag	-
44	20 bytes	SALT	Salt for PKBDF2
64	8 bytes	IV	Initialization vector for 3DES-EDE
72	20 bytes	Blob Signature	Checksum of the blob
crypto-offset	48 bytes	Ciphertext Data	encryption and signing key (length 48)

Mac OS X Keychain Services API provides functions to perform most of the Keychain operations needed by applications. By using the SecKeychainUnlock function exposed by the mentioned API, it is trivial to create a small bruteforce attack program. An example of such a cracker is shown below,

Keychain Trivial Cracker

```
#include <Security/SecKeychain.h>

int main(void)
{
  OSStatus err;
  char passphrase[128];

  /* attack default keychain */
  SecKeychainRef keychain = NULL;

  /* argv[1] contains target keychain's name */
  while(fgets(passphrase, 128, stdin) != NULL) {
    err = SecKeychainUnlock(keychain,
        strlen(password), password, TRUE);
    if (!err) {
      printf ("Password␣Found␣:␣%s\n", password);
      exit(0);
    }
  }
}
```

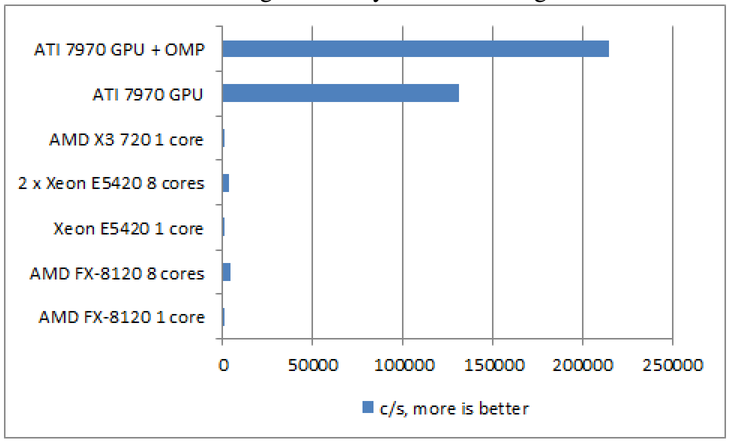
However, such programs are capable of running only on Mac OS systems. In addition, the cracking speed of such programs is typically limited to < 500 passwords per second on modern processors and it is not possible to take advantage of multiple cores due to single-threaded nature of securityd. To overcome these limitations, we have build a custom multi-core cross-platform cracking software for Mac OS X Keychain.

The interesting parts of the Keychain are called "blobs" (see [15]) and there are two types of blobs: database blobs and key blobs. There's only one DbBlob (at the end of the file), and that contains the file encryption key (amongst other things), encrypted with the master key. The master key is derived purely from the user's password, and a salt, also found in the DbBlob. PKCS #5 v2.0 PBKDF2 [18] is used for deriving the master key. The Mac OS X keychain uses the HMAC-SHA-1 function with 1000 iterations, a salt length of 20 bytes and intended length of 24 bytes. In other words, Master Key = PBKDF2-HMAC-SHA(PASSWORD, SALT, 1000, 24). This master key is used to decrypt the encrypted file encryption key. The Mac OS X keychain uses CMS padding [19] for wrapping the file encryption key. The original un-encrypted file encryption key material has length equal to 44 bytes which is padded to 48 bytes before 3DES encryption. We exploit this padding knowledge to figure out if we have successfully decrypted the encrypted file encryption key. We do not know of any other existing research work which uses this technique. The following snippet shows the key steps involved,

Keychain Cracker

```
/* CIPHERTEXT is encrypted file encryption key */

unsigned int master[8];
pbkdf2(PASSWORD,  strlen(PASSWORD), SALT, SALTLEN, 1000, master);
DES_cblock key1, key2, key3;
DES_cblock ivec;
DES_key_schedule ks1, ks2, ks3;
memcpy(key1, key, 8);
memcpy(key2, key + 8, 8);
memcpy(key3, key + 16, 8);
DES_set_key((C_Block *) key1, &ks1);
DES_set_key((C_Block *) key2, &ks2);
DES_set_key((C_Block *) key3, &ks3);
memcpy(ivec, IV, 8);
DES_ede3_cbc_encrypt(CIPHERTEXT, out, 48, &ks1, &ks2, &ks3, &ivec
    ,  DES_DECRYPT);

// now check padding
pad = out[47];
if(pad > 8)
   // "Bad padding byte. Wrong Password.
   return -1;
if(pad != 4)
   // "Bad padding value. Wrong Password.
   return -1;
n = CTLEN - pad;
for(i = n; i < CTLEN; i++)
   if(out[i] != pad)
      // "Bad padding. Wrong Password.
   return -1;

// padding check passed, password found!
printf("Password␣Found");
```

Figure 2: Keychain Cracking Benchmarks



For full implementation details see src/keychain2john.c, src/keychain_fmt_plug.c and src/opencl_keychain_fmt.c files. The algorithms used in our JtR plug-in have been verified by Robert Vežnaver who has written a master's thesis titled "Forensic analysis of the Mac OS X keychain" [17] independently.

Our initial GPU implementation which does PBKDF2 operations on GPU and 3DES operations on multiple-cores is roughly 332X faster than the single-core AMD X3 720 CPU results.

OSX Keychain Benchmarks

```
$ ../run/john -fo:keychain -t # AMD FX-8120 (single core)
Benchmarking: Mac OS X Keychain PBKDF2-HMAC-SHA-1 3DES [32/64]...
     DONE
Raw:  796 c/s real, 796 c/s virtual

$../run/john -fo:keychain -t # AMD FX-8120 (8 cores)
Benchmarking: Mac OS X Keychain PBKDF2-HMAC-SHA-1 3DES [32/64]...
     (8xOMP) DONE
Raw:  4015 c/s real, 502 c/s virtual

$ ../run/john -fo:keychain -t # Xeon E5420 (1 core)
Benchmarking: Mac OS X Keychain PBKDF2-HMAC-SHA-1 3DES [32/64]...
     DONE
Raw:  489 c/s real, 494 c/s virtual

$ ../run/john -fo:keychain -t # 2 x Xeon E5420 (8 cores)
Benchmarking: Mac OS X Keychain PBKDF2-HMAC-SHA-1 3DES [32/64]...
     (8xOMP) DONE
Raw:  3900 c/s real, 489 c/s virtual

$ ../run/john -fo:keychain -t # AMD X3 720 (1 core)
Benchmarking: Mac OS X Keychain PBKDF2-HMAC-SHA-1 3DES [32/64]...
     DONE
Raw:  646 c/s real, 652 c/s virtual

$ ../run/john -fo:keychain-opencl -t # ATI 7970 GPU
OpenCL platform 0: AMD Accelerated Parallel Processing, 2 device(
    s).
Using device 0: Cayman
Benchmarking: Mac OS X Keychain PBKDF2-HMAC-SHA-1 3DES [OpenCL
    ]... DONE
Raw:  131657 c/s real, 262106 c/s virtual

$ ../run/john -fo:keychain-opencl -t # ATI 7970 GPU + OpenMP
OpenCL platform 1: AMD Accelerated Parallel Processing, 2 device(
    s).
Using device 0: Tahiti
Benchmarking: Mac OS X Keychain PBKDF2-HMAC-SHA-1 3DES [OpenCL
    ]... (8xOMP) DONE
Raw:  214809 c/s real, 99200 c/s virtual:W
```

In our opinion, the default number of iterations (1, 000) should be increased for added security against brute-force attacks. However, the current file format used by Keychain does not provide a way to do so without breaking compatibility with existing Mac OS systems. Our cracker is the fastest as well as the only GPU based cracker for Mac OS Keychain files.

2.3 Analysis of 1Password “Agile Keychain”

1Password is a popular password manager available for Windows, iPad, iPhone, Android and Mac platforms. 1Password uses a file format (called Agile Keychain format) which is different from Apple's Keychain file format. The goal of the Agile Keychain file is to build on the successes of the Mac OS X keychain while increasing the flexibility and portability of the keychain design [20]. 1Password stores its data in a folder called "1Password.agilekeychain". 1Password uses JSON (JavaScript Object Notation) format to store its data which has a benefit that its files can be loaded directly into a web browser. It is possible to access the data, without installing 1Password software , by using a web browser. Our JtR plug-in and security analysis of Agile Keychain is an extension of the original research done by Antonin Amand (author of agilekeychain [16])

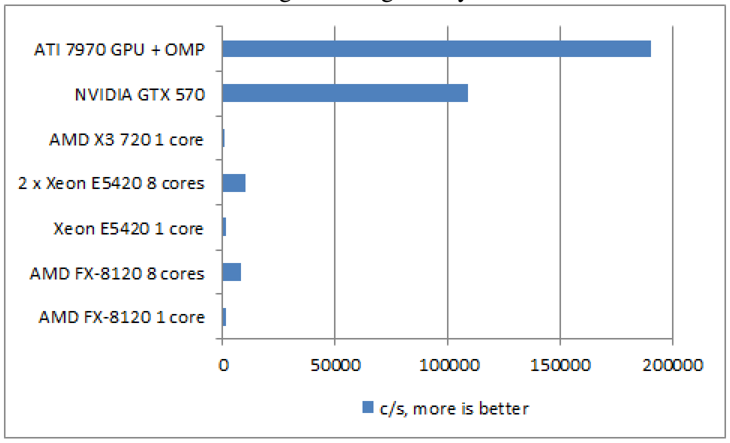
The core of the encryption is AES (Advanced Encryption Standard) using 128-bit encryption keys and performed in Cipher Block Chaining (CBC) mode along with a randomized Initialization Vector. Instead of encrypting data with the password directly, a random key of 1024 bytes is used. This key is stored in the encryptionKeys.js file, encrypted using a key derived from the users master password by using PBKDF2 function. A sample encryptionKeys.js is shown below,

Sample encryptionKeys.js

```
{"list":[{"data":"U2FsdGVkX19xRuqhzKOV5efr...","validation":"
    U2FsdGVkX19dIEp7VK09LOf...",identifier":"1
    D169F66FAAC4745A4C708B254944791","level":"SL5","iterations"
    :1000}

SALT␣=␣identifier's␣value

User␣Encryption␣Key␣=␣PBKDF2-HMAC-SHA(PASSWORD,␣SALT,␣iterations,
    ␣16)
```

We have written a Python program (run/agilekc2john.py) which parses Agile Keychain data and generates a "hash" which is understood by JtR. 1Password uses PKCS#7 padding for wrapping the random encryption key. We exploit this padding knowledge to figure out if we have successfully decrypted the random encryption key.

Figure 3: Agile Keychain Benchmarks



Agile Keychain Cracker

```
/* CIPHERTEXT is encrypted random key */

#define CTLEN 1040

unsigned int master[8];
pbkdf2(PASSWORD, strlen(PASSWORD), SALT, SALTLEN, iterations,
     master);
AES_KEY akey;
AES_set_decrypt_key(master, 128, &akey);
AES_cbc_encrypt(CIPHERTEXT, out, CTLEN, &akey, iv, AES_DECRYPT);

// now check padding
pad = out[CTLEN - 1];
if(pad < 1 || pad > 16) /* AES block size is 128 bits = 16 bytes
     */
   // "Bad padding byte. You probably have a wrong password"
   return -1;

n = CTLEN - pad;
key_size = n / 8;

if(key_size != 128 && key_size != 192 && key_size != 256)
   // "invalid key size"
   return -1;
for(i = n; i < CTLEN; i++)
   if(out[i] != pad)
      // "Bad padding. You probably have a wrong password"
      return -1;

// padding check passed, password found!
printf("Password␣Found");
```

Agile Keychain Benchmarks

```
$ ../run/john -fo:agilekeychain -t # AMD FX-8120 (single core)
Benchmarking: 1Password Agile Keychain PBKDF2-HMAC-SHA-1 AES
    [32/64]... DONE
Raw:  1327 c/s real, 1327 c/s virtual

$../run/john -fo:agilekeychain -t # AMD FX-8120 (8 cores)
Benchmarking: 1Password Agile Keychain PBKDF2-HMAC-SHA-1 AES
    [32/64]... (8xOMP) DONE
Raw:  7953 c/s real, 997 c/s virtual

$ ../run/john -fo:agilekeychain -t # Xeon E5420 (1 core)
Benchmarking: 1Password Agile Keychain PBKDF2-HMAC-SHA-1 AES
    [32/64]... DONE
Raw:  1247 c/s real, 1247 c/s virtual

$ ../run/john -fo:agilekeychain -t # 2 x Xeon E5420 (8 cores)
Benchmarking: 1Password Agile Keychain PBKDF2-HMAC-SHA-1 AES
    [32/64]... (8xOMP) DONE
Raw:  9941 c/s real, 1244 c/s virtual

$ ../run/john -fo:agilekeychain -t # AMD X3 720 (1 core)
Benchmarking: 1Password Agile Keychain PBKDF2-HMAC-SHA-1 AES
    [32/64]... DONE
Raw:  1536 c/s real, 1536 c/s virtual

$ ../run/john -fo:agilekeychain-opencl -t -pla=1 # ATI 7970
Benchmarking: 1Password Agile Keychain PBKDF2-HMAC-SHA-1 AES [
    OpenCL]... (8xOMP) DONE
Raw:  190464 c/s real, 72511 c/s virtual

$ ../run/john -fo:agilekeychain-opencl -t # NVIDIA GTX 570
Benchmarking: 1Password Agile Keychain PBKDF2-HMAC-SHA-1 AES [
    OpenCL]... (8xOMP) DONE
Raw:  108850 c/s real, 46234 c/s virtual
```

In our opinion, the default number of iterations (1,000) should be increased for added security against brute-force attacks. It is trivial to do so by increasing the value of "iterations" parameter in encryptionKey.ks file. However doing this on mobile platforms might have an adverse impace on responsiveness and usability. Our cracker is the only known cracker for Agile Keychain files. Agile Keychain design has one flaw that it doesn't encrypt and protect the metadata (like URL) for a given password. This opens up another attack vector against 1Password software.

Agile Keychain metadata flaw

```
{"uuid":"23F59720EA334163AF","updatedAt":1257723121,"locationKey"
    :"github.com","openContents":{"usernameHash":"11
    dfdb7483af4c2531","passwordStrength":72,"contentsHash":"41716
    f8f","passwordHash":"f48491cd25d4061233d6"},"keyID":"978
    B7BA055427B5E77B","title":"Github","location":"https://github
    .com/session","encrypted":"U20000","createdAt":1257723121,"
    typeName":"webforms.WebForm"}
```

It was interesting to see our cracker being tested and blogged about by official 1Password developer Jeffrey Goldberg [21].

2.4 Analysis of GNOME Keyring

GNOME Keyring is a collection of components in GNOME that store secrets, passwords, keys, certificates and make them available to applications. GNOME Keyring is integrated with the user's login, so that their secret storage can be unlocked when the user logins into their session. GNOME Keyring is a daemon application designed to take care of the user's security credentials, such as user names and passwords. The sensitive data is encrypted and stored in a keyring file in the user's home folder (in ~/.gnome2/keyrings folder) and have "keyring" extension. The default keyring uses the login password for encryption, so users don't need to remember yet another password [22].

GNOME Keyring is implemented as a daemon and uses the process name gnome-keyring-daemon. Applications can store and request passwords by using the libgnome-keyring library. GNOME Keyring is used by various applications like Firefox, Chromium and SSH to stores credentials.

Our program gkcrack [23] is the only program that can crack password protected GNOME Keyrings. However, gkcrack program is not a multi-core capable API due to the gnome-keyring-daemon being single threaded. Besides, gkcrack requires Keyring daemon to be running and also keyring files must be accessible to the daemon.

Code 2: Trivial GNOME Keyring cracker

```c
#include <gnome-keyring.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  char passphrase[128];
  GnomeKeyringResult r;

  /* argv[1] contains target keychain's name */
  while(fgets(passphrase, N, stdin) != NULL) {
    r = gnome_keyring_unlock_sync(argv[1], passphrase);
    if (r == GNOME_KEYRING_RESULT_OK) {
      printf("Password␣Found␣:␣%s\n", passphrase);
      exit(0);
    }
  }
  return 0;
}
```

The following table describes the file format used by GNOME Keyring and is based on official documentation document “file-format.txt” [24].

Table 3: GNOME Keyring file format

Offset	Length	Name	Purpose
0	16 bytes	Magic	"GnomeKeyring\n\r\0\n"
16	2 bytes	Version	Identify version
18	1 byte	crypto	identify crypto algorithm
19	1 byte	hash	identify hash algorithm
20	XX bytes	keyring name	keyring name
20 + XX	4 bytes	ctime	-
24 + XX	4 bytes	mtime	modified time
28 + XX	4 bytes	flags	-
32 + XX	4 bytes	lock_timeout	Idle time after which the Keyring is locked automatically
36 + XX	4 bytes	hash_iterations	Number of iterations, used in KDF
40 + XX	8 bytes	salt	-
48 + XX	4 bytes	num_items	Number of items
52 + XX	YY bytes	num_items data	-
52 + XX + YY	4 bytes	num_encrypted bytes	-
56 + XX + YY	16 bytes	encryted hash	(for decrypt ok verify)

To overcome these limitations of gkcrack, we have implemented an alternate parser and cracker (a JtR plug-in) for Keyring databases. This parser (see src/keyring2john.c for details) outputs a “hash” which can be cracked by corresponding JtR plug-in. GNOME Keyring uses a custom key derivation function based on SHA256 hash function.

GNOME Keyring KDF

```
/* derive KEY and IV from PASSWORD and SALT */

symkey_generate_simple(PASSWORD, SALT, iterations, key, iv)
{
   at_key = key;
   at_iv = iv;
   needed_key = 16;
   needed_iv = 16;
   n_digest = 32;  /* SHA256 digest size */

   for (pass = 0;; ++pass) {
      SHA256_Init(&ctx);
      /* Hash in the previous buffer on later passes */
      if (pass > 0)
         SHA256_Update(&ctx, digest, n_digest);

      if (password) {
         SHA256_Update(&ctx, password, n_password);

      if (salt && n_salt)
         SHA256_Update(&ctx, salt, n_salt);

     SHA256_Final(digest, &ctx);

     for (i = 1; i < iterations; ++i) {
        SHA256_Init(&ctx);
        SHA256_Update(&ctx, digest, n_digest);
        SHA256_Final(digest, &ctx);
     }
     /* Copy as much as possible into the destinations */
     i = 0;
     while (needed_key && i < n_digest) {
        *(at_key++) = digest[i];
        needed_key--;
        i++;
     }
     while (needed_iv && i < n_digest) {
        if (at_iv)
            *(at_iv++) = digest[i];
            needed_iv--;
            i++;
     }
     if (needed_key == 0 && needed_iv == 0)
        break;
   }
}
```

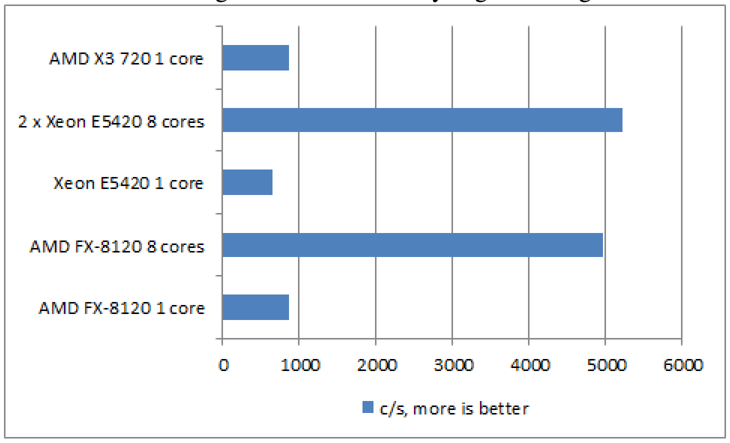
By using the above KDF an AES key and IV are derived, which are then used for decrypting data. AES-128 is used in CBC mode in GNOME Keyring for encrypting and decrypting data. We have written a custom cracker for GNOME Keyring files and following snippet shows the main steps involved, GNOME Keyring cracker

```
decrypt_buffer(buffer, len, salt, iterations, password)
{
  unsigned char key[32];
  unsigned char iv[32];
  AES_KEY akey;
  symkey_generate_simple(password, strlen(password), salt, 8,
      iterations, key, iv);

  (AES_set_decrypt_key(key, 128, &akey);
  AES_cbc_encrypt(buffer, buffer, len, &akey, iv, AES_DECRYPT);
}

verify_decrypted_buffer(buffer, len)
{
  unsigned char digest[16];
  MD5_CTX ctx;
  MD5_Init(&ctx);
  MD5_Update(&ctx, buffer + 16, len - 16);
  MD5_Final(digest, &ctx);
  return memcmp(buffer, digest, 16) == 0;
}

decrypt_buffer(input, crypto_size, salt, iterations, password);
if (verify_decrypted_buffer(input, cur_salt->crypto_size) == 1)
  /* Password found */
else
  /* Password is incorrect */
```

For details see src/keyring2john.c, src/keyring_fmt_plug.c and doc/README.keyring files in JtR source tree.

We compare the performance of gkcrack and GNOME Keyring JtR plug-in on different machines below,

Figure 4: GNOME Keyring Cracking Benchmarks



GNOME Keyring cracking benchmarks

```
$ ../run/john -fo:keyring -t # AMD X3 720 (1 core)
Benchmarking: GNOME Keyring iterated-SHA256 AES [32/64]... DONE
Raw:  875 c/s real, 875 c/s virtual

$../run/john -fo:keyring -t # AMD FX-8120 (single core)
Benchmarking: GNOME Keyring iterated-SHA256 AES [32/64]... DONE
Raw:  874 c/s real, 874 c/s virtua

$../run/john -fo:keyring -t # AMD FX-8120 (8 cores)
Benchmarking: GNOME Keyring iterated-SHA256 AES [32/64]... (8xOMP
    ) DONE
Raw:  4970 c/s real, 618 c/s virtual

$ ../run/john -fo:keyring -t # Xeon E5420 (1 core)
Benchmarking: GNOME Keyring iterated-SHA256 AES [32/64]... DONE
Raw:  650 c/s real, 650 c/s virtual

$ ../run/john -fo:keyring -t # 2 x Xeon E5420 (8 cores)
Benchmarking: GNOME Keyring iterated-SHA256 AES [32/64]... (8xOMP
    ) DONE
Raw:  5214 c/s real, 651 c/s virtual
```

We were also able to attach debugger to a running Keyring daemon process and harvest passwords in clear-text. This attacks works (in-spite of symbols being stripped out) by breaking and tracing calls to gcry_md_write function which is part of Libgcrypt library.

It is possible to port our CPU based GNOME Keyring cracker to run on GPUs by using OpenCL. OpenCL port of our GNOME Keyring cracker are under development. We predict a performance improvement of > 100X based on our experience with Apple's Keychain format.

2.5 Analysis of KDE KWallet

KDE Wallet Manager is a tool to manage the passwords on a KDE system [26] and is described in [27]. KDE Wallet Manager stores passwords in encrypted files, called "wallets" (located in ~/.kde4/share/apps/kwallet folder), which have "kwl" extension. KDE KWallet is implemented as a daemon and uses the process name kwalletd. Applications can store and request passwords by using the libsecret library. Our program kwalletcrack[28] is the only program that can crack password protected KDE KWallet "wallets".

The following table describes the file format used by KDE Wallet and is based on original KWallet paper[27].

Table 4: KDE KWallet file format

Offset	Length	Name
0	12 bytes	Magic String “KWALLET\n\r\0\r\n”
12	1 byte	Format Version - Major (0)
13	1 byte	Format Version - Minor (0)
14	1 byte	Cipher Version (0 - CBC Blowfish)
15	1 byte	Hash Version (0 - SHA-1)
16	8 bytes	Whitening block
36	4 bytes BE	Length of the data stream
40	?? bytes	QDataStream output
??	?? bytes	Padding (random data)
??	20 bytes	Data hash

KDE KWallet uses a custom key derivation function based on SHA256 hash function.

GNOME Keyring KDF

```
password2hash(password, output)
{
    SHA_CTX ctx;
    unsigned char block1[20] = { 0 };
    int i;

    SHA1_Init(&ctx);
    SHA1_Update(&ctx, password, MIN(strlen(password), 16));
    for (i = 0; i < 2000; i++) {
        SHA1_Final(block1, &ctx);
        SHA1_Init(&ctx);
        SHA1_Update(&ctx, block1, 20);
    }
    memcpy(hash, block1, 20);
}
```

It is possible to mount time–memory trade-off attacks [29] (i.e. use Rainbow Tables) against KDE KWallet since it doesn’t employ any salting. Blowfish CBC [29] with 160-bit key is used for encryption and decryption of data.

The following program show the main steps involved in decryption of data and detection whether the password was correct or not.

Code 3: KDE KWallet cracker

```
verify_passphrase(passphrase)
{
    unsigned char key[20];
    password2hash(passphrase, key); /* use custom KDF */
    SHA_CTX ctx;
    BlowFish _bf;
    CipherBlockChain bf(&_bf);
    bf.setKey((void *) key, 20 * 8);
    bf.decrypt(CIPHERTEXT, CTLEN);

    // strip the leading data, one block of random data
    t = CIPHERTEXT + 8;

    // strip the file size off
    long fsize = 0;
    fsize |= (long (*t) << 24) &0xff000000;
    t++;
    fsize |= (long (*t) << 16) &0x00ff0000;
    t++;
    fsize |= (long (*t) << 8) &0x0000ff00;
    t++;
    fsize |= long (*t) & 0x000000ff;
    t++;
    if (fsize < 0 || fsize > long (encrypted_size) - 8 - 4) {
        // file structure error. wrong password
        return -1;
    }
    SHA1_Init(&ctx);
    SHA1_Update(&ctx, t, fsize);
    SHA1_Final(testhash, &ctx);
    // compare hashes
    sz = encrypted_size;
    for (i = 0; i < 20; i++) {
        if (testhash[i] != buffer[sz - 20 + i]) {
            return -1; /* wrong password */
        }
    }
    printf("Password␣Found!");
}
```

The speed achieved by our cracker is 1900+ passwords per second on AMD X3 720 CPU @ 2.8GHz (using single core). It is possible to port our CPU based kwalletcrack program to run on GPUs by using OpenCL. We predict a performance improvement > 100X based on our experience with Apple's Keychain format. JtR plug-in and OpenCL port of our KDE KWallet cracker are under development.

2.6 Analysis of KeePass Password Safe (both 1.x and 2.x)

KeePass is a free open source password manager, which helps you to manage your passwords in a secure way [30]. It is a Windows application with unofficial ports for Linux, Mac OS X and Android platforms KeePass stores passwords in an encrypted file (database). This database is locked with a master password, a key file and/or the current Windows account details. To open a database, all key sources (password, key file) are required. Together, these key sources form the Composite Master Key [31].

The 2.x database format is documented only in code and differs from 1.x version (which is well documented). For more details on database format and encryption / decryption process see [32]. The following table shows the header structure of KeePass 1.x databases.

Table 5: Database Format 1.x

FileSignature1	4 bytes int LE order	0x9AA2D903, Magic
FileSignature2	4 bytes int LE order	0xB54BFB67, Magic
Flags	4 byte int LE order	Determine what algorithms are used
Version	4 byte int LE order	Version of the database format
Final Random Seed / FRS	16 bytes	Initial random number to start on the sha256 of the key
Init Vector / IV	16 bytes	Initialization vector used for all algorithms
Num Group	4 byte int LE order	Encrypted 128 random value using P' with Twofish algorithm
Num Entries	4 byte int LE order	Encrypted 128 random value using P' with Twofish algorithm
Content Hash / CHASH	32 bytes	SHA256 hash of only the contents (entire file minus starting 124 bytes)
Transformed Random Seed / TRS	32 bytes	Random seed used to combine with the master key when calculating the final key
Key Encoding Rounds / ITER	4 byte int LE order	Number of rounds to do AES block encryption on the Master Key

The Contents of the binary file format are in the general format of an unencrypted 124 byte header followed by the encrypted data.

The following table show the header structure of KeePass 2.x databases.

Table 6: Database Format used in 2.x (some fields are out of order)

FileSignature1	4 bytes int LE order	0x9AA2D903, Magic
FileSignature2	4 bytes int LE order	0xB54BFB67, Magic
Version	4 byte int LE order	Versioning Information
Field ID	1 byte	Identifies type of entry which follows
Init Vector / IV	16 bytes	Initialization vector used for all algorithms
Field ID	1 byte	Identifies type of entry which follows
Expected Start Bytes	32 bytes	Used for password validation
Field ID	1 byte	Identifies type of entry which follows
Transformed Random Seed / TRS	32 bytes	Random seed used to combine with the master key when calculating the final key
Field ID	1 byte	Identifies type of entry which follows
Key Encoding Rounds / ITERATIONS	4 byte int LE order	Number of rounds to do AES block encryption on the Master Key
Field ID	1 byte	Identifies type of entry which follows
Final Random Seed / FRS	16 bytes	Initial random number to start on the sha256 of the key

To generate the final 256-bit key that is used for the block cipher (for data encryption), KeePass first hashes the user's password using SHA-256, encrypts the result N times using the Advanced Encryption Standard (AES) algorithm (called key transformation rounds from on now), and then hashes it again using SHA-256. This key-stretching algorithm slows down brute-force attacks significantly[33].

KeePass custom KDF

```
/* custom KDF based on AES and SHA256 */

transform_key(char *PASSWORD, final_key)
{
     // First, hash the PASSWORD
    SHA256_CTX ctx;
    AES_KEY akey;
    SHA256_Init(&ctx);
    SHA256_Update(&ctx, PASSWORD, strlen(PASSWORD));
    SHA256_Final(hash, &ctx);
    if(version == 2) { /* 2.x database */
        SHA256_Init(&ctx);
        SHA256_Update(&ctx, hash, 32);
        SHA256_Final(hash, &ctx);
    }
    AES_set_encrypt_key(TRS, 256, &akey);
    // Next, encrypt the created hash
    for(i = 0; i < ITERATIONS1; i++) {
        AES_encrypt(hash, hash, &akey);
        AES_encrypt(hash+16, hash+16, &akey);
    }
    // Finally, hash it again...
    SHA256_Init(&ctx);
    SHA256_Update(&ctx, hash, 32);
    SHA256_Final(hash, &ctx);
    // and hash the result together with the Final Random Seed
    SHA256_Init(&ctx);
    if(version == 1) {
        SHA256_Update(&ctx, FRS, 16);
    }
    else {
        SHA256_Update(&ctx, FRS, 32);
    }
    SHA256_Update(&ctx, hash, 32);
    SHA256_Final(final_key, &ctx);
}
```

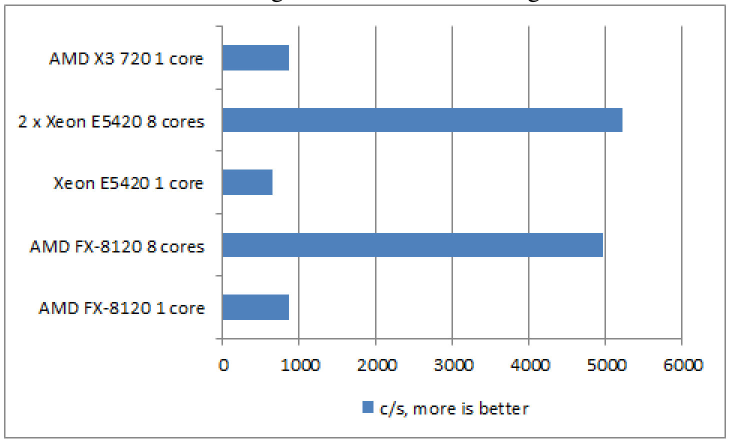
To validate the password for 1.x version, the full contents of the database are required. However version 2.x contains a 32-byte field, expected_startbytes which can be used for password validation. The following box shows the key processing and password validation algorithms which differ slightly between 1.x and 2.x versions of KeePass. Support for key files in cracking has not been implemented in the initial version of KeePass cracker but it can be easily added. We have written a custom cracker for KeePass files and following snippet shows the main steps involved,

Code 4: KeePass Cracker

```
transform_key(PASSWORD, final_key); /* use custom KDF */

AES_set_decrypt_key(final_key, 256, &akey);
if(VERSION == 1) {
  AES_cbc_encrypt(CIPHERTEXT, PLAINTEXT, SIZE, &akey, IV,
      AES_DECRYPT);
  pad_byte = PLAINTEXT[SIZE-1];
  datasize = cur_salt->contentsize - pad_byte;
  SHA256_Init(&ctx);
  SHA256_Update(&ctx, PLAINTEXT, datasize);
  SHA256_Final(out, &ctx);
  if(out == CHASH) {
    /* password found */
  }
}
else {
  AES_cbc_encrypt(CIPHERTEXT, PLAINTEXT, 32, &akey, iv,
      AES_DECRYPT);
  if(memcmp(PLAINTEXT, expected_bytes, 32) == 0)
    /* password found */
}
```

We compare the performance of KeePass JtR plug-in on different machines below,

Figure 5: KeePass cracking benchmarks



KeePass cracking benchmarks

```
$ ../run/john -fo:keepass -t # AMD X3 720 (1 core)
Benchmarking: KeePass SHA-256 AES [32/64 OpenSSL]... DONE
Raw:  69.3 c/s real, 68.6 c/s virtual

$ ../run/john -fo:keepass -t # AMD FX-8120 (single core)
Benchmarking: KeePass SHA-256 AES [32/64 OpenSSL]... DONE
Raw:  82.0 c/s real, 82.0 c/s virtual

$ ../run/john -fo:keepass -t # AMD FX-8120 (8 cores)
Benchmarking: KeePass SHA-256 AES [32/64 OpenSSL]... (8xOMP) DONE
Raw:  409 c/s real, 51.2 c/s virtual

$ ../run/john -fo:keepass -t # Xeon E5420 (1 core)
Benchmarking: KeePass SHA-256 AES [32/64 OpenSSL]... DONE
Raw:  65.3 c/s real, 65.3 c/s virtual

$ ../run/john -fo:keepass -t # 2 x Xeon E5420 (8 cores)
Benchmarking: KeePass SHA-256 AES [32/64 OpenSSL]... (8xOMP) DONE
Raw:  519 c/s real, 64.9 c/s virtual
```

Our work is the only multi-core capable KeePass cracking software available. It should be noted that KeePass's usage of using AES in KDF is quite unique. Cracking KeePass can be accelerated by using GPU(s) to implement the KDF (key derivation function). It is possible to port our CPU based KeePass program to run on GPUs by using OpenCL. We predict a performance improvement > 100X based on our experience with Apple's Keychain format.

2.7 Analysis of SSH private keys

SSH is widely used network protocol for secure communication, remote command execution and remote shell services among networked computers. SSH supports password-based authentication but an attacker can mount a MiTM (man-in-the-middle) attack if the unknown public key is verified and allowed by the end user. In addition, a brute-force attack can be used to discover accounts protected by weak passwords. So, it if often recommended to use public key authentication instead of password-based authentication.

Private key files generated by ssh-keygen command can be password protected. The basic idea behind password protecting the private key files is that even if the attacker has access to private key files, he won't be able to use them for gaining further access. It is possible (and even trivial) to write software for cracking password protected private key files. A sample program for doing so is shown below.

Code 5: SSH trivial Cracker

```
process_private_keyfile(filename, PASSWORD)
{
    BIO *bp;
    EVP_CIPHER_INFO cipher;
    EVP_PKEY pk;
    bp = BIO_new(BIO_s_file());
    BIO_read_filename(bp, filename);
    PEM_read_bio(bp, &nm, &header, &data, &len);

    PEM_do_header(&cipher, data, &len, NULL, PASSWORD);

    if ((dsapkc = d2i_DSAPrivateKey(NULL, &data, len)) != NULL) {
      /* found PASSWORD */
    }
    else
      /* wrong PASSWORD */
}
```

Our first version of JtR plug-in (see src/ssh_fmt.c) was based on similar technique described above. However, it had issues utilizing multiple cores (due to usage of OpenSSL functions, some of which are not thread-safe). In addition, for larger key sizes the cracking speed was slower (due to larger quantity of data being decrypted). To avoid such problems, the SSH JtR plug-in was re-designed and re-written from scratch.

Instead of using standard key derivations functions like PBKDF2, SSH employs a very weak custom KDF shown below, SSH custom KDF

```
generate_key_bytes(PASSWORD, REQUIRED_SIZE, final_key)
{
    unsigned char digest[16] = {0};
    int keyidx = 0;
    int digest_inited = 0;
    int size = 0;

    while (REQUIRED_SIZE > 0) {
        MD5_CTX ctx;
        MD5_Init(&ctx);
        if (digest_inited)
            MD5_Update(&ctx, digest, 16);
        MD5_Update(&ctx, PASSWORD, strlen(PASSWORD));
        /* use first 8 bytes of salt */
        MD5_Update(&ctx, SALT, 8);
        MD5_Final(digest, &ctx);
        digest_inited = 1;
        if (REQUIRED_SIZE > 16)
            size = 16;
        else
            size = REQUIRED_SIZE;
        /* copy part of digest to keydata */
        for(i = 0; i < size; i++)
            final_key[keyidx++] = digest[i];
        REQUIRED_SIZE -= size;
    }
}
```

This allows cracking of password protected private key files at very high speeds. This custom KDF function drives either 3DES or AES-128 in CBC mode which are used for encrypting / decrypting key material. We have written a custom cracker for SSH private files and following snippet shows the main steps involved,

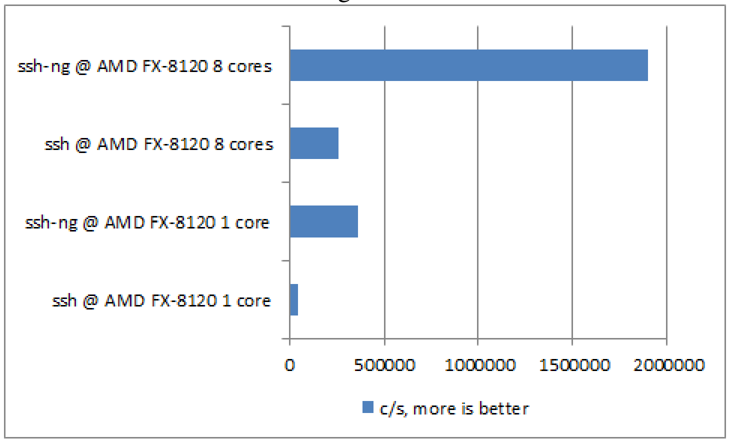
Code 6: SSH Cracker

```
int check_padding_and_structure(out, length)
{
    pad = out[length - 1];
    if(pad > 16) return -1; // Bad padding byte
    n = length - pad;
    for(i = n; i < length; i++) // check padding
        if(out[i] != pad) return -1;

    /* match structure with known standard structure */
    outfile = BIO_new(BIO_s_mem());
    ASN1_parse(outfile, out, legnth, 0);
    BIO_gets(outfile, (char*)output, N);
    res = memem(output, 128, "SEQUENCE", 8);
    if (!res) goto bad;
    BIO_gets(outfile, (char*)output, N);
    res = memem(output, 128, ":00", 3);
    if (!res) goto bad;
    res = memem(output, 128, "INTEGER", 7);
    if (!res) goto bad;
    BIO_gets(outfile, (char*)output, N);
    res = memem(output, 128, "INTEGER", 7);
    if (!res) goto bad;
    /* now this integer has to be big, check minimum length */
    ul = strlen((char*)res);
    p = res;
    while(*p) {
        if (isspace(*p))
            ul--;
        p++;
    }
    if (ul < 32) goto bad;
    return 0;
bad:
    return -1;
}

generate_key_bytes(PASSWORD, 16, key);
AES_set_decrypt_key(key, 128, &akey);
memcpy(iv, SALT, 16);

// We don't decrypt all the encrypted key material!
AES_cbc_encrypt(CIPHERTEXT, PLAINTEXT, 32, &akey, iv, AES_DECRYPT)
    ;

// 2 blocks (32 bytes) are enough to self-recover from bad IV,
    required for correct padding check
AES_cbc_encrypt(CIPHERTEXT + LENGTH - 32, PLAINTEXT + LENGTH - 32,
     32, &akey, iv, AES_DECRYPT);

if (check_padding_and_structure(PLAINTEXT, LENGTH) == 0)
    /* Password Found */
else
    /* Password was not correct */
```

For details see src/ssh_ng_fmt_plug.c and run/sshng2john.py in JtR source tree.

The key technique (through which we gain a speed-up of 5X) is that we only do partial decryption of encrypted key material. After this partial decryption, we employ ASN.1 BER partial decoding to detect if the decrypted structure matches the standard key structure. RSA private key files have structure { version = 0, n, e, d, p, q, d mod p-1, d mod q-1, q**-1 mod p } and DSA private key files have structure {version = 0, p, q, g, y, x }. We exploit this knowledge (structure of private keys) to detect if decryption of key material is correct. We haven't found any false positives (so far) by employing the combination of partial decryption and decoding. To the best of our knowledge, the techniques used by us in cracking password protected private keys are original and haven't been described in existing research literature.

Figure 6: SSH Benchmarks



SSH cracking benchmarks

```
$ ../run/john -fo:ssh -t # AMD FX-8120 (single core)
Benchmarking: SSH RSA/DSA (one 2048-bit RSA and one 1024-bit DSA
    key) [32/64]... DONE
Raw:  42040 c/s real, 42040 c/s virtual

$ ../run/john -fo:ssh-ng -t # AMD FX-8120 (single core)
Benchmarking: ssh-ng SSH RSA / DSA [32/64]... DONE
Raw:  362003 c/s real, 362003 c/s virtual

$ ../run/john -fo:ssh -t # AMD FX-8120 (8 cores)
Benchmarking: SSH RSA/DSA (one 2048-bit RSA and one 1024-bit DSA
    key) [32/64]... (8xOMP) DONE
Raw:  259072 c/s real, 32303 c/s virtual

$ ../run/john -fo:ssh-ng -t # AMD FX-8120 (8 cores)
Benchmarking: ssh-ng SSH RSA / DSA [32/64]... (8xOMP) DONE
Raw:  1899K c/s real, 237376 c/s virtual
```

Our “ssh-ng” code running on 1 core is faster than older “ssh” code running on 8 cores. A correct full BER decoding results in data which looks like following snippet,

Code 7: Correct BER decoding

```
1      0:d=0  hl=4 l= 602 cons: SEQUENCE
2      4:d=1  hl=2 l=   1 prim: INTEGER:00
3      7:d=1  hl=3 l= 129 prim: INTEGER:D38F56EA0785A66B258D3C1FBC...
4    139:d=1  hl=2 l=   1 prim: INTEGER:23
5    142:d=1  hl=3 l= 128 prim: INTEGER:12223AA6586A8A9B783F4E4BDC...
6    340:d=1  hl=2 l=  65 prim: INTEGER:DAB31996BF129CC5E09F291AD8...
7    407:d=1  hl=2 l=  65 prim: INTEGER:F0912D2E5EA55D1738073BA4BE...
8    474:d=1  hl=2 l=  64 prim: INTEGER:12BEE4EFA9FA47F3B42AE64421...
9    540:d=1  hl=2 l=  64 prim: INTEGER:3C86DDCC43E5D297DC882484BC...
```

We however only verify the partial key structure (till line number 3). The security mechanism used in password protected private keys is quite weak. We recommend usage of PBKDF2 / bcrypt / crypt as KDF instead of weak MD5 based custom KDF function to increases resistance against brute-force attacks.

This work represent the state-of-the-art cracker for SSH password protected private keys.

2.8 Analysis of PuTTY private key files

PuTTY is a free Telnet and SSH client for Windows and Unix platforms. It is de facto SSH client on Windows platforms. Our JtR plug-in and security analysis of PuTTY private key files is based on the original research done by Michael Vogt (author of P-ppk-crack [34]). PuTTY uses a custom file format to store private keys.

Instead of using standard key derivations functions like PBKDF2, PuTTY employs a very weak custom KDF shown below,

PuTTY custom KDF

```
password2key(PASSWORD)
{
    int passlen = strlen(PASSWORD);
    unsigned char key[40];
    SHA_CTX s;
    SHA1_Init(&s);
    SHA1_Update(&s, (void*)"\0\0\0\0", 4);
    SHA1_Update(&s, passphrase, passlen);
    SHA1_Final(key + 0, &s);
    SHA1_Init(&s);
    SHA1_Update(&s, (void*)"\0\0\0\1", 4);
    SHA1_Update(&s, passphrase, passlen);
    SHA1_Final(key + 20, &s);

    /* variable key now contains AES-256 key */

}
```

This allows cracking of password protected private key files at very high speeds. This custom KDF function drives AES-256 in CBC mode which is used for encrypting / decrypting key material. PuTTY private key files have a MAC value which allows us to easily verify if we have decrypted the data correctly.

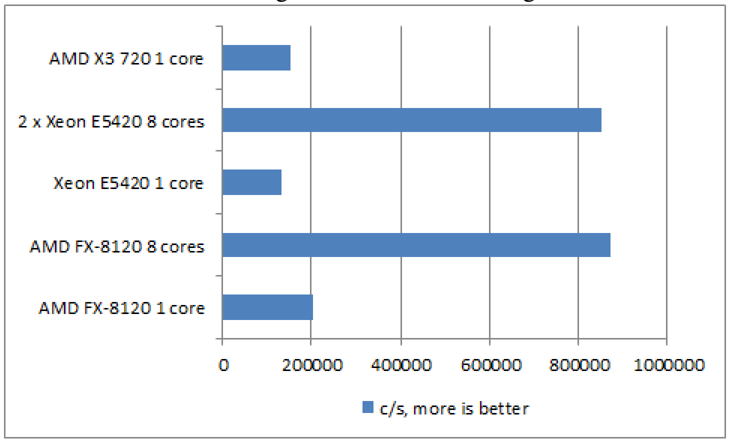
The following snippet shows the main steps involved in decryption and verification of key material,

Code 8: PuTTY cracker

```c
unsigned char iv[32] = { 0 };
(AES_set_decrypt_key(key, 256, &akey);
AES_cbc_encrypt(private_blob, out , private_blob_length, &akey, iv, AES_DECRYPT);

/* Verify the MAC. */
char realmac[41];
unsigned char binary[20];
unsigned char *macdata;
int maclen;
int free_macdata;
int i;
unsigned char *p;
int namelen = strlen(cur_salt->alg);
int enclen = strlen(cur_salt->encryption);
int commlen = strlen(cur_salt->comment);
maclen = (4 + namelen +
    4 + enclen +
    4 + commlen +
    4 + cur_salt->public_blob_len +
    4 + cur_salt->private_blob_len);
macdata = (unsigned char*)malloc(maclen);
p = macdata;
#define DO_STR(s,len) PUT_32BIT(p,(len));memcpy(p+4,(s),(len));p+=4+(len)
DO_STR(cur_salt->alg, namelen);
DO_STR(cur_salt->encryption, enclen);
DO_STR(cur_salt->comment, commlen);
DO_STR(cur_salt->public_blob, cur_salt->public_blob_len);
DO_STR(out, cur_salt->private_blob_len);
free_macdata = 1;
}
if (cur_salt->is_mac) {
  SHA_CTX s;
  unsigned char mackey[20];
  unsigned int length = 20;
  // HMAC_CTX ctx;
  char header[] = "putty-private-key-file-mac-key";
  SHA1_Init(&s);
  SHA1_Update(&s, header, sizeof(header)-1);
  if (cur_salt->cipher && passphrase)
    SHA_Update(&s, passphrase, passlen);
  SHA1_Final(mackey, &s);
  hmac_sha1(mackey, 20, macdata, maclen, binary, length);
  /* HMAC_Init(&ctx, mackey, 20, EVP_sha1());
   * HMAC_Update(&ctx, macdata, maclen);
   * HMAC_Final(&ctx, binary, &length);
   * HMAC_CTX_cleanup(&ctx); */
} else {
  SHA_Simple(macdata, maclen, binary);
}
if (free_macdata)
  MEM_FREE(macdata);
for (i = 0; i < 20; i++)
  sprintf(realmac + 2 * i, "%02x", binary[i]);

if (strcmp(cur_salt->mac, realmac) == 0)
  return 1; /* Password Found */

error:
return 0; /* Wrong Password */
}
```

For details see src/putty_fmt_plug.c and src/putty2john.c in JtR source tree. We compare the performance of PuTTY JtR plug-in on different machines below,

Figure 7: PuTTY cracking benchmarks



PuTTYs cracking benchmarks

```
$ ../run/john -fo:putty -t # AMD X3 720 (1 core)
Benchmarking: PuTTY Private Key SHA-1 / AES [32/64]... DONE
Only one salt:  154270 c/s real, 154270 c/s virtual

$ ../run/john -fo:putty -t # AMD FX-8120 (single core)
Benchmarking: PuTTY Private Key SHA-1 / AES [32/64]... DONE
Only one salt:  203400 c/s real, 203400 c/s virtual

$ ../run/john -fo:putty -t # AMD FX-8120 (8 cores)
Benchmarking: PuTTY Private Key SHA-1 / AES [32/64]... (8xOMP)
    DONE
Only one salt:  873984 c/s real, 109384 c/s virtual

$ ../run/john -fo:putty -t # Xeon E5420 (1 core)
Benchmarking: PuTTY Private Key SHA-1 / AES [32/64]... DONE
Only one salt:  131350 c/s real, 132664 c/s virtual

$ ../run/john -fo:putty -t # 2 x Xeon E5420 (8 cores)
Benchmarking: PuTTY Private Key SHA-1 / AES [32/64]... (8xOMP)
    DONE
Only one salt:  852992 c/s real, 106490 c/s virtua
```

Our work is the only multi-core capable PuTTY cracking software available. Cracking PuTTY password protected private keys can be accelerated by using GPU(s) to implement the KDF (key derivation function).

2.9 Analysis of Apple Legacy FileVault and Mac OS X disk image files

FileVault is a method of using encryption with volumes on Apple Mac computers which does encryption and decryption on the fly [35]. FileVault is used by password protected Mac OS X disk image files. Legacy FileVault supports two different header formats v1 and v2 with v1 being largely obsolete under modern Mac systems. Our JtR plug-in and security analysis of Apple Legacy FileVault and Mac OS X disk image files is an extension of the original research published in the VileFault paper[36]. However the information present in [36] is correct only for the v1 format. The source code published along [36]paper does not work for v2 format images nor for images using AES-256 encryption. We have fixed these shortcomings in our current work. This work was done in collaboration with Milen Rangelov (author of hashkill [37]).

The key used to encrypt data is encrypted ("wrapped") and stored in the header region of the disk image. Wrapping (encryption) of keys done using 3DES-EDE. Wrapped Key = 3DES-EDE(derived_key, IV, Actual Encryption Key) where derived_key = PBKDF2(salt, User Password, iterations). Be default, 1000 iterations are used and there is no option for changing this value.

Data blocks are encrypted in 4KiB "chunks" using AES-128 or AES-256 in CBC mode using the decrypted (un-wrapped) Wrapped Key. The IV is output of HMAC-SHA1 which takes the chunk number and Hmac-sha1 key read from the header. Encrypted Data Chunk = AES(Decrypted Wrapped Key, IV, chunkno, AES_ENCRYPT) where IV = trunc128 (HMAC-SHA1(hmac-key || chunkno). The following table describes the v2 file format used by Apple's Legacy FileVault.

Table 7: Legacy FileVault v2 format

Length	Name	Purpose
4 bytes	kdf_algorithm	Specifies PBKDF2 algorithm used
4 bytes	kdf_prng_algorithm	-
4 bytes	kdf_iteration_count	PBKDF2 iterations parameter (Currently 1000)
4 bytes	kdf_salt_len	Salt Length
32 bytes	kdf_salt	Salt Value
4 bytes	blob_enc_iv_size	Size of IV used while wrapping key
32 bytes	blob_enc_iv	IV used while wrapping key
4 bytes	blob_enc_key_bits	Key Size of Encryption Algorithm (Currently 168 bits)
4 bytes	blob_enc_algorithm	Encryption Algorithm Used (Currently 3DES)
4 bytes	blob_enc_padding	Specifies padding mode
4 bytes	blob_enc_mode	Encryption mode used (Currently CBC)
4 bytes	encrypted_keyblob_size	Size of Encrypted Key
48 bytes	encrypted_keyblob	Encrypted (using 3DES)Key

The following snippet demonstrates the PBKDF2 derived_key derivation from user password, Actual Encryption Key un-wrapping by using derived_key and decryption of encrypted data.

Code 9: dmg decryption

```
AES_KEY aes_decrypt_key;

/* derive dervied_key from user password, this is used in un-
    wrapping operation */
pbkdf2(UserPassword, strlen(UserPassword), salt, 20, 1000,
    derived_key);

/* decrypt encrypted_keyblob (the wrapped key), un-wrap operation
     */
DES_ede3_cbc_encrypt(encrypted_keyblob, decrypted_key,
    DES_key_schedule values..., IV, DES_DECRYPT);

/* un-wrapped key (decrypted_key) is used now for data decryption
 * Derive IV to be used in AES decryption based on chunk number
     to be decrypted */

memcpy(aes_key, decrypted_key, 32);
memcpy(hmacsha1_key, decrypted_key, 20);
HMAC_CTX_init(&ctx);
HMAC_Init_ex(&ctx, hmacsha1_key, 20, EVP_sha1(), NULL);
HMAC_Update(&ctx, ChunkNumber, 4);
HMAC_Final(&ctx, iv);

/* Decrypt chunk using derived iv and derived AES key */
AES_set_decrypt_key(aes_key, 128, &aes_decrypt_key);
AES_cbc_encrypt(chunk, data, data_size, &aes_decrypt_key, iv,
    AES_DECRYPT);

/* data now contains decrypted data */
```

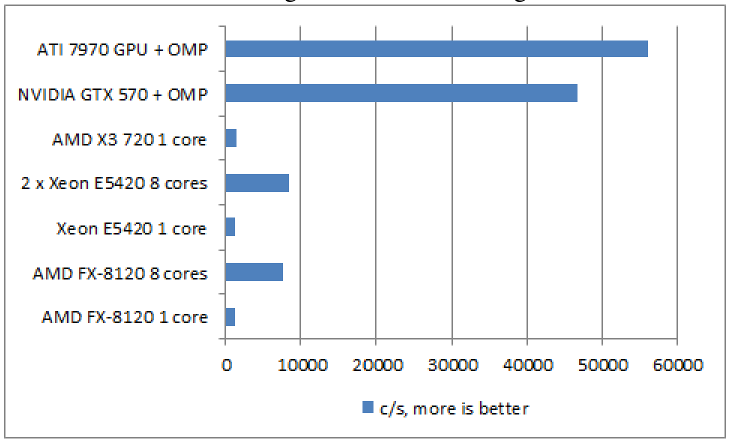
The original heuristics used in [36] to detect if the decryption happened successfully were totally wrong. We have identified a new set of proper heuristics which are capable of detecting where the data was decrypted successfully. The following snippet shows our new heuristics functions.

Code 10: dmg decryption heuristics

```c
/* a return code of 1 indicates a successful decryption */

int verify_decrytion(data)
{
    r = memmem(data, data_length, (void*)"koly", 4);
    if(r) {
        unsigned int *u32Version = (unsigned int *)(r + 4);
        if(HTONL(*u32Version) == 4)
            return 1;
    }
    if(memmem(data, data_length, (void*)"EFI␣PART", 8))
        return 1;
    if(memmem(data, data_length, (void*)"Apple", 5)) {
        return 1;
    if(memmem(data, data_length, (void*)"Press␣any␣key␣to␣reboot",
         23))
        return 1;
    return 0; /* incorrect decryption */
}
```

For details see src/dmg_fmt_plug.c and src/dmg2john.c in JtR source tree. We compare the performance of DMG JtR plug-in on different machines below,

Figure 8: DMG cracking benchmarks



We compare the performance of dmg JtR plug-in on different machines below, dmg cracking benchmarks

```
$ ../run/john -fo:dmg -t # AMD FX-8120 (single core)
Benchmarking: Apple DMG PBKDF2-HMAC-SHA-1 3DES / AES [32/64]...
    DONE
Raw:  1253 c/s real, 1266 c/s virtual

$../run/john -fo:dmg -t # AMD FX-8120 (8 cores)
Benchmarking: Apple DMG PBKDF2-HMAC-SHA-1 3DES / AES [32/64]...
    (8xOMP) DONE
Raw:  7603 c/s real, 955 c/s virtual

$ ../run/john -fo:dmg -t # Xeon E5420 (1 core)
Benchmarking: Apple DMG PBKDF2-HMAC-SHA-1 3DES / AES [32/64]...
    DONE
Raw:  1180 c/s real, 1180 c/s virtual

$ ../run/john -fo:dmg -t # 2 x Xeon E5420 (8 cores)
Benchmarking: Apple DMG PBKDF2-HMAC-SHA-1 3DES / AES [32/64]...
    (8xOMP) DONE
Raw:  8369 c/s real, 1171 c/s virtual

$ ../run/john -fo:dmg -t # AMD X3 720 (1 core)
Benchmarking: Apple DMG PBKDF2-HMAC-SHA-1 3DES / AES [32/64]...
    DONE
Raw:  1446 c/s real, 1446 c/s virtual

$ ../run/john -fo:dmg-opencl -t -pla=1 # ATI 7970
Benchmarking: Apple DMG PBKDF2-HMAC-SHA-1 3DES / AES [OpenCL]...
    (8xOMP) DONE
Raw:  56195 c/s real, 8844 c/s virtual

$ ../run/john -fo:dmg-opencl -t # NVIDIA GTX 570
Benchmarking: Apple DMG PBKDF2-HMAC-SHA-1 3DES / AES [OpenCL]...
    (8xOMP) DONE
Raw:  46747 c/s real, 8302 c/s virtual
```

2.10 Analysis of AES encrypted ZIP files

Zip is a popular file format used for data compression and archiving. Zip file format also supports encryption of data. The traditional encryption algorithm used in the Zip file format is weak and plenty of softwares exist to crack it. Support for strong AES encryption for ZIP archives was added in WinZip 9.0 which was released in year 2004. AES encryption as used in WinZip is described in [7]. For details about Zip file format, see APPNOTE.TXT [8] document. The following ZIP file format table only describes the fields relevant to password cracking.

Table 8: ZIP Encrypted file storage format

Salt	8 / 12 / 16 bytes	Salt size is dependent on Key Size
Password verification value	2 bytes	2 byte string used to validate correct password
Encrypted file data	Variable	Actual encrypted file data
Authentication code	10 bytes	Authentication code

The "salt" or "salt value" is a random or pseudo-random sequence of bytes that is combined with the encryption password to create encryption and authentication key. This two-byte value is produced as part of the process that derives the encryption and decryption keys from the password. When encrypting, a verification value is derived from the encryption password and stored with the encrypted file. Before decrypting, a verification value can be derived from the decryption password and compared to the value stored with the file, serving as a quick check that will detect most, but not all, incorrect passwords. There is a 1 in 65,536 chance that an incorrect password will yield a matching verification value; therefore, a matching verification value cannot be absolutely relied on to indicate a correct password. Encryption is applied only to the content of files. It is performed after compression, and not to any other associated data. The file data is encrypted byte-for-byte using the AES encryption algorithm operating in "CTR" mode [7].

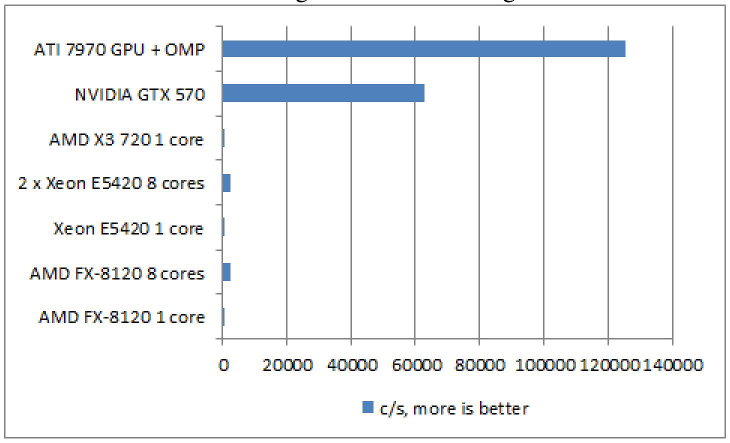
It is important to note that "Authentication code" is message authentication code (or MAC) of the data in the file after compression and encryption. Hence, it can't be used for verifying if the decryption happened correctly.

Code 11: AES ZIP cracking algorithm

```
AES_KEY aes_decrypt_key;

/* derive dervied_key from user password, this is used in un-
    wrapping operation */
pbkdf2(UserPassword, strlen(UserPassword), salt, 20, 1000,
    derived_key);

/* decrypt encrypted_keyblob (the wrapped key), un-wrap operation
    */
DES_ede3_cbc_encrypt(encrypted_keyblob, decrypted_key,
    DES_key_schedule values..., IV, DES_DECRYPT);

/* un-wrapped key (decrypted_key) is used now for data decryption
 * Derive IV to be used in AES decryption based on chunk number
    to be decrypted */

memcpy(aes_key, decrypted_key, 32);
memcpy(hmacsha1_key, decrypted_key, 20);
HMAC_CTX_init(&ctx);
HMAC_Init_ex(&ctx, hmacsha1_key, 20, EVP_sha1(), NULL);
HMAC_Update(&ctx, ChunkNumber, 4);
HMAC_Final(&ctx, iv);

/* Decrypt chunk using derived iv and derived AES key */
AES_set_decrypt_key(aes_key, 128, &aes_decrypt_key);
AES_cbc_encrypt(chunk, data, data_size, &aes_decrypt_key, iv,
    AES_DECRYPT);

/* data now contains decrypted data */
```

For more details see src/zip_fmt_plug.c and src/zip2john.c in JtR source tree. We compare the performance of ZIP JtR plug-in on different machines below,

Figure 9: ZIP cracking benchmarks



ZIP cracking benchmarks

```
$ ../run/john -fo:zip -t # AMD FX-8120 (single core)
Benchmarking: WinZip PBKDF2-HMAC-SHA-1 [32/64]... DONE
Raw:  538 c/s real, 538 c/s virtual

$../run/john -fo:zip -t # AMD FX-8120 (8 cores)
Benchmarking: WinZip PBKDF2-HMAC-SHA-1 [32/64]... (8xOMP) DONE
Raw:  2676 c/s real, 335 c/s virtual

$ ../run/john -fo:zip -t # Xeon E5420 (1 core)
Benchmarking: WinZip PBKDF2-HMAC-SHA-1 [32/64]... DONE
Raw:  328 c/s real, 328 c/s virtual

$ ../run/john -fo:zip -t # 2 x Xeon E5420 (8 cores)
Benchmarking: WinZip PBKDF2-HMAC-SHA-1 [32/64]... (8xOMP) DONE
Raw:  2566 c/s real, 321 c/s virtual

$ ../run/john -fo:zip -t # AMD X3 720 (1 core)
Benchmarking: WinZip PBKDF2-HMAC-SHA-1 [32/64]... DONE
Raw:  400 c/s real, 403 c/s virtual

$ ../run/john -fo:zip-opencl -t -pla=1 # ATI 7970
Benchmarking: ZIP-AES PBKDF2-HMAC-SHA-1 [OpenCL]... (8xOMP) DONE
Raw:  125672 c/s real, 2304K c/s virtual

$ ../run/john -fo:zip-opencl -t # NVIDIA GTX 570
Benchmarking: ZIP-AES PBKDF2-HMAC-SHA-1 [OpenCL]... (8xOMP) DONE
Raw:  62836 c/s real, 62552 c/s virtual
```

2.11 Analysis of PGP / GPG Secret Keys

Pretty Good Privacy (PGP) is a data encryption and decryption computer program that provides cryptographic privacy and authentication for data communication. PGP is often used for signing, encrypting and decrypting texts, e-mails, files, directories and whole disk partitions to increase the security of e-mail communications [2]. PGP and GnuPG follow the OpenPGP standard (RFC 4880) for encrypting and decrypting data. Our JtR plug-in and security analysis of PGP private key files is based on the original research done by Jonas Gehring (author of pgpry [4])

PGP secret keys can be password protected. The basic idea behind password protecting the secret key files is that even if the attacker has access to the secret key files, he won't be able to use them for gaining further access.

PGP / GPG use various custom key derivation functions with variable number of iterations to deter brute-force attacks. PGP calls its custom custom key derivation functions as string-to-key (s2k) functions. The various s2k functions vary a lot in their speed and resistance to brute-force attacks. A weak s2k function is shown in the snippet below, PGP custom KDF

```
S2KSimpleMD5Generator(char *password, unsigned char *key, int
    length)
{
    MD5_CTX ctx;
    uint32_t numHashes = (length + MD5_DIGEST_LENGTH - 1) /
        MD5_DIGEST_LENGTH;
    int i, j;

    for (i = 0; i < numHashes; i++) {
        MD5_Init(&ctx);
        for (j = 0; j < i; j++) {
            MD5_Update(&ctx, "\0", 1);
        }
        MD5_Update(&ctx, password, strlen(password));
        MD5_Final(key + (i * MD5_DIGEST_LENGTH), &ctx);
    }
}
```

It is possible to mount time-memory trade-off attacks against such simple s2k functions due to lack of any salting. This allows cracking of password protected private key files at very high speeds. Such weak s2k functions are no longer used even in the default configuration of GPG. . The default s2k function is shown below, PGP custom KDF

```
S2KItSaltedSHA1Generator(char *password, unsigned char *key, int
    length)
{
    unsigned char keybuf[KEYBUFFER_LENGTH];
    SHA_CTX ctx;
    int i, j;
    int32_t tl;
    int32_t mul;
    int32_t bs;
    uint8_t *bptr;
    int32_t n;

    uint32_t numHashes = (length + SHA_DIGEST_LENGTH - 1) /
        SHA_DIGEST_LENGTH;
    memcpy(keybuf, cur_salt->salt, 8);

    for (i = 0; i < numHashes; i++) {
        SHA1_Init(&ctx);
        for (j = 0; j < i; j++) {
            SHA1_Update(&ctx, "\0", 1);
        }
        // Find multiplicator
        tl = strlen(password) + 8;
        mul = 1;
        while (mul < tl && ((64 * mul) % tl)) {
            ++mul;
        }
        // Try to feed the hash function with 64-byte blocks
        bs = mul * 64;
        bptr = keybuf + tl;
        n = bs / tl;
        memcpy(keybuf + 8, password, strlen(password));
        while (n-- > 1) {
            memcpy(bptr, keybuf, tl);
            bptr += tl;
        }
        n = cur_salt->count / bs;
        while (n-- > 0) {
            SHA1_Update(&ctx, keybuf, bs);
        }
        SHA1_Update(&ctx, keybuf, cur_salt->count % bs);
        SHA1_Final(key + (i * SHA_DIGEST_LENGTH), &ctx);
    }
}
```

We have written a custom cracker for GPG secret key files and following snippet shows the main steps involved,

Code 12: PGP Cracker

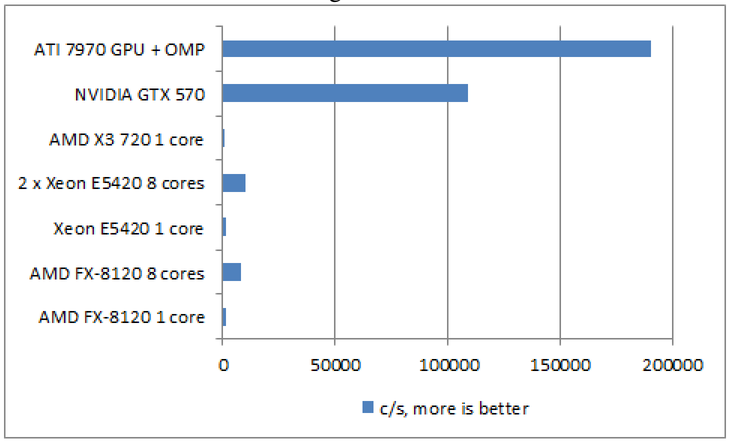
```
/* derive decryption key from PASSWORD and SALT using custom KDF
     */

S2KItSaltedSHA1Generator(char *password, unsigned char *key, int
     length);

S2KItSaltedSHA1Generator(PASSWORD, keydata, KEY_SIZE);

/* decrypt encrypted key material */

// Decrypt first data block in order to check the first two bits
     of
// the MPI. If they are correct, there's a good chance that the
// password is correct, too.

unsigned char ivec[32];
unsigned char out[4096];
int tmp = 0;
uint32_t num_bits;
int checksumOk;
int i;

// Quick Hack
memcpy(ivec, cur_salt->iv, blockSize(cur_salt->cipher_algorithm));
CAST_KEY ck;
CAST_set_key(&ck, ks, keydata);
CAST_cfb64_encrypt(cur_salt->data, out, CAST_BLOCK, &ck, ivec, &
     tmp, CAST_DECRYPT);

num_bits = ((out[0] << 8) | out[1]);
if (num_bits < MIN_BN_BITS || num_bits > cur_salt->bits) {
   return 0;
}

// Decrypt all data
memcpy(ivec, cur_salt->iv, blockSize(cur_salt->cipher_algorithm));
tmp = 0;
CAST_KEY ck;
CAST_set_key(&ck, ks, keydata);
CAST_cfb64_encrypt(cur_salt->data, out, cur_salt->datalen, &ck,
     ivec, &tmp, CAST_DECRYPT);

// Verify
checksumOk = 0;
uint8_t checksum[SHA_DIGEST_LENGTH];
SHA_CTX ctx;
SHA1_Init(&ctx);
SHA1_Update(&ctx, out, cur_salt->datalen - SHA_DIGEST_LENGTH);
SHA1_Final(checksum, &ctx);
if (memcmp(checksum, out + cur_salt->datalen - SHA_DIGEST_LENGTH,
     SHA_DIGEST_LENGTH) == 0) {
   checksumOk = 1;
}

// If the checksum is ok, try to parse the first MPI of the
     private key
if (checksumOk) {
   BIGNUM *b = NULL;
   uint32_t blen = (num_bits + 7) / 8;
   if (blen < cur_salt->datalen && ((b = BN_bin2bn(out + 2, blen,
        NULL)) != NULL)) {
      BN_free(b);
      return 1;
   }
}

return 0;

```

For details see src/gpg_fmt_plug.c and src/gpg2john.cpp in JtR source tree.

The key technique (through which we gain a speed-up) is that we only do partial decryption (1 block) of encrypted key material and then check. <FIXME>

Our CPU version of the cracking software achieves around 896 c/s on a single core and 7097 c/s on 2 x Xeon E5420 (8 cores total). The GPU versionachieves a speedup of around 97x over single core CPU result. Currently, the GPU implementation transfers candidate passwords from CPU to GPU which is sub-optimal. Future version of JtR will remove this limitation and higher cracking speeds can be expected. We compare the performance of PGP JtR plug-in on different machines below,

Figure 10: GPG Benchmarks



ZIP cracking benchmarks

```
$ ../run/john -fo:agilekeychain -t # AMD FX-8120 (single core)
Benchmarking: 1Password Agile Keychain PBKDF2-HMAC-SHA-1 AES
    [32/64]... DONE
Raw:  1327 c/s real, 1327 c/s virtual

$../run/john -fo:agilekeychain -t # AMD FX-8120 (8 cores)
Benchmarking: 1Password Agile Keychain PBKDF2-HMAC-SHA-1 AES
    [32/64]... (8xOMP) DONE
Raw:  7953 c/s real, 997 c/s virtual

$ ../run/john -fo:agilekeychain -t # Xeon E5420 (1 core)
Benchmarking: 1Password Agile Keychain PBKDF2-HMAC-SHA-1 AES
    [32/64]... DONE
Raw:  1247 c/s real, 1247 c/s virtual

$ ../run/john -fo:agilekeychain -t # 2 x Xeon E5420 (8 cores)
Benchmarking: 1Password Agile Keychain PBKDF2-HMAC-SHA-1 AES
    [32/64]... (8xOMP) DONE
Raw:  9941 c/s real, 1244 c/s virtual

$ ../run/john -fo:agilekeychain -t # AMD X3 720 (1 core)
Benchmarking: 1Password Agile Keychain PBKDF2-HMAC-SHA-1 AES
    [32/64]... DONE
Raw:  1536 c/s real, 1536 c/s virtual

$ ../run/john -fo:agilekeychain-opencl -t -pla=1 # ATI 7970
Benchmarking: 1Password Agile Keychain PBKDF2-HMAC-SHA-1 AES [
    OpenCL]... (8xOMP) DONE
Raw:  190464 c/s real, 72511 c/s virtual

$ ../run/john -fo:agilekeychain-opencl -t # NVIDIA GTX 570
Benchmarking: 1Password Agile Keychain PBKDF2-HMAC-SHA-1 AES [
    OpenCL]... (8xOMP) DONE
Raw:  108850 c/s real, 46234 c/s virtual
```

Cracking password protected GPG private key files is extensively analyzed in [5]. However, Milo et al, haven't released any source code demonstrating the speedups they claim in the paper and asking for more information about testing enviornment resulted in no answer.

2.12 Analysis of EncFS

1Password is a popular password manager available for Windows, iPad, iPhone, Android and Mac platforms. 1Password uses a file format (called Agile Keychain format) which is different from Apple's Keychain file format. The goal of the Agile Keychain file is to build on the successes of the Mac OS X keychain while increasing the flexibility and portability of the keychain design [20]. 1Password stores its data in a folder called "1Password.agilekeychain". 1Password uses JSON (JavaScript Object Notation) format to store its data which has a benefit that its files can be loaded directly into a web browser. It is possible to access the data, without installing 1Password software

, by using a web browser. Our JtR plug-in and security analysis of Agile Keychain is an extension of the original research done by Antonin Amand (author of agilekeychain [16])

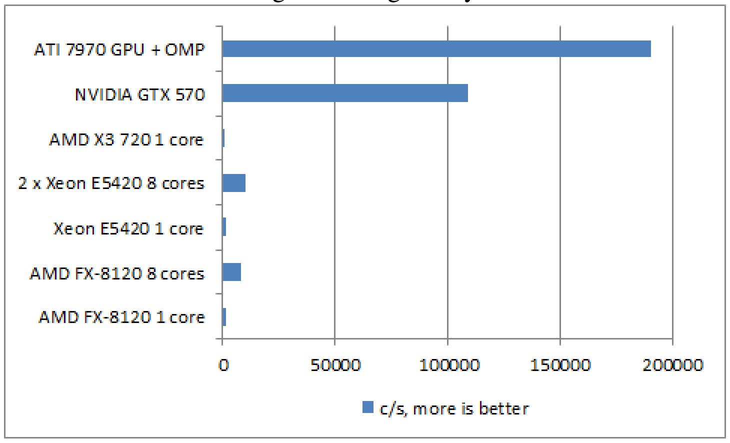
The core of the encryption is AES (Advanced Encryption Standard) using 128-bit encryption keys and performed in Cipher Block Chaining (CBC) mode along with a randomized Initialization Vector. Instead of encrypting data with the password directly, a random key of 1024 bytes is used. This key is stored in the encryptionKeys.js file, encrypted using a key derived from the users master password by using PBKDF2 function. A sample encryptionKeys.js is shown below,

Sample encryptionKeys.js

```
{"list":[{"data":"U2FsdGVkX19xRuqhzKOV5efr...","validation":"
    U2FsdGVkX19dIEp7VK09LOf...",identifier":"1
    D169F66FAAC4745A4C708B254944791","level":"SL5","iterations"
    :1000}

SALT␣=␣identifier's␣value

User␣Encryption␣Key␣=␣PBKDF2-HMAC-SHA(PASSWORD,␣SALT,␣iterations,
    ␣16)
```

We have written a Python program (run/agilekc2john.py) which parses Agile Keychain data and generates a “hash” which is understood by JtR. 1Password uses PKCS#7 padding for wrapping the random encryption key. We exploit this padding knowledge to figure out if we have successfully decrypted the radom encryption key.

Figure 11: Agile Keychain Benchmarks



Agile Keychain Cracker

```
/* CIPHERTEXT is encrypted random key */

#define CTLEN 1040

unsigned int master[8];
pbkdf2(PASSWORD, strlen(PASSWORD), SALT, SALTLEN, iterations,
     master);
AES_KEY akey;
AES_set_decrypt_key(master, 128, &akey);
AES_cbc_encrypt(CIPHERTEXT, out, CTLEN, &akey, iv, AES_DECRYPT);

// now check padding
pad = out[CTLEN - 1];
if(pad < 1 || pad > 16) /* AES block size is 128 bits = 16 bytes
     */
   // "Bad padding byte. You probably have a wrong password"
   return -1;

n = CTLEN - pad;
key_size = n / 8;

if(key_size != 128 && key_size != 192 && key_size != 256)
   // "invalid key size"
   return -1;
for(i = n; i < CTLEN; i++)
   if(out[i] != pad)
      // "Bad padding. You probably have a wrong password"
      return -1;

// padding check passed, password found!
printf("Password␣Found");
```

Agile Keychain Benchmarks

```
$ ../run/john -fo:agilekeychain -t # AMD FX-8120 (single core)
Benchmarking: 1Password Agile Keychain PBKDF2-HMAC-SHA-1 AES
    [32/64]... DONE
Raw:  1327 c/s real, 1327 c/s virtual

$../run/john -fo:agilekeychain -t # AMD FX-8120 (8 cores)
Benchmarking: 1Password Agile Keychain PBKDF2-HMAC-SHA-1 AES
    [32/64]... (8xOMP) DONE
Raw:  7953 c/s real, 997 c/s virtual

$ ../run/john -fo:agilekeychain -t # Xeon E5420 (1 core)
Benchmarking: 1Password Agile Keychain PBKDF2-HMAC-SHA-1 AES
    [32/64]... DONE
Raw:  1247 c/s real, 1247 c/s virtual

$ ../run/john -fo:agilekeychain -t # 2 x Xeon E5420 (8 cores)
Benchmarking: 1Password Agile Keychain PBKDF2-HMAC-SHA-1 AES
    [32/64]... (8xOMP) DONE
Raw:  9941 c/s real, 1244 c/s virtual

$ ../run/john -fo:agilekeychain -t # AMD X3 720 (1 core)
Benchmarking: 1Password Agile Keychain PBKDF2-HMAC-SHA-1 AES
    [32/64]... DONE
Raw:  1536 c/s real, 1536 c/s virtual

$ ../run/john -fo:agilekeychain-opencl -t -pla=1 # ATI 7970
Benchmarking: 1Password Agile Keychain PBKDF2-HMAC-SHA-1 AES [
    OpenCL]... (8xOMP) DONE
Raw:  190464 c/s real, 72511 c/s virtual

$ ../run/john -fo:agilekeychain-opencl -t # NVIDIA GTX 570
Benchmarking: 1Password Agile Keychain PBKDF2-HMAC-SHA-1 AES [
    OpenCL]... (8xOMP) DONE
Raw:  108850 c/s real, 46234 c/s virtual
```

In our opinion, the default number of iterations (1,000) should be increased for added security against brute-force attacks. It is trivial to do so by increasing the value of "iterations" parmater in encryptionKey.ks file. Our cracker is the only known cracker for Agile Keychain files. Agile Keychain design has one flaw that it doesn't encrypt and protect the metadata (like URL) for a given password. This opens up another attack vector against 1Password software.

Agile Keychain metadata flaw

```
{"uuid":"23F59720EA334163AF","updatedAt":1257723121,"locationKey"
    :"github.com","openContents":{"usernameHash":"11
    dfdb7483af4c2531","passwordStrength":72,"contentsHash":"41716
    f8f","passwordHash":"f48491cd25d4061233d6"},"keyID":"978
    B7BA055427B5E77B","title":"Github","location":"https://github
    .com/session","encrypted":"U20000","createdAt":1257723121,"
    typeName":"webforms.WebForm"}
```

It was amusing to see our cracker being tested and blogged about by official 1Password developer Jeffrey Goldberg [21].

2.13 Analysis of Microsoft Office file formats

2.13.1 Analysis of Outlook (97-2013) pst files

Personal Storage Table (PST) is an open, proprietary file format used to store messages, calendar events, and other items within Microsoft software such as Microsoft Exchange Client, Windows Messaging, and Microsoft Outlook [6]. Password protection can be used to protect the content of the PST files. However, even Microsoft itself admits that the password adds very little protection, due to the existence of commonly available tools which can remove or simply bypass the password protection. The password to access the table is stored itself in the PST file. Outlook checks to make sure that it matches the user-specified password and refuses to operate if there is no match.

PST is a complex files format

The data is readable by the libpst project code.

Microsoft (MS) offers three values for the encryption setting: none, compressible, and high.

None the PST data is stored as plain text. Compressible the PST data is encrypted with a byte-substitution cipher with a fixed substitution table. High (sometimes called "better") encryption is similar to a WWII German Enigma cipher with three fixed rotors.

Note that neither of the two encryption modes uses the user-specified password as any part of the key for the encryption.

http://linux.die.net/man/5/outlook.pst

The following item types are known, but not all of these are implemented in the code yet.

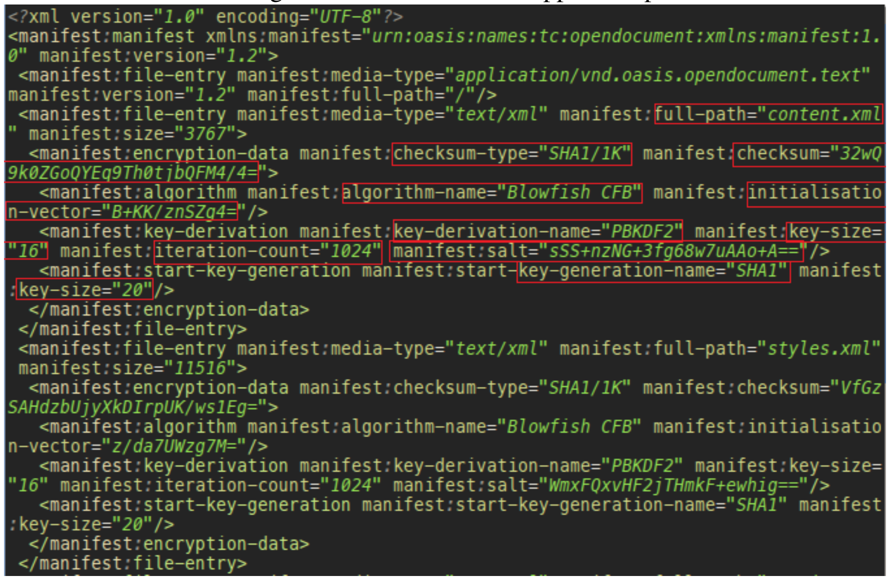
0x67ff Password checksum,

CRC algorithm : http://msdn.microsoft.com/en-us/library/ff385753%28v=office.12%29

http://www.passcape.com/outlook_passwords#b2

The Open Document Format for Office Applications (ODF), also known as OpenDocument (OD), is an XML-based file format for spreadsheets, charts, presentations and word processing documents. Our work is the first open-source multi-core cracking software for ODF files. Uses PBKDF2. OpenDocument files can also take the format of a ZIP compressed archive containing a number of files and directories; these can contain binary content and benefit from ZIP's lossless compression to reduce.

Figure 12: manifest.xml snipped sample



```
<?xml version="1.0" encoding="UTF-8"?>
<manifest:manifest xmlns:manifest="urn:oasis:names:tc:opendocument:xmlns:manifest:1.
0" manifest:version="1.2">
 <manifest:file-entry manifest:media-type="application/vnd.oasis.opendocument.text"
manifest:version="1.2" manifest:full-path="/"/>
 <manifest:file-entry manifest:media-type="text/xml" manifest:full-path="content.xml
" manifest:size="3767">
  <manifest:encryption-data manifest:checksum-type="SHA1/1K" manifest:checksum="32wQ
9k0ZGoQYEq9Th0tjbQFM4/4=">
   <manifest:algorithm manifest:algorithm-name="Blowfish CFB" manifest:initialisatio
n-vector="B+KK/znSZg4="/>
   <manifest:key-derivation manifest:key-derivation-name="PBKDF2" manifest:key-size=
"16" manifest:iteration-count="1024" manifest:salt="sSS+nzNG+3fg68w7uAAo+A=="/>
   <manifest:start-key-generation manifest:start-key-generation-name="SHA1" manifest
:key-size="20"/>
  </manifest:encryption-data>
 </manifest:file-entry>
 <manifest:file-entry manifest:media-type="text/xml" manifest:full-path="styles.xml"
 manifest:size="11516">
  <manifest:encryption-data manifest:checksum-type="SHA1/1K" manifest:checksum="VfGz
SAHdzbUjyXkDIrpUK/ws1Eg=">
   <manifest:algorithm manifest:algorithm-name="Blowfish CFB" manifest:initialisatio
n-vector="z/da7UWzg7M="/>
   <manifest:key-derivation manifest:key-derivation-name="PBKDF2" manifest:key-size=
"16" manifest:iteration-count="1024" manifest:salt="WmxFQxvHF2jTHmkF+ewhig=="/>
   <manifest:start-key-generation manifest:start-key-generation-name="SHA1" manifest
:key-size="20"/>
  </manifest:encryption-data>
 </manifest:file-entry>
```

Benchmarks,

ODF cracking benchmarks

```
$../run/john -fo:odf -t # AMD FX-8120 (single core)
Benchmarking: ODF SHA-1 Blowfish [32/64]... DONE
Raw:  1189 c/s real, 1189 c/s virtual

RETAKE $../run/john -fo:odf -t # AMD FX-8120 (8 cores)
Benchmarking: ODF SHA-1 Blowfish [32/64]... (8xOMP) DONE
Raw:  5263 c/s real, 865 c/s virtual

$ ../run/john -fo:odf -t # Xeon E5420 (1 core)

$ ../run/john -fo:odf -t # 2 x Xeon E5420 (8 cores)

# GPU TODO

$ ../run/john -fo:pwsafe-opencl -t # ATI 6970 GPU
Benchmarking: Password Safe SHA-256 [OpenCL]... DONE
Raw:  11815 c/s real, 768000 c/s virtual

$ ../run/john -fo:pwsafe-opencl -t # GeForce GTX 570 OpenCL
Benchmarking: Password Safe SHA-256 [OpenCL]... DONE
Raw:  27131 c/s real, 27131 c/s virtual

$ ../run/john -fo:pwsafe-cuda -t # GeForce GTX 570 CUDA
Benchmarking: Password Safe SHA-256 [CUDA]... DONE
Raw:  107185 c/s real, 107185 c/s virtual
```

http://cpan.uwinnipeg.ca/htdocs/Spreadsheet-ParseExcel/Spreadsheet/ParseExcel.pm.html#Decryption
http://www.password-crackers.com/blog/?p=16
http://www.password-crackers.com/en/articles/12/#II

2.13.2 Guaranteed decryption of Office files using 40-bit RC4 encryption

The Open Document Format for Office Applications (ODF), also known as OpenDocument (OD), is an XML-based file format for spreadsheets, charts, presentations and word processing documents. Our work is the first open-source multi-core cracking software for ODF files. Uses PBKDF2. OpenDocument files can also take the format of a ZIP compressed archive containing a number of files and directories; these can contain binary content and benefit from ZIP's lossless compression to reduce Benchmarks,

ODF cracking benchmarks

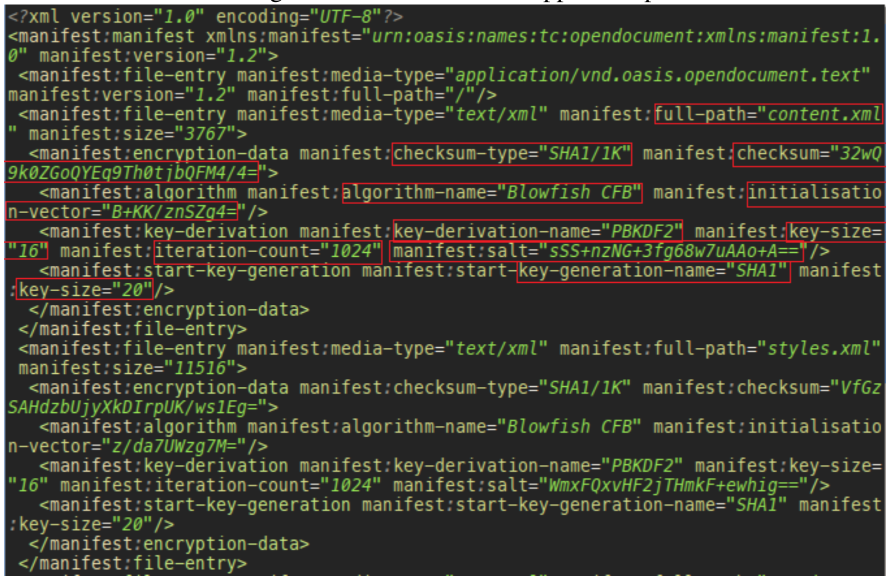
```
$../run/john -fo:odf -t # AMD FX-8120 (single core)
Benchmarking: ODF SHA-1 Blowfish [32/64]... DONE
Raw:  1189 c/s real, 1189 c/s virtual

RETAKE $../run/john -fo:odf -t # AMD FX-8120 (8 cores)
Benchmarking: ODF SHA-1 Blowfish [32/64]... (8xOMP) DONE
Raw:  5263 c/s real, 865 c/s virtual

$ ../run/john -fo:odf -t # Xeon E5420 (1 core)

$ ../run/john -fo:odf -t # 2 x Xeon E5420 (8 cores)

# GPU TODO

$ ../run/john -fo:pwsafe-opencl -t # ATI 6970 GPU
Benchmarking: Password Safe SHA-256 [OpenCL]... DONE
Raw:  11815 c/s real, 768000 c/s virtual

$ ../run/john -fo:pwsafe-opencl -t # GeForce GTX 570 OpenCL
Benchmarking: Password Safe SHA-256 [OpenCL]... DONE
Raw:  27131 c/s real, 27131 c/s virtual

$ ../run/john -fo:pwsafe-cuda -t # GeForce GTX 570 CUDA
Benchmarking: Password Safe SHA-256 [CUDA]... DONE
Raw:  107185 c/s real, 107185 c/s virtual
```

2.13.3 Analysis of Office 2003 file encryption

2.13.4 Analysis of Office 2007 file encryption

The Open Document Format for Office Applications (ODF), also known as OpenDocument (OD), is an XML-based file format for spreadsheets, charts, presentations and word processing documents. Our work is the first open-source multi-core cracking software for ODF files. Uses PBKDF2. OpenDocument files can also take the format of a ZIP compressed archive containing a number of files and directories; these can contain binary content and benefit from ZIP's lossless compression to reduce.

Figure 13: manifest.xml snipped sample



```
<?xml version="1.0" encoding="UTF-8"?>
<manifest:manifest xmlns:manifest="urn:oasis:names:tc:opendocument:xmlns:manifest:1.
0" manifest:version="1.2">
 <manifest:file-entry manifest:media-type="application/vnd.oasis.opendocument.text"
manifest:version="1.2" manifest:full-path="/"/>
 <manifest:file-entry manifest:media-type="text/xml" manifest:full-path="content.xml
" manifest:size="3767">
  <manifest:encryption-data manifest:checksum-type="SHA1/1K" manifest:checksum="32wQ
9k0ZGoQYEq9Th0tjbQFM4/4=">
   <manifest:algorithm manifest:algorithm-name="Blowfish CFB" manifest:initialisatio
n-vector="B+KK/znSZg4="/>
   <manifest:key-derivation manifest:key-derivation-name="PBKDF2" manifest:key-size=
"16" manifest:iteration-count="1024" manifest:salt="sSS+nzNG+3fg68w7uAAo+A=="/>
   <manifest:start-key-generation manifest:start-key-generation-name="SHA1" manifest
:key-size="20"/>
  </manifest:encryption-data>
 </manifest:file-entry>
 <manifest:file-entry manifest:media-type="text/xml" manifest:full-path="styles.xml"
 manifest:size="11516">
  <manifest:encryption-data manifest:checksum-type="SHA1/1K" manifest:checksum="VfGz
SAHdzbUjyXkDIrpUK/ws1Eg=">
   <manifest:algorithm manifest:algorithm-name="Blowfish CFB" manifest:initialisatio
n-vector="z/da7UWzg7M="/>
   <manifest:key-derivation manifest:key-derivation-name="PBKDF2" manifest:key-size=
"16" manifest:iteration-count="1024" manifest:salt="WmxFQxvHF2jTHmkF+ewhig=="/>
   <manifest:start-key-generation manifest:start-key-generation-name="SHA1" manifest
:key-size="20"/>
  </manifest:encryption-data>
 </manifest:file-entry>
```

Benchmarks,

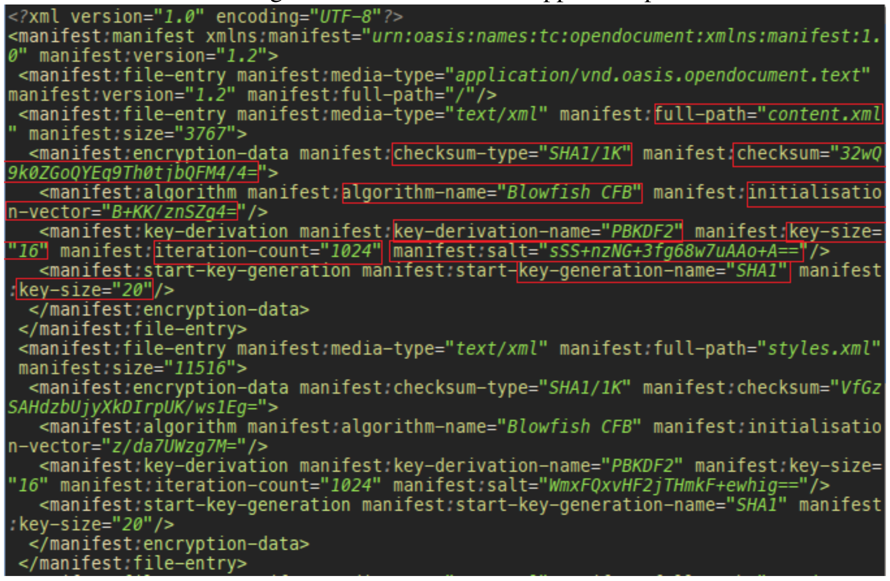
ODF cracking benchmarks

```
$../run/john -fo:odf -t # AMD FX-8120 (single core)
Benchmarking: ODF SHA-1 Blowfish [32/64]... DONE
Raw:  1189 c/s real, 1189 c/s virtual

RETAKE $../run/john -fo:odf -t # AMD FX-8120 (8 cores)
Benchmarking: ODF SHA-1 Blowfish [32/64]... (8xOMP) DONE
Raw:  5263 c/s real, 865 c/s virtual

$ ../run/john -fo:odf -t # Xeon E5420 (1 core)

$ ../run/john -fo:odf -t # 2 x Xeon E5420 (8 cores)

# GPU TODO

$ ../run/john -fo:pwsafe-opencl -t # ATI 6970 GPU
Benchmarking: Password Safe SHA-256 [OpenCL]... DONE
Raw:  11815 c/s real, 768000 c/s virtual

$ ../run/john -fo:pwsafe-opencl -t # GeForce GTX 570 OpenCL
Benchmarking: Password Safe SHA-256 [OpenCL]... DONE
Raw:  27131 c/s real, 27131 c/s virtual

$ ../run/john -fo:pwsafe-cuda -t # GeForce GTX 570 CUDA
Benchmarking: Password Safe SHA-256 [CUDA]... DONE
Raw:  107185 c/s real, 107185 c/s virtual
```

2.13.5 Analysis of Office 2010 file encryption

The Open Document Format for Office Applications (ODF), also known as OpenDocument (OD), is an XML-based file format for spreadsheets, charts, presentations and word processing documents. Our work is the first open-source multi-core cracking software for ODF files. Uses PBKDF2. OpenDocument files can also take the format of a ZIP compressed archive containing a number of files and directories; these can contain binary content and benefit from ZIP's lossless compression to reduce.

Figure 14: manifest.xml snipped sample



```
<?xml version="1.0" encoding="UTF-8"?>
<manifest:manifest xmlns:manifest="urn:oasis:names:tc:opendocument:xmlns:manifest:1.
0" manifest:version="1.2">
 <manifest:file-entry manifest:media-type="application/vnd.oasis.opendocument.text"
manifest:version="1.2" manifest:full-path="/"/>
 <manifest:file-entry manifest:media-type="text/xml" manifest:full-path="content.xml
" manifest:size="3767">
  <manifest:encryption-data manifest:checksum-type="SHA1/1K" manifest:checksum="32wQ
9k0ZGoQYEq9Th0tjbQFM4/4=">
   <manifest:algorithm manifest:algorithm-name="Blowfish CFB" manifest:initialisatio
n-vector="B+KK/znSZg4="/>
   <manifest:key-derivation manifest:key-derivation-name="PBKDF2" manifest:key-size=
"16" manifest:iteration-count="1024" manifest:salt="sSS+nzNG+3fg68w7uAAo+A=="/>
   <manifest:start-key-generation manifest:start-key-generation-name="SHA1" manifest
:key-size="20"/>
  </manifest:encryption-data>
 </manifest:file-entry>
 <manifest:file-entry manifest:media-type="text/xml" manifest:full-path="styles.xml"
 manifest:size="11516">
  <manifest:encryption-data manifest:checksum-type="SHA1/1K" manifest:checksum="VfGz
SAHdzbUjyXkDIrpUK/ws1Eg=">
   <manifest:algorithm manifest:algorithm-name="Blowfish CFB" manifest:initialisatio
n-vector="z/da7UWzg7M="/>
   <manifest:key-derivation manifest:key-derivation-name="PBKDF2" manifest:key-size=
"16" manifest:iteration-count="1024" manifest:salt="WmxFQxvHF2jTHmkF+ewhig=="/>
   <manifest:start-key-generation manifest:start-key-generation-name="SHA1" manifest
:key-size="20"/>
  </manifest:encryption-data>
 </manifest:file-entry>
```

Benchmarks,

ODF cracking benchmarks

```
$../run/john -fo:odf -t # AMD FX-8120 (single core)
Benchmarking: ODF SHA-1 Blowfish [32/64]... DONE
Raw:  1189 c/s real, 1189 c/s virtual

RETAKE $../run/john -fo:odf -t # AMD FX-8120 (8 cores)
Benchmarking: ODF SHA-1 Blowfish [32/64]... (8xOMP) DONE
Raw:  5263 c/s real, 865 c/s virtual

$ ../run/john -fo:odf -t # Xeon E5420 (1 core)

$ ../run/john -fo:odf -t # 2 x Xeon E5420 (8 cores)

# GPU TODO

$ ../run/john -fo:pwsafe-opencl -t # ATI 6970 GPU
Benchmarking: Password Safe SHA-256 [OpenCL]... DONE
Raw:  11815 c/s real, 768000 c/s virtual

$ ../run/john -fo:pwsafe-opencl -t # GeForce GTX 570 OpenCL
Benchmarking: Password Safe SHA-256 [OpenCL]... DONE
Raw:  27131 c/s real, 27131 c/s virtual

$ ../run/john -fo:pwsafe-cuda -t # GeForce GTX 570 CUDA
Benchmarking: Password Safe SHA-256 [CUDA]... DONE
Raw:  107185 c/s real, 107185 c/s virtual
```

Construct 2 file format tables from office2john.c.

2.13.6 Analysis of Office 2013 file encryption

The Open Document Format for Office Applications (ODF), also known as OpenDocument (OD), is an XML-based file format for spreadsheets, charts, presentations and word processing documents. Our work is the first open-source multi-core cracking software for ODF files. Uses PBKDF2. OpenDocument files can also take the format of a ZIP compressed archive containing a number of files and directories; these can contain binary content and benefit from ZIP's lossless compression to reduce.

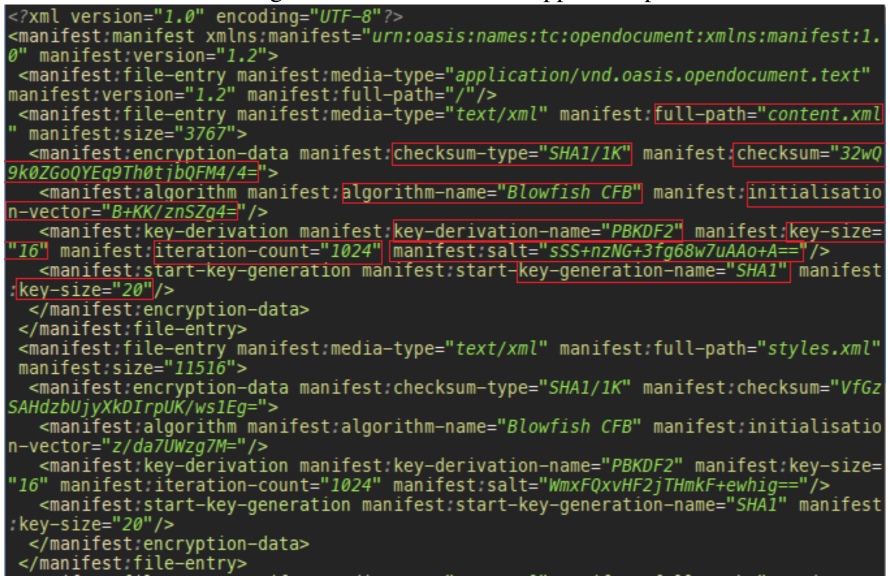


Figure 15: manifest.xml snipped sample

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest:manifest xmlns:manifest="urn:oasis:names:tc:opendocument:xmlns:manifest:1.
0" manifest:version="1.2">
 <manifest:file-entry manifest:media-type="application/vnd.oasis.opendocument.text"
manifest:version="1.2" manifest:full-path="/"/>
 <manifest:file-entry manifest:media-type="text/xml" manifest:full-path="content.xml
" manifest:size="3767">
  <manifest:encryption-data manifest:checksum-type="SHA1/1K" manifest:checksum="32wQ
9k0ZGoQYEq9Th0tjbQFM4/4=">
   <manifest:algorithm manifest:algorithm-name="Blowfish CFB" manifest:initialisatio
n-vector="B+KK/znSZg4="/>
   <manifest:key-derivation manifest:key-derivation-name="PBKDF2" manifest:key-size=
"16" manifest:iteration-count="1024" manifest:salt="sSS+nzNG+3fg68w7uAAo+A=="/>
   <manifest:start-key-generation manifest:start-key-generation-name="SHA1" manifest
:key-size="20"/>
  </manifest:encryption-data>
 </manifest:file-entry>
 <manifest:file-entry manifest:media-type="text/xml" manifest:full-path="styles.xml"
 manifest:size="11516">
  <manifest:encryption-data manifest:checksum-type="SHA1/1K" manifest:checksum="VfGz
SAHdzbUjyXkDIrpUK/ws1Eg=">
   <manifest:algorithm manifest:algorithm-name="Blowfish CFB" manifest:initialisatio
n-vector="z/da7UWzg7M="/>
   <manifest:key-derivation manifest:key-derivation-name="PBKDF2" manifest:key-size=
"16" manifest:iteration-count="1024" manifest:salt="WmxFQxvHF2jTHmkF+ewhig=="/>
   <manifest:start-key-generation manifest:start-key-generation-name="SHA1" manifest
:key-size="20"/>
  </manifest:encryption-data>
 </manifest:file-entry>
```

Benchmarks,

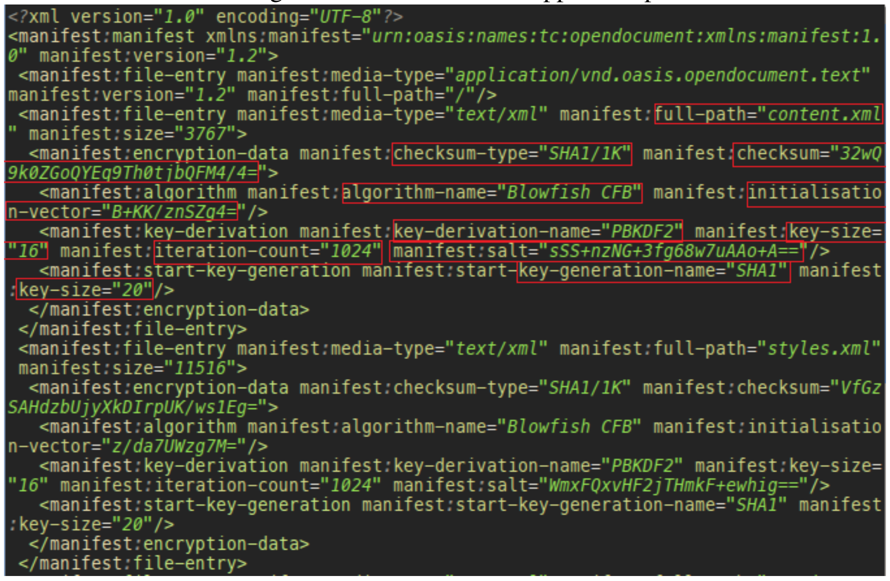
ODF cracking benchmarks

```
$../run/john -fo:odf -t # AMD FX-8120 (single core)
Benchmarking: ODF SHA-1 Blowfish [32/64]... DONE
Raw:  1189 c/s real, 1189 c/s virtual

RETAKE $../run/john -fo:odf -t # AMD FX-8120 (8 cores)
Benchmarking: ODF SHA-1 Blowfish [32/64]... (8xOMP) DONE
Raw:  5263 c/s real, 865 c/s virtual

$ ../run/john -fo:odf -t # Xeon E5420 (1 core)

$ ../run/john -fo:odf -t # 2 x Xeon E5420 (8 cores)

# GPU TODO

$ ../run/john -fo:pwsafe-opencl -t # ATI 6970 GPU
Benchmarking: Password Safe SHA-256 [OpenCL]... DONE
Raw:  11815 c/s real, 768000 c/s virtual

$ ../run/john -fo:pwsafe-opencl -t # GeForce GTX 570 OpenCL
Benchmarking: Password Safe SHA-256 [OpenCL]... DONE
Raw:  27131 c/s real, 27131 c/s virtual

$ ../run/john -fo:pwsafe-cuda -t # GeForce GTX 570 CUDA
Benchmarking: Password Safe SHA-256 [CUDA]... DONE
Raw:  107185 c/s real, 107185 c/s virtual
```

2.14 Analysis of OpenOffice / LibreOffice file format

The Open Document Format for Office Applications (ODF), also known as OpenDocument (OD), is an XML-based file format for spreadsheets, charts, presentations and word processing documents. Our work is the first open-source multi-core cracking software for ODF files. Uses PBKDF2. OpenDocument files can also take the format of a ZIP compressed archive containing a number of files and directories; these can contain binary content and benefit from ZIP's lossless compression to reduce.

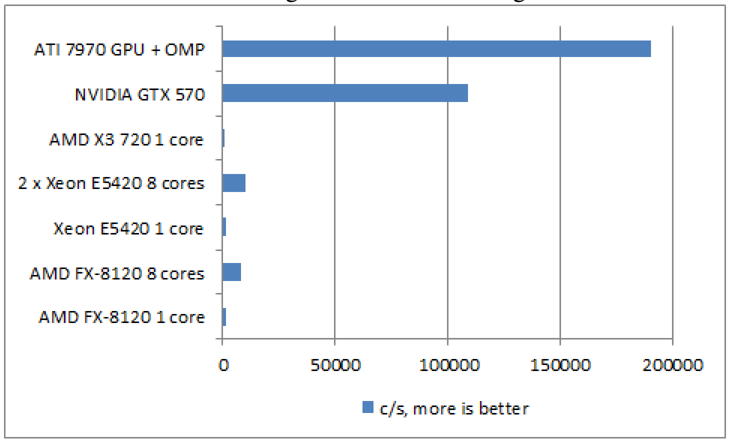
Figure 16: manifest.xml snipped sample



```
<?xml version="1.0" encoding="UTF-8"?>
<manifest:manifest xmlns:manifest="urn:oasis:names:tc:opendocument:xmlns:manifest:1.
0" manifest:version="1.2">
 <manifest:file-entry manifest:media-type="application/vnd.oasis.opendocument.text"
manifest:version="1.2" manifest:full-path="/"/>
 <manifest:file-entry manifest:media-type="text/xml" manifest:full-path="content.xml
" manifest:size="3767">
  <manifest:encryption-data manifest:checksum-type="SHA1/1K" manifest:checksum="32wQ
9k0ZGoQYEq9Th0tjbQFM4/4=">
   <manifest:algorithm manifest:algorithm-name="Blowfish CFB" manifest:initialisatio
n-vector="B+KK/znSZg4="/>
   <manifest:key-derivation manifest:key-derivation-name="PBKDF2" manifest:key-size=
"16" manifest:iteration-count="1024" manifest:salt="sSS+nzNG+3fg68w7uAAo+A=="/>
   <manifest:start-key-generation manifest:start-key-generation-name="SHA1" manifest
:key-size="20"/>
  </manifest:encryption-data>
 </manifest:file-entry>
 <manifest:file-entry manifest:media-type="text/xml" manifest:full-path="styles.xml"
 manifest:size="11516">
  <manifest:encryption-data manifest:checksum-type="SHA1/1K" manifest:checksum="VfGz
SAHdzbUjyXkDIrpUK/ws1Eg=">
   <manifest:algorithm manifest:algorithm-name="Blowfish CFB" manifest:initialisatio
n-vector="z/da7UWzg7M="/>
   <manifest:key-derivation manifest:key-derivation-name="PBKDF2" manifest:key-size=
"16" manifest:iteration-count="1024" manifest:salt="WmxFQxvHF2jTHmkF+ewhig=="/>
   <manifest:start-key-generation manifest:start-key-generation-name="SHA1" manifest
:key-size="20"/>
  </manifest:encryption-data>
 </manifest:file-entry>
```

Benchmarks,
We compare the performance of ODF JtR plug-in on different machines below,

Figure 17: ODFcracking benchmarks



ODF cracking benchmarks

```
$../run/john -fo:odf -t # AMD FX-8120 (single core)
Benchmarking: ODF SHA-1 Blowfish [32/64]... DONE
Raw:  1189 c/s real, 1189 c/s virtual

RETAKE $../run/john -fo:odf -t # AMD FX-8120 (8 cores)
Benchmarking: ODF SHA-1 Blowfish [32/64]... (8xOMP) DONE
Raw:  5263 c/s real, 865 c/s virtual

$ ../run/john -fo:odf -t # Xeon E5420 (1 core)

$ ../run/john -fo:odf -t # 2 x Xeon E5420 (8 cores)

# GPU TODO

$ ../run/john -fo:pwsafe-opencl -t # ATI 6970 GPU
Benchmarking: Password Safe SHA-256 [OpenCL]... DONE
Raw:  11815 c/s real, 768000 c/s virtual

$ ../run/john -fo:pwsafe-opencl -t # GeForce GTX 570 OpenCL
Benchmarking: Password Safe SHA-256 [OpenCL]... DONE
Raw:  27131 c/s real, 27131 c/s virtual

$ ../run/john -fo:pwsafe-cuda -t # GeForce GTX 570 CUDA
Benchmarking: Password Safe SHA-256 [CUDA]... DONE
Raw:  107185 c/s real, 107185 c/s virtual
```

By using techniques described in this section, it is possible to write a cracker for earlier version of ODF files. See run/sxc2john.py and src/sxc_fmt_plug.c file in JtR source tree.

2.15 Analysis of PDF files

2.15.1 Analysis of PDF files using RC4 encryption

Code 13: PDF RC4 Cracker

```
try_key(unsigned char *key)
{
   unsigned char output[32];
   RC4_KEY arc4;
   RC4_set_key(&arc4, 5, key);
   /* encrypt padding */
   RC4(&arc4, 32, padding, output);

   /* compare padding to original value of u */
   if(memcmp(output, u, 32) == 0) {
      puts("Found␣RC4␣40-bit␣key!")
      exit(0);
   }
}

keyspace_search()
{
   int i, j, k;
   int is = 0x00;
   int js = 0x00;
   int ks = 0x00;
   int ls = 0x00;
   int ms = 0x00;

   for(i = is; i <= 255; i++) {
      for(j = js; j <= 255; j++) {
#pragma omp parallel for
         for(k = ks; k <= 255; k++) {
            int l, m;
            for(l = ls; l <= 255; l++) {
               for(m = ms; m <= 255; m++) {
                  unsigned char hashBuf[5];
                  hashBuf[0] = (char)i;
                  hashBuf[1] = (char)j;
                  hashBuf[2] = (char)k;
                  hashBuf[3] = (char)l;
                  hashBuf[4] = (char)m;
                  try_key(hashBuf);
               }
            }
         }
      }
   }
}
```

2.15.2 Guaranteed decryption of PDF files using 40-bit RC4 encryption

Code 14: PDF 40-bit RC4 Cracker

```c
try_key(unsigned char *key)
{
    unsigned char output[32];
    RC4_KEY arc4;
    RC4_set_key(&arc4, 5, key);
    /* encrypt padding */
    RC4(&arc4, 32, padding, output);

    /* compare padding to original value of u */
    if(memcmp(output, u, 32) == 0) {
        puts("Found␣RC4␣40-bit␣key!")
        exit(0);
    }
}

keyspace_search()
{
    int i, j, k;
    int is = 0x00;
    int js = 0x00;
    int ks = 0x00;
    int ls = 0x00;
    int ms = 0x00;

    for(i = is; i <= 255; i++) {
        for(j = js; j <= 255; j++) {
#pragma omp parallel for
            for(k = ks; k <= 255; k++) {
                int l, m;
                for(l = ls; l <= 255; l++) {
                    for(m = ms; m <= 255; m++) {
                        unsigned char hashBuf[5];
                        hashBuf[0] = (char)i;
                        hashBuf[1] = (char)j;
                        hashBuf[2] = (char)k;
                        hashBuf[3] = (char)l;
                        hashBuf[4] = (char)m;
                        try_key(hashBuf);
                    }
                }
            }
        }
    }
}
```

2.15.3 Analysis of Adobe Acrobat 9 encrypted files (R5 algorithm)

Code 15: PDF 40-bit RC4 Cracker

```
try_key(unsigned char *key)
{
   unsigned char output[32];
   RC4_KEY arc4;
   RC4_set_key(&arc4, 5, key);
   /* encrypt padding */
   RC4(&arc4, 32, padding, output);

   /* compare padding to original value of u */
   if(memcmp(output, u, 32) == 0) {
      puts("Found␣RC4␣40-bit␣key!")
      exit(0);
   }
}

keyspace_search()
{
   int i, j, k;
   int is = 0x00;
   int js = 0x00;
   int ks = 0x00;
   int ls = 0x00;
   int ms = 0x00;

   for(i = is; i <= 255; i++) {
      for(j = js; j <= 255; j++) {
#pragma omp parallel for
         for(k = ks; k <= 255; k++) {
            int l, m;
            for(l = ls; l <= 255; l++) {
               for(m = ms; m <= 255; m++) {
                  unsigned char hashBuf[5];
                  hashBuf[0] = (char)i;
                  hashBuf[1] = (char)j;
                  hashBuf[2] = (char)k;
                  hashBuf[3] = (char)l;
                  hashBuf[4] = (char)m;
                  try_key(hashBuf);
               }
            }
         }
      }
   }
}
```

2.15.4 Analysis of Adobe Acrobat 10 and 11 encrypted files (R6 algorithm)

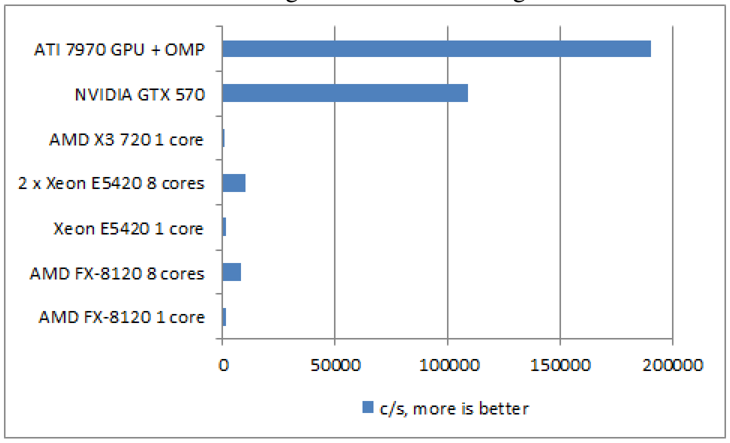
Code 16: PDF 40-bit RC4 Cracker

```
try_key(unsigned char *key)
{
    unsigned char output[32];
    RC4_KEY arc4;
    RC4_set_key(&arc4, 5, key);
    /* encrypt padding */
    RC4(&arc4, 32, padding, output);

    /* compare padding to original value of u */
    if(memcmp(output, u, 32) == 0) {
        puts("Found␣RC4␣40-bit␣key!")
        exit(0);
    }
}

keyspace_search()
{
    int i, j, k;
    int is = 0x00;
    int js = 0x00;
    int ks = 0x00;
    int ls = 0x00;
    int ms = 0x00;

    for(i = is; i <= 255; i++) {
        for(j = js; j <= 255; j++) {
#pragma omp parallel for
            for(k = ks; k <= 255; k++) {
                int l, m;
                for(l = ls; l <= 255; l++) {
                    for(m = ms; m <= 255; m++) {
                        unsigned char hashBuf[5];
                        hashBuf[0] = (char)i;
                        hashBuf[1] = (char)j;
                        hashBuf[2] = (char)k;
                        hashBuf[3] = (char)l;
                        hashBuf[4] = (char)m;
                        try_key(hashBuf);
                    }
                }
            }
        }
    }
}
```

2.15.5 Comparison of various KDF functions used in Adobe Acrobat

We compare the performance of ODF JtR plug-in on different machines below,

Figure 18: ODFcracking benchmarks



2.16 Analysis of RAR files.

RAR stands for Roshal ARchive and it is a proprietary archive file format that supports data compression, error recovery, and file spanning. RAR is a highly popular compression format embraced even by the software cracking scene. The RAR file format is well documented in technote.txt.

Table 9: RAR file format

Magic	6 bytes	0x9AA2D903, Magic
Version	2 bytes	0xB54BFB67, Magic
Cipher Name	32 bytes	Determine what algorithms are used
Cipher Mode	32 bytes	Version of the database format
Hash Spec	32 bytes	Initial random number to start on the sha256 of the key
Payload Offset	4 bytes	Initialization vector used for all algorithms
Key Bytes	4 bytes	Encrypted 128 random value using P' with Twofish algorithm
mkDigest	20 bytes	SHA1
mkDigestSalt	32 bytes	SHA256 hash of only the contents (entire file minus starting 124 bytes)
mkDigestIterations	4 bytes	Number of iterations in Phttp://en.wikipedia.org/wiki/Personal_Storage_TableBKDF2 function
UUID	40	Number of rounds to do AES block encryption on the Master Key
keyblock structure (8 entries)	48 bytes each	

Where, keyblock structure has the following format,

active	4 bytes	denotes whether this key slot is active or not
passwordIterations	4 bytes	parameters used for password processing
passwordSalt	32 bytes	parameters used for password processing
keyMaterialOffset	4 bytes	parameters used for AF store/load
stripes	4 bytes	parameters used for AF store/load

RAR can encrypt data in two different way: First, in "-hp" mode, both file data and file headers (which contains file names and other metadata) are encrypted. Encryption algorithm is changed to cipher block chaining (CBC) mode over AES (Advanced Encryption Standard) with 128 bit key length.Encryption of both file data and file headers. RAR uses custom key stretching algorithm to deter brute-force attacks. In "-p" mode only file data is encrypted. At first, it seems files encrypted using "-hp" seem to offer more security since even the file headers are encrypted. However, in practice file encrypted using -"hp" can be attacked in two different way, 1) known partial plain-text attack 2) File header CRC verification. File encrypted using "-p" are harder to brute-force, the decrypted (but still compressed) file data streams contain no information if they are valid compressed data stream. Hence to attack "-p" mode files, a full Un-RAR engine must be implemented which de-compresses the decrypted data, the computes the CRC over un-compressed data and compares the CRC with the value stored in the file header.

The "known partial plain-text attack" on "-hp" mode files was first found out Marc Bevland and used in his unrarhp tool. Our initiial implementation of RAR cracker could only deal with "-hp" mode files. It has been later extended by magnum (JtR jumbo's maintainer) to support "-p" mode files. magnum has even implemented GPU cracking support of RAR files!

[Insert RAR key stretching algorithm] [Insert RAR cracking snippet for "-hp" mode RAR files] [Benchmark CPU, our GPU, igrargpu]

2.17 Analysis of LUKS

Linux Unified Key Setup or LUKS is a disk-encryption specification. The reference implementation for LUKS operates on Linux and is based on an enhanced version of cryptsetup, using dm-crypt as the disk encryption backend. Device-mapper crypt (dm-crypt) target provides transparent encryption of block devices using the kernel crypto API. LUKS is the standard for Linux hard disk encryption. By providing a standard on-disk-format, it does not only facilitate compatibility among distributions, but also provides secure management of multiple user passwords. In contrast to existing solution, LUKS stores all setup necessary setup information in the partition header, enabling the user to transport or migrate his data seamlessly. While LUKS is a standard on-disk format, there is also a reference implementation. LUKS for dm-crypt is implemented in an enhanced version of cryptsetup. cryptsetup is used to conveniently setup dm-crypt managed block devices under Linux.

Table 10: LUKS header fields (size is 208 bytes + 48 * LUKS_NUMKEYS = 592 bytes)

Magic	6 bytes	0x9AA2D903, Magic
Version	2 bytes	0xB54BFB67, Magic
Cipher Name	32 bytes	Determine what algorithms are used
Cipher Mode	32 bytes	Version of the database format
Hash Spec	32 bytes	Initial random number to start on the sha256 of the key
Payload Offset	4 bytes	Initialization vector used for all algorithms
Key Bytes	4 bytes	Encrypted 128 random value using P' with Twofish algorithm
mkDigest	20 bytes	SHA1
mkDigestSalt	32 bytes	SHA256 hash of only the contents (entire file minus starting 124 bytes)
mkDigestIterations	4 bytes	Number of iterations in PBKDF2 function
UUID	40	Number of rounds to do AES block encryption on the Master Key
keyblock structure (8 entries)	48 bytes each	

Where, keyblock structure has the following format,

active	4 bytes	denotes whether this key slot is active or not
passwordIterations	4 bytes	parameters used for password processing
passwordSalt	32 bytes	parameters used for password processing
keyMaterialOffset	4 bytes	parameters used for AF store/load
stripes	4 bytes	parameters used for AF store/load

Our naive brute-force software (based on Revelation Python sources) is super slow and achieves a speed of merely 0.3 p/s. This slowness can be partially attributed to interpretive nature of Python code. Our second implementation in C (based on official cryptsetup sources) is three times faster and achieves roughly 1 p/s. [Give estimates for cracking 8 byte alpha and alphanumeric passwords]. LUKS has upto 8 key slots. One clever attack is that we can choose to attack a key slot which has minimum cryptographic strength (i.e use lesser iterations in its key derivation function). Can I use LUKS or cryptsetup with a more secure (external) medium for key storage, e.g. TPM or a smartcard? Yes, see the answers on using a file-supplied key. You do have to write the glue-logic yourself though. Basically you can have cryptsetup read the key from STDIN and write it there with your own tool that in turn gets the key from the more secure key storage.

2.18 Analysis of TrueCrypt

TrueCrypt is a popular on-the-fly encryption. It can create a file-hosted container or write a partition which consists of an encrypted volume with its own file system (contained within a regular file) which can then be mounted as if it were a real disk. TrueCrypt also supports device-hosted volumes, which can be created on either an individual partition or an entire disk. Because presence of a TrueCrypt volume can not be verified without the password, disk and filesystems utilities may report the filesystem as unformatted or corrupted that may lead to data loss after incorrect user intervention or automatic "repair".

The standard volume header uses the first 512 bytes of the TrueCrypt container. It contains the master keys needed to decrypt the volume. The 512 bytes hidden volume header is stored 1536 bytes from the end of the host volume. TrueCrypt volumes have no "signature" or ID strings. Until decrypted, they appear to consist solely of random data.

Free space on each TrueCrypt volume is filled with random data when the volume is created.

It is not possible to identify TrueCrypt containers by simply looking for some well-defined magic string. This provides strong deniability. Information about the exact PKBDF2 function and cipher(s) used in an encrypted container is not stored in the header. As a consequence all possible combination must be tried. This slow down the brute force attack considerably. The various possible PBKDF2 algorithms used by TrueCrypt are : PBKDF2-HMAC-SHA256 with 2000 rounds etc. The various possible encrytion ciphers (including chained ciphers) are AES-XTS etc.

Table 11: TrueCrypt Volume Format Specification (512 bytes)

SALT	64	Unencrypted	Salt
MAGIC	4 bytes	Encrypted	ASCII string "TRUE"
Version	2 bytes	Encrypted	Volume header format version
Min. Version	2 bytes	Encrypted	Encrypted
crc_keys	4 bytes	Encrypted	CRC32 of the key section
vol_ctime	8 bytes	Encrypted	Volume creation time
hdr_ctime	8 bytes	Encrypted	Header creation time
sz_hidvol	8 bytes	Encrypted	Size of hidden volume
sz_vol	8 bytes	Encrypted	Size of volume
off_mk_scope	8 bytes	Encrypted	Byte offset of the start of the master key scope
sz_mk_scope	8 bytes	Encrypted	Size of the encrypted area withint he master key scope
Flags	4 bytes	Encrypted	Flag bits
sec_sz	4 bytes	Encrypted	Sector size (in bytes)
unused	120 bytes	Encrypted	Reserved
crc_dhdr	4 bytes	Encrypted	CRC32 of decrypted header (except keys)
keys	256 bytes	Encrypted	Concatenated primary and secondary master keys

Benchmarks of all TC crackers out there (Excel bar chart).

Pseudo-code for cracking TrueCrypt volume

```
$ john -fo:raw-sha1 -t # AMD FX-8120 (1 core)
Benchmarking: Raw SHA-1 [128/128 SSE2 intrinsics 8x]... DONE
Raw:  17957K c/s real, 17957K c/s virtual

$ john -fo:django -t # AMD FX-8120
Benchmarking: Django PBKDF2-HMAC-SHA-256 (x10000) [32/64]... DONE
Raw:  46.0 c/s real, 46.0 c/s virtual

$ john -fo:django -t # AMD FX-8120 (all cores)
Benchmarking: Django PBKDF2-HMAC-SHA-256 (x10000) [32/64]... (8
    xOMP) DONE
Raw:  203 c/s real, 26.9 c/s virtual

$ ../run/john -fo=django -t # Xeon E5420 (1 core)
Benchmarking: Django PBKDF2-HMAC-SHA-256 (x10000) [32/64]... DONE
Raw:  34.3 c/s real, 34.3 c/s virtual

$ ../run/john -fo=django -t # 2 x Xeon E5420 (8 cores)
Benchmarking: Django PBKDF2-HMAC-SHA-256 (x10000) [32/64]... (8
    xOMP) DONE
Raw:  160 c/s real, 20.2 c/s virtual
```

Also mention real-life usage of these tools in competition ;)

VNC cracking

```
unsigned char OUT[16] = { 0 };

/* key processing */
for(i = 0; i < strlen((const char*)key); i++)
  PASSWORD[i] = BIT_FLIP(PASSWORD[i]);

/* encrypt challenge using PASSWORD */
memcpy(des_key, PASSWORD, 8);
DES_set_odd_parity(&des_key);
DES_set_key_checked(&des_key, &schedule);
DES_cbc_encrypt(CHALLENGE[0:8], &OUT[0], 8,
    &schedule, &ivec, DES_ENCRYPT);
DES_cbc_encrypt(CHALLENGE[8:16], &OUT[8], 8,
    &schedule, &ivec, DES_ENCRYPT);

if(OUT == RESPONSE) {
  /* password found */
}
```

2.19 .pfx / .p12 files

Work in progress.

JtR-jumbo is a community enhanced version of JtR with

[Compiled debug version of OpenSSL to trace which encryption functions are called]

Compare our "trivial" cracker with Elcomsoft's EDPR (get benchmarks from all servers).

It defines a file format commonly used to store X.509 private keys with accompanying public key certificates, protected with a password-based symmetric key, and is the successor to PFX from Microsoft. PFX has received heavy criticism of being one of the most complex cryptographic protocols,[1] but nevertheless remains the only standard way today to store private keys and certificates in a single encrypted file.

Our .P12 cracker cheats by not not implementing its own crypto functions, instead it replies on OpenSSL's verifyxyz function to do the heavy lifting.

http://www.drh-consultancy.demon.co.uk/pkcs12faq.html/

#12 supports the following encryption algorithms.

128 bit RC4 with SHA1 40 bit RC4 with SHA1 3 key triple DES with SHA1 (168 bits) 2 key triple DES with SHA1 (112 bits) 128 bit RC2 with SHA1 40 bit RC2 with SHA1

In addition the PKCS#5 v1.5 modes are possible as well. This also permits the following.

DES with MD5 (56bit) DES with MD2 (56bit)

What's this I hear about iteration counts? A. The algorithm used to generate keys from passwords and the MAC has an optional iteration count. This determines how many times part of the algorithm is repeated. It's a way of slowing down the key derivation process to make it harder to make dictionary attacks on the password. The -info option now prints information about iteration counts. Q. What iteration counts are used?

A. By default I set both iteration counts to 2048. If you use the -nomaciter option the MAC iteration count is also set to 1 some software such as MSIE4 needs this option because it does not support mac iteration counts. If you use the noiter option the iteration count is set to 1: since this makes dictionary attacks on the password easier this is not recommended.

MSIE5 uses 2000 for the encryption iteration count. If you have the 'enable strong protection' option checked then it uses 2000 for the MAC count otherwise it uses 1 (for compatability with earlier versions of MSIE).

ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-12/pkcs-12v1.pdf

Both are PKCS #12 files (Personal Information Exchange Syntax)

2.20 Analysis of Mozilla master passwords

Work in progress.

2.21 Analysis of encrypted 7-Zip files

Work in progress.

3 Analysis of security of various authentication protocols

3.1 Analysis of Kerberos v5 authentication protocol

Kerberos is a computer network authentication protocol which works on the basis of "tickets" to allow nodes communicating over a non-secure network to prove their identity to one another in a secure manner .

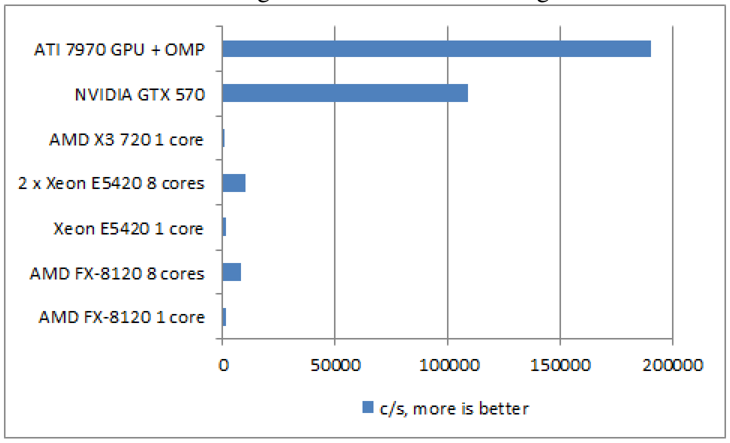
Code 17: LastPass Cracker

```
/* Derives AES-256 decryption key from USERNAME and PASSWORD */


PKCS5_PBKDF2_HMAC(PASSWORD, strlen(PASSWORD), USERNAME, strlen(
    USERNAME), ITERATIONS, EVP_sha256(), 32, key);

/* Try decrypting encrypted_username */
AES_KEY akey;
unsigned char iv[16] = { 0 };
AES_set_decrypt_key(key, 256, &akey)
AES_cbc_encrypt(encrypted_username, decrypted_username, 32, &akey,
     iv, AES_DECRYPT);

if(strcmp(decrypted_username, username) == 0))
    /* Password Found */
else
   /* Password is not correct */
```

Essentially we decrypt the encrypted_username value and compare it against the original username to verify if the gived password was correct or not. For details see src/lastpass_fmt_plug.c in JtR source tree.

We compare the performance of LastPass cracker on different machines below,

Figure 19: LastPass Cracking Benchmarks



LastPass cracking benchmarks

```
$../run/john -fo:keepass -t # AMD FX-8120 (single core)
Benchmarking: KeePass SHA-256 AES [32/64 OpenSSL]... DONE
Raw:  82.0 c/s real, 82.0 c/s virtual

$../run/john -fo:keepass -t # AMD FX-8120 (8 cores)
Benchmarking: KeePass SHA-256 AES [32/64 OpenSSL]... (8xOMP) DONE
Raw:  409 c/s real, 51.2 c/s virtual

# GPU RESULTS, TODO
```

Describe Ettercap + JtR work

3.2 Analysis of MongoDB authentication protocol

DONE. At least it has some protection unlike Redis which sends the password in clear text

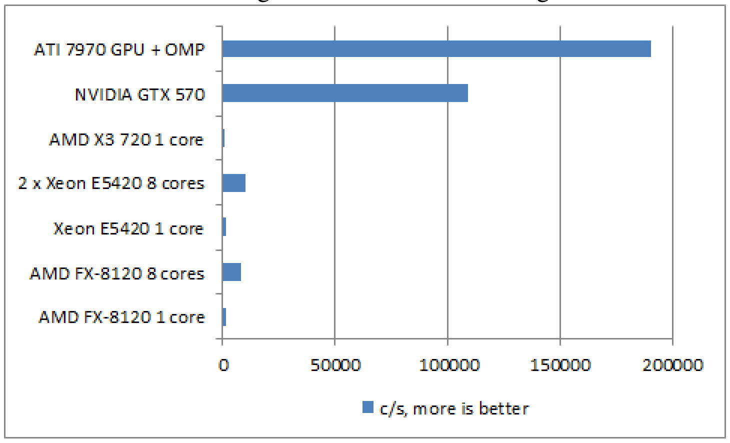
Code 18: LastPass Cracker

```c
/* Derives AES-256 decryption key from USERNAME and PASSWORD */


PKCS5_PBKDF2_HMAC(PASSWORD, strlen(PASSWORD), USERNAME, strlen(
     USERNAME), ITERATIONS, EVP_sha256(), 32, key);

/* Try decrypting encrypted_username */
AES_KEY akey;
unsigned char iv[16] = { 0 };
AES_set_decrypt_key(key, 256, &akey)
AES_cbc_encrypt(encrypted_username, decrypted_username, 32, &akey,
      iv, AES_DECRYPT);

if(strcmp(decrypted_username, username) == 0))
    /* Password Found */
else
   /* Password is not correct */
```

Essentially we decrypt the encrypted_username value and compare it against the original username to verify if the gived password was correct or not. For details see src/lastpass_fmt_plug.c in JtR source tree.

We compare the performance of LastPass cracker on different machines below,

Figure 20: LastPass Cracking Benchmarks



LastPass cracking benchmarks

```
$../run/john -fo:keepass -t # AMD FX-8120 (single core)
Benchmarking: KeePass SHA-256 AES [32/64 OpenSSL]... DONE
Raw:  82.0 c/s real, 82.0 c/s virtual

$../run/john -fo:keepass -t # AMD FX-8120 (8 cores)
Benchmarking: KeePass SHA-256 AES [32/64 OpenSSL]... (8xOMP) DONE
Raw:  409 c/s real, 51.2 c/s virtual

# GPU RESULTS, TODO
```

Describe Ettercap + JtR work

3.3 Analysis of MySQL challenge-response authentication protocol

DONE. Describe Ettercap + JtR work

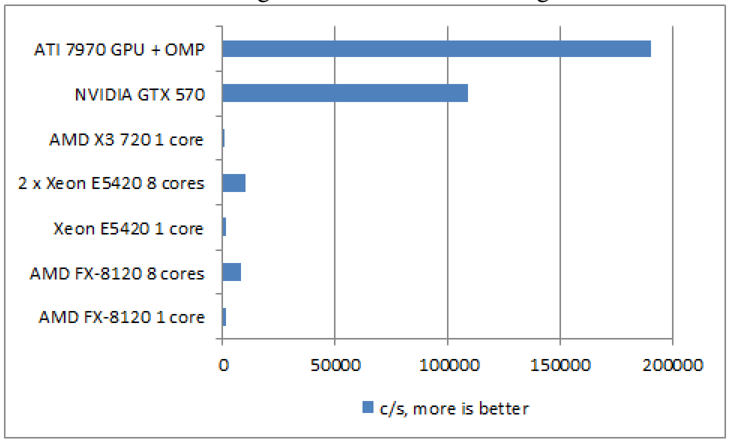
Code 19: LastPass Cracker

```
/* Derives AES-256 decryption key from USERNAME and PASSWORD */


PKCS5_PBKDF2_HMAC(PASSWORD, strlen(PASSWORD), USERNAME, strlen(
    USERNAME), ITERATIONS, EVP_sha256(), 32, key);

/* Try decrypting encrypted_username */
AES_KEY akey;
unsigned char iv[16] = { 0 };
AES_set_decrypt_key(key, 256, &akey)
AES_cbc_encrypt(encrypted_username, decrypted_username, 32, &akey,
     iv, AES_DECRYPT);

if(strcmp(decrypted_username, username) == 0))
    /* Password Found */
else
   /* Password is not correct */
```

Essentially we decrypt the encrypted_username value and compare it against the original username to verify if the gived password was correct or not. For details see src/lastpass_fmt_plug.c in JtR source tree.

We compare the performance of LastPass cracker on different machines below,

Figure 21: LastPass Cracking Benchmarks



LastPass cracking benchmarks

```
$../run/john -fo:keepass -t # AMD FX-8120 (single core)
Benchmarking: KeePass SHA-256 AES [32/64 OpenSSL]... DONE
Raw:  82.0 c/s real, 82.0 c/s virtual

$../run/john -fo:keepass -t # AMD FX-8120 (8 cores)
Benchmarking: KeePass SHA-256 AES [32/64 OpenSSL]... (8xOMP) DONE
Raw:  409 c/s real, 51.2 c/s virtual

# GPU RESULTS, TODO
```

Describe Ettercap + JtR work

3.4 Analysis of PostgreSQL authentication protocol

DONE. Describe Ettercap + JtR + Nmap + Metasploit work. Man in the middle downgrade attack.

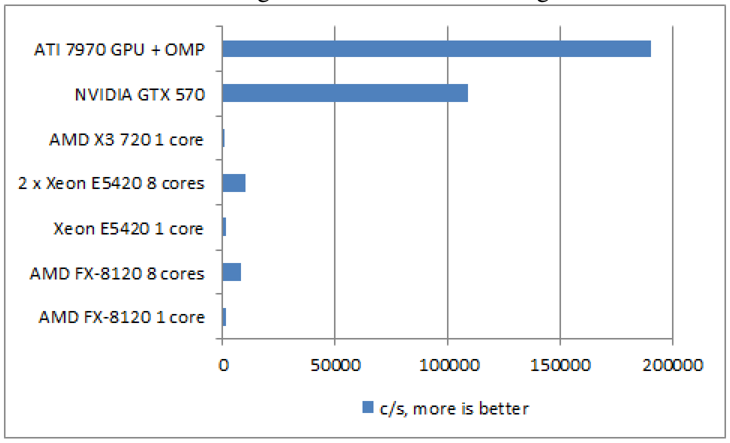
Code 20: LastPass Cracker

```c
/* Derives AES-256 decryption key from USERNAME and PASSWORD */


PKCS5_PBKDF2_HMAC(PASSWORD, strlen(PASSWORD), USERNAME, strlen(
    USERNAME), ITERATIONS, EVP_sha256(), 32, key);

/* Try decrypting encrypted_username */
AES_KEY akey;
unsigned char iv[16] = { 0 };
AES_set_decrypt_key(key, 256, &akey)
AES_cbc_encrypt(encrypted_username, decrypted_username, 32, &akey,
     iv, AES_DECRYPT);

if(strcmp(decrypted_username, username) == 0))
    /* Password Found */
else
   /* Password is not correct */
```

Essentially we decrypt the encrypted_username value and compare it against the original username to verify if the gived password was correct or not. For details see src/lastpass_fmt_plug.c in JtR source tree.

We compare the performance of LastPass cracker on different machines below,

Figure 22: LastPass Cracking Benchmarks



LastPass cracking benchmarks

```
$../run/john -fo:keepass -t # AMD FX-8120 (single core)
Benchmarking: KeePass SHA-256 AES [32/64 OpenSSL]... DONE
Raw:  82.0 c/s real, 82.0 c/s virtual

$../run/john -fo:keepass -t # AMD FX-8120 (8 cores)
Benchmarking: KeePass SHA-256 AES [32/64 OpenSSL]... (8xOMP) DONE
Raw:  409 c/s real, 51.2 c/s virtual

# GPU RESULTS, TODO
```

Describe Ettercap + JtR work

3.5 Analysis of Oracle O5LOGON protocol

DONE. Describe Ettercap + JtR + Nmap work

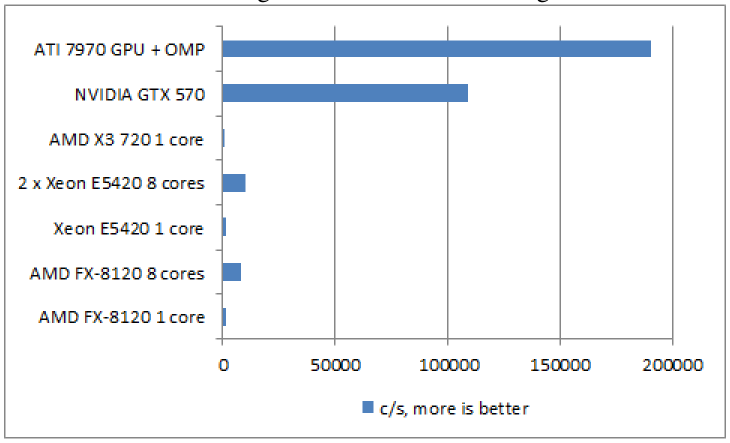
Code 21: LastPass Cracker

```
/* Derives AES-256 decryption key from USERNAME and PASSWORD */


PKCS5_PBKDF2_HMAC(PASSWORD, strlen(PASSWORD), USERNAME, strlen(
     USERNAME), ITERATIONS, EVP_sha256(), 32, key);

/* Try decrypting encrypted_username */
AES_KEY akey;
unsigned char iv[16] = { 0 };
AES_set_decrypt_key(key, 256, &akey)
AES_cbc_encrypt(encrypted_username, decrypted_username, 32, &akey,
      iv, AES_DECRYPT);

if(strcmp(decrypted_username, username) == 0))
     /* Password Found */
else
    /* Password is not correct */
```

Essentially we decrypt the encrypted_username value and compare it against the original username to verify if the gived password was correct or not. For details see src/lastpass_fmt_plug.c in JtR source tree.

We compare the performance of LastPass cracker on different machines below,

Figure 23: LastPass Cracking Benchmarks



LastPass cracking benchmarks

```
$../run/john -fo:keepass -t # AMD FX-8120 (single core)
Benchmarking: KeePass SHA-256 AES [32/64 OpenSSL]... DONE
Raw:  82.0 c/s real, 82.0 c/s virtual

$../run/john -fo:keepass -t # AMD FX-8120 (8 cores)
Benchmarking: KeePass SHA-256 AES [32/64 OpenSSL]... (8xOMP) DONE
Raw:  409 c/s real, 51.2 c/s virtual

# GPU RESULTS, TODO
```

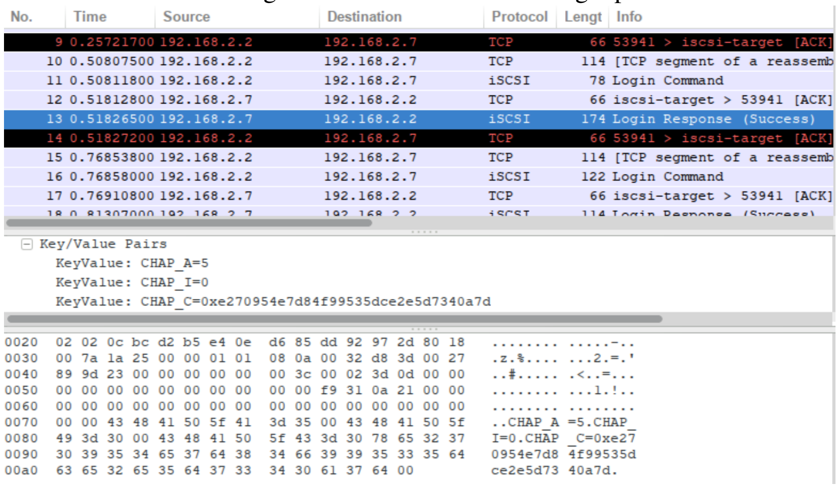
Describe Ettercap + JtR work

3.6 Analysis of iSCSI CHAP authentication protocol

iSCSI (Internet Small Computer System Interface) is an Internet Protocol (IP) based networking standard for linking storage facilities. iSCSI allows clients (called initiators) to send SCSI commands (CDBs) to SCSI storage devices (targets) on remote servers to facililate data transfer. It is a storage area network (SAN) protocol, allowing organizations to consolidate storage into data center storage arrays while providing hosts (such as database and web servers) with the illusion of locally attached disks.

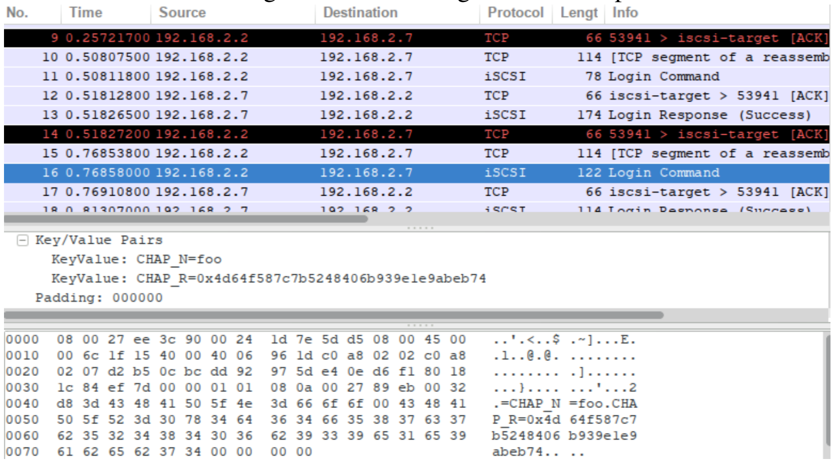
iSCSI targets can be password protected by using CHAP protocol. <Decribe algorithm>.We have extended Ettercap to sniff and decode the key packets involved in iSCSI CHAP authentication protocol.

Figure 24: iSCSI initiator to target packet



```
No.  Time       Source       Destination  Protocol Lengt Info
   9 0.25721700 192.168.2.2  192.168.2.7  TCP       66 53941 > iscsi-target [ACK]
  10 0.50807500 192.168.2.2  192.168.2.7  TCP      114 [TCP segment of a reassemb
  11 0.50811800 192.168.2.2  192.168.2.7  iSCSI     78 Login Command
  12 0.51812800 192.168.2.7  192.168.2.2  TCP       66 iscsi-target > 53941 [ACK]
  13 0.51826500 192.168.2.7  192.168.2.2  iSCSI    174 Login Response (Success)
  14 0.51827200 192.168.2.2  192.168.2.7  TCP       66 53941 > iscsi-target [ACK]
  15 0.76853800 192.168.2.2  192.168.2.7  TCP      114 [TCP segment of a reassemb
  16 0.76858000 192.168.2.2  192.168.2.7  iSCSI    122 Login Command
  17 0.76910800 192.168.2.7  192.168.2.2  TCP       66 iscsi-target > 53941 [ACK]
  18 0.81307000 192.168.2.7  192.168.2.2  iSCSI    114 Login Response (Success)

⊟ Key/Value Pairs
    KeyValue: CHAP_A=5
    KeyValue: CHAP_I=0
    KeyValue: CHAP_C=0xe270954e7d84f99535dce2e5d7340a7d

0020  02 02 0c bc d2 b5 e4 0e  d6 85 dd 92 97 2d 80 18   ........ .....-..
0030  00 7a 1a 25 00 00 01 01  08 0a 00 32 d8 3d 00 27   .z.%.... ...2.=.'
0040  89 9d 23 00 00 00 00 00  00 3c 00 02 3d 0d 00 00   ..#..... .<..=...
0050  00 00 00 00 00 00 00 00  00 00 f9 31 0a 21 00 00   ........ ...1.!..
0060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0070  00 00 43 48 41 50 5f 41  3d 35 00 43 48 41 50 5f   ..CHAP_A =5.CHAP_
0080  49 3d 30 00 43 48 41 50  5f 43 3d 30 78 65 32 37   I=0.CHAP _C=0xe27
0090  30 39 35 34 65 37 64 38  34 66 39 39 35 33 35 64   0954e7d8 4f99535d
00a0  63 65 32 65 35 64 37 33  34 30 61 37 64 00         ce2e5d73 40a7d.
```

Figure 25: iSCSI target to initiator packet



```
No.   Time       Source       Destination   Protocol Lengt Info
    9 0.25721700 192.168.2.2  192.168.2.7   TCP        66 53941 > iscsi-target [ACK]
   10 0.50807500 192.168.2.2  192.168.2.7   TCP       114 [TCP segment of a reassemb
   11 0.50811800 192.168.2.2  192.168.2.7   iSCSI      78 Login Command
   12 0.51812800 192.168.2.7  192.168.2.2   TCP        66 iscsi-target > 53941 [ACK]
   13 0.51826500 192.168.2.7  192.168.2.2   iSCSI     174 Login Response (Success)
   14 0.51827200 192.168.2.2  192.168.2.7   TCP        66 53941 > iscsi-target [ACK]
   15 0.76853800 192.168.2.2  192.168.2.7   TCP       114 [TCP segment of a reassemb
   16 0.76858000 192.168.2.2  192.168.2.7   iSCSI     122 Login Command
   17 0.76910800 192.168.2.7  192.168.2.2   TCP        66 iscsi-target > 53941 [ACK]
   18 0.81307000 192.168.2.7  192.168.2.2   iSCSI     114 Login Response (Success)

⊟ Key/Value Pairs
    KeyValue: CHAP_N=foo
    KeyValue: CHAP_R=0x4d64f587c7b5248406b939e1e9abeb74
  Padding: 000000

0000   08 00 27 ee 3c 90 00 24  1d 7e 5d d5 08 00 45 00   ..'.<..$ .~]...E.
0010   00 6c 1f 15 40 00 40 06  96 1d c0 a8 02 02 c0 a8   .l..@.@. ........
0020   02 07 d2 b5 0c bc dd 92  97 5d e4 0e d6 f1 80 18   ........ .]......
0030   1c 84 ef 7d 00 00 01 01  08 0a 00 27 89 eb 00 32   ...}.... ...'...2
0040   d8 3d 43 48 41 50 5f 4e  3d 66 6f 6f 00 43 48 41   .=CHAP_N =foo.CHA
0050   50 5f 52 3d 30 78 34 64  36 34 66 35 38 37 63 37   P_R=0x4d 64f587c7
0060   62 35 32 34 38 34 30 36  62 39 33 39 65 31 65 39   b5248406 b939e1e9
0070   61 62 65 62 37 34 00 00  00 00                     abeb74.. ..
```

We have written a custom cracker for sniffed iSCSI CHAP authentication hashes and the following snippet shows the main steps involved,

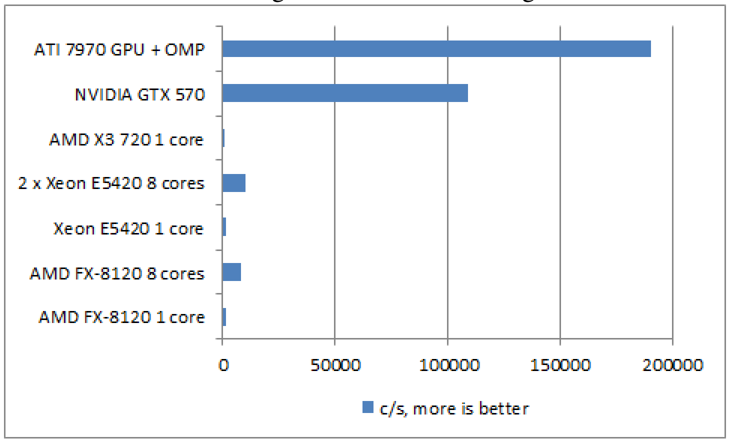
Code 22: iSCSI Cracker

```
/* Derives AES-256 decryption key from USERNAME and PASSWORD */


PKCS5_PBKDF2_HMAC(PASSWORD, strlen(PASSWORD), USERNAME, strlen(
    USERNAME), ITERATIONS, EVP_sha256(), 32, key);

/* Try decrypting encrypted_username */
AES_KEY akey;
unsigned char iv[16] = { 0 };
AES_set_decrypt_key(key, 256, &akey)
AES_cbc_encrypt(encrypted_username, decrypted_username, 32, &akey,
     iv, AES_DECRYPT);

if(strcmp(decrypted_username, username) == 0))
    /* Password Found */
else
   /* Password is not correct */
```

Essentially we decrypt the encrypted_username value and compare it against the original username to verify if the gived password was correct or not. For details see src/lastpass_fmt_plug.c in JtR source tree.

We compare the performance of LastPass cracker on different machines below,

Figure 26: iSCSI Cracking Benchmarks



iSCSI cracking benchmarks

```
$../run/john -fo:keepass -t # AMD FX-8120 (single core)
Benchmarking: KeePass SHA-256 AES [32/64 OpenSSL]... DONE
Raw:  82.0 c/s real, 82.0 c/s virtual

$../run/john -fo:keepass -t # AMD FX-8120 (8 cores)
Benchmarking: KeePass SHA-256 AES [32/64 OpenSSL]... (8xOMP) DONE
Raw:  409 c/s real, 51.2 c/s virtual

# GPU RESULTS, TODO
```

Describe Ettercap + JtR work

3.7 Analysis of VNC protocol

Virtual Network Computing (VNC) is a graphical desktop sharing system that uses the RFB protocol (remote framebuffer) to remotely control another computer. It transmits the keyboard and mouse events from one computer to another, relaying the graphical screen updates back in the other direction, over a network. VNC is platform-independent – a VNC viewer on one operating system may connect to a VNC server on the same or any other operating system. A VNC system consists of a client, a server, and a communication protocol. The VNC server is the program on the machine that shares its screen. The server passively allows the client to take control of it. The VNC client (or viewer) is the program that watches, controls, and interacts with the server. The client controls the server. The VNC protocol (RFB) is very simple, based on one graphic primitive from server to client ("Put a rectangle of pixel data at the specified X,Y position") and event messages from client to server. VNC by default uses TCP port 5900+N,[5][6] where N is the display number. The first step in attacking VNC cracking involves passive sniffing of the VNC traffic. Once the traffic has been captured.

VNC encryption key can be potentially broken only by mere passive sniffing of the traffic. In our opinion, VNC authentication protocol offers poor security and hasn't been fixed even in the newer versions of the RFB protocol.

[Paste wireshark screenshots]

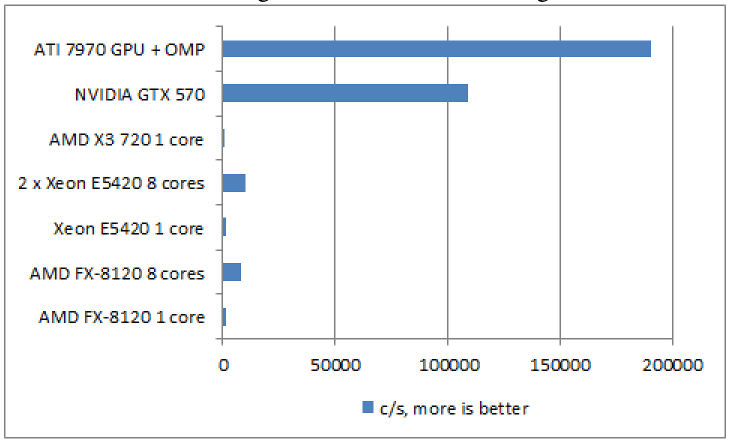
Code 23: LastPass Cracker

```c
/* Derives AES-256 decryption key from USERNAME and PASSWORD */


PKCS5_PBKDF2_HMAC(PASSWORD, strlen(PASSWORD), USERNAME, strlen(
    USERNAME), ITERATIONS, EVP_sha256(), 32, key);

/* Try decrypting encrypted_username */
AES_KEY akey;
unsigned char iv[16] = { 0 };
AES_set_decrypt_key(key, 256, &akey)
AES_cbc_encrypt(encrypted_username, decrypted_username, 32, &akey,
     iv, AES_DECRYPT);

if(strcmp(decrypted_username, username) == 0))
    /* Password Found */
else
   /* Password is not correct */
```

Essentially we decrypt the encrypted_username value and compare it against the original username to verify if the gived password was correct or not. For details see src/lastpass_fmt_plug.c in JtR source tree.

We compare the performance of LastPass cracker on different machines below,

Figure 27: LastPass Cracking Benchmarks



LastPass cracking benchmarks

```
$../run/john -fo:keepass -t # AMD FX-8120 (single core)
Benchmarking: KeePass SHA-256 AES [32/64 OpenSSL]... DONE
Raw:  82.0 c/s real, 82.0 c/s virtual

$../run/john -fo:keepass -t # AMD FX-8120 (8 cores)
Benchmarking: KeePass SHA-256 AES [32/64 OpenSSL]... (8xOMP) DONE
Raw:  409 c/s real, 51.2 c/s virtual

# GPU RESULTS, TODO
```

Describe Ettercap + JtR work

3.8 Analysis of LastPass authentication protocol

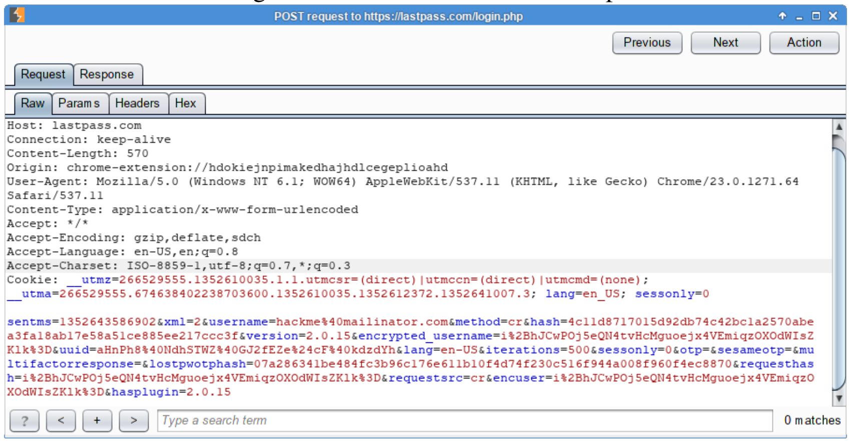
LastPass is a free online password manager and Form Filler that makes your web browsing easier and more secure. User's sensitive data is encrypted locally before upload so even LastPass cannot get access to it [12]. LastPass Password Manager protects passwords by using local AES encryption and a master password.

LastPass Password Manager is a closed source software and uses a proprietary file format. Earlier versions of LastPass used a weak KDF function and were susceptible to brute foce at high speeds (see [13]). However [13] is secretive (being from a commercial password cracking company) and does not contain any internal details. The lastest verions of LastPass Password Manager employ PBKDF2-SHA256 with variable number of iterations to slow down brute-force attacks.

In this work, we present security analysis of the lastest version of LastPass Password Manager. LastPass denied our requests to open up their proprietary file format for third-party security analysis. So, instead of analyzing the LastPass file format and finding possible offline attacks against it, we shifted to studying the authentication protocol used by LastPass.

The following screenshot shows the traffic exchanged between the LastPass Password Manager plug-in (running in the browser) and LastPass backend servers,

Figure 28: LastPass authentication protocol



```
Host: lastpass.com
Connection: keep-alive
Content-Length: 570
Origin: chrome-extension://hdokiejnpimakedhajhdlcegeplioahd
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.11 (KHTML, like Gecko) Chrome/23.0.1271.64
Safari/537.11
Content-Type: application/x-www-form-urlencoded
Accept: */*
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Cookie: __utmz=266529555.1352610035.1.1.utmcsr=(direct)|utmccn=(direct)|utmcmd=(none);
__utma=266529555.674638402238703600.1352610035.1352612372.1352641007.3; lang=en_US; sessonly=0

sentms=1352643586902&xml=2&username=hackme%40mailinator.com&method=cr&hash=4c11d8717015d92db74c42bc1a2570abe
a3fa18ab17e58a51ce885ee217ccc3f&version=2.0.15&encrypted_username=i%2BhJCwPOj5eQN4tvHcMguoejx4VEmiqzOXOdWIsZ
K1k%3D&uuid=aHnPh8%40NdhSTWZ%40GJ2fEZe%24cF%40kdzdYh&lang=en-US&iterations=500&sessonly=0&otp=&sesameotp=&mu
ltifactorresponse=&lostpwotphash=07a286341be484fc3b96c176e611b10f4d74f230c516f944a008f960f4ec8870&requesthas
h=i%2BhJCwPOj5eQN4tvHcMguoejx4VEmiqzOXOdWIsZK1k%3D&requestsrc=cr&encuser=i%2BhJCwPOj5eQN4tvHcMguoejx4VEmiqzO
XOdWIsZK1k%3D&hasplugin=2.0.15
```

After some analsysis, we found out that the query parameter "encrypted_username" is essentially username (known value) encrypted with a key derived from user password. LastPass uses a PBKDF2 as its key derivation function. We have written a custom cracker for sniffed LastPass authentication traffic and the following snippet shows the main steps involved,

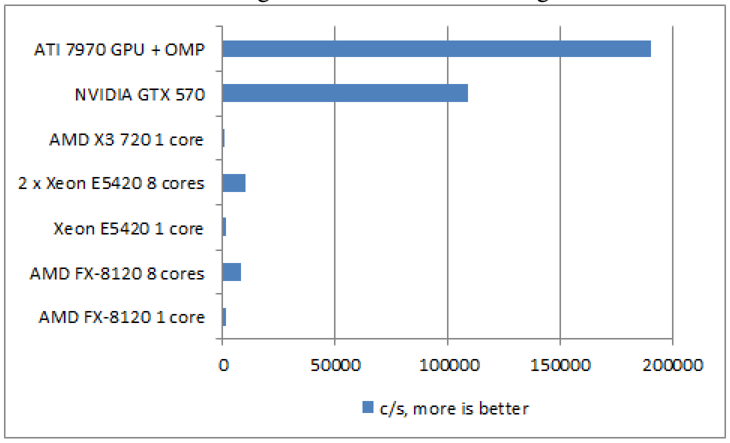
Code 24: LastPass Cracker

```c
/* Derives AES-256 decryption key from USERNAME and PASSWORD */


PKCS5_PBKDF2_HMAC(PASSWORD, strlen(PASSWORD), USERNAME, strlen(
    USERNAME), ITERATIONS, EVP_sha256(), 32, key);

/* Try decrypting encrypted_username */
AES_KEY akey;
unsigned char iv[16] = { 0 };
AES_set_decrypt_key(key, 256, &akey)
AES_cbc_encrypt(encrypted_username, decrypted_username, 32, &akey,
     iv, AES_DECRYPT);

if(strcmp(decrypted_username, username) == 0))
    /* Password Found */
else
   /* Password is not correct */
```

Essentially we decrypt the encrypted_username value and compare it against the original username to verify if the gived password was correct or not. For details see src/lastpass_fmt_plug.c in JtR source tree.

We compare the performance of LastPass cracker on different machines below,

Figure 29: LastPass Cracking Benchmarks



LastPass cracking benchmarks

```
$../run/john -fo:keepass -t # AMD FX-8120 (single core)
Benchmarking: KeePass SHA-256 AES [32/64 OpenSSL]... DONE
Raw:  82.0 c/s real, 82.0 c/s virtual

$../run/john -fo:keepass -t # AMD FX-8120 (8 cores)
Benchmarking: KeePass SHA-256 AES [32/64 OpenSSL]... (8xOMP) DONE
Raw:  409 c/s real, 51.2 c/s virtual

# GPU RESULTS, TODO
```

Offline attacks on LastPass offline database are work in progress.

3.9 Analysis of Clipperz authentication protocol

Clipperz is a popular free online password manager [14]. It does encryption on the local browser which gurantees confidentiality of data. Clipperz supports exporting encrypted databases into offline versions. Offline versions use the same cryptographic technology as used by the online version.

Clipperz does not believe in "security through obscurity" (unlike LastPass) and all the code behing Clipperz is open-source [**?**]. Clipperz uses SRP (Secure Remote Password protocol, see [17], [18] and [19]) for online and offline authentication. SRP is essentaily an authentication protocol for password-based, mutual authentication over an insecure network connection and requires both sides of the connection to have knowledge of the user's password. SRP offers security and deployment advantages over other challenge-response protocols, such as Kerberos and SSL, in that it does not require trusted key servers or certificate infrastructures. Instead, small verification keys derived from each user's password are stored and used by each SRP server application [19]. "SRP does not store plaintext passwords on the server side but instead uses what is known as a "non plaintext-equivalent verifier" [16].

Password verifier is derived from a Private key (called x) by using the formula v = g^x, where x (Private key) = H(s, H(I | ':' | p)), g is generator modulo N, I is username, p is cleartext password, H() is one-way hash function and s is salt. In theory, "compromized verification keys (called v) are of little value to an attacker". However in practice, it is possible to brute-force the original password from the verification key at high speeds.

Ideally, for increassed resistance against brute-force attacks, a costly (slow) one-way hash function (H) like PBKDF2 should be used. However, in reality we have seen very fast hash functions (like single iterations of SHA1 or SHA256) being used (See[14] and [20]). This allows an attacker to mount brute-force attack at high-speeds.

The following snippet show how the salt and the verifier (verification key) are stored in the database,

Clipperz secret data

```
<script>_clipperz_dump_data_ = {  ...
    2f2134e38b23534adfcd43c2f7223caf3...': {
        s: 'e0bc11ee4db80a3ecabd293f...',
        v: 'e8be8c8d9c1d5dc79ecc7b15...',
        version: '0.2',
    }
    ...
}
```

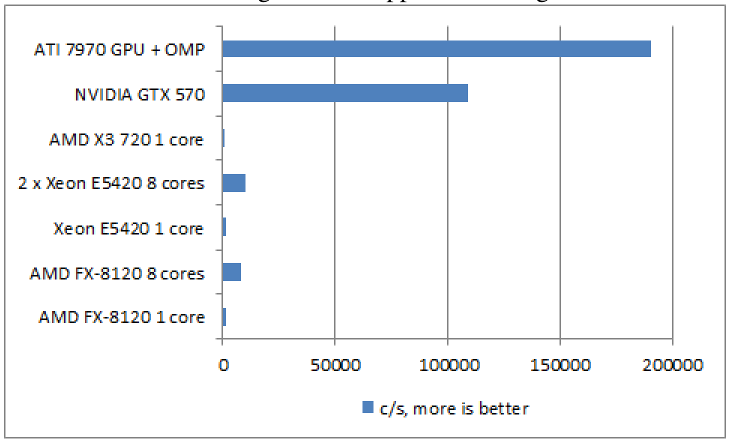
The following snippet shows how we can derive a verifer from a given salt and user password and check if the gives user password was correct,

Clipperz Cracker

```
n = A known safe prime

g = 2

# P algorithm

h1 = hashlib.sha256(password + username).digest()
P = hashlib.sha256(h1).hexdigest()

# x (Private Key) algorithm

x1 =  hashlib.sha256(s + P).digest()
x = hashlib.sha256(x1).hexdigest()

# v (Verification key) algorithm, v = g ^ x
# z_base = 2, z_mod = n
z_exp = BN_bin2bn(x, 32)
BN_mod_exp(z_rop, z_base, z_exp, z_mod);
BN_bn2bin(z_rop, output);

if output == v:
    print "Password␣is", password
```

For details see src/clipperz_fmt_plug.c in JtR source tree. We compare the performance of Clipperz cracker on different machines below,

Figure 30: Clipperz Cracking Benchmarks



Clipperz cracking benchmarks

```
$../run/john -fo:keepass -t # AMD FX-8120 (single core)
Benchmarking: KeePass SHA-256 AES [32/64 OpenSSL]... DONE
Raw:  82.0 c/s real, 82.0 c/s virtual

$../run/john -fo:keepass -t # AMD FX-8120 (8 cores)
Benchmarking: KeePass SHA-256 AES [32/64 OpenSSL]... (8xOMP) DONE
Raw:  409 c/s real, 51.2 c/s virtual

# GPU RESULTS, TODO
```

3.10 Analysis of Druva inSync

Clipperz is a popular free online password manager [14]. It does encryption on the local browser which gurantees confidentiality of data. Clipperz supports exporting encrypted databases into offline versions. Offline versions use the same cryptographic technology as used by the online version.

Clipperz does not believe in "security through obscurity" (unlike LastPass) and all the code behing Clipperz is open-source [**?**]. Clipperz uses SRP (Secure Remote Password protocol, see [17], [18] and [19]) for online and offline authentication. SRP is essentailly an authentication protocol for password-based, mutual authentication over an insecure network connection and requires both sides of the connection to have knowledge of the user's password. SRP offers security and deployment advantages over other challenge-response protocols, such as Kerberos and SSL, in that it does not require trusted key servers or certificate infrastructures. Instead, small verification keys derived from each user's password are stored and used by each SRP server application [19]. "SRP does not store plaintext passwords on the server side but instead uses what is known as a "non plaintext-equivalent verifier" [16].

Password verifier is derived from a Private key (called x) by using the formula v = g^x, where x (Private key) = H(s, H(I | ':' | p)), g is generator modulo N, I is username, p is cleartext password, H() is one-way hash function and s is salt. In theory, "compromized verification keys (called v) are of little value to an attacker". However in practice, it is possible to brute-force the original password from the verification key at high speeds.

Ideally, for increassed resistance against brute-force attacks, a costly (slow) one-way hash function (H) like PBKDF2 should be used. However, in reality we have seen very fast hash functions (like single iterations of SHA1 or SHA256) being used (See[14] and [20]). This allows an attacker to mount brute-force attack at high-speeds.

The following snippet show how the salt and the verifier (verification key) are stored in the database,

Clipperz secret data

```
<script>_clipperz_dump_data_ = {  ...
    2f2134e38b23534adfcd43c2f7223caf3...': {
        s: 'e0bc11ee4db80a3ecabd293f...',
        v: 'e8be8c8d9c1d5dc79ecc7b15...',
        version: '0.2',
    }
    ...
}
```

The following snippet shows how we can derive a verifer from a given salt and user password and check if the gives user password was correct,

Clipperz Cracker

```
n = A known safe prime

g = 2

# P algorithm

h1 = hashlib.sha256(password + username).digest()
P = hashlib.sha256(h1).hexdigest()

# x (Private Key) algorithm

x1 =  hashlib.sha256(s + P).digest()
x = hashlib.sha256(x1).hexdigest()

# v (Verification key) algorithm, v = g ^ x
# z_base = 2, z_mod = n
z_exp = BN_bin2bn(x, 32)
BN_mod_exp(z_rop, z_base, z_exp, z_mod);
BN_bn2bin(z_rop, output);

if output == v:
    print "Password␣is", password
```

For details see src/clipperz_fmt_plug.c in JtR source tree. We compare the performance of Clipperz cracker on different machines below,

4 Analysis of security of various password hashing algorithms

4.1 Analysis of RACF cracker

RACF (Resource Access Control Facility) is IBM security system that provides access control and auditing functionality for the z/OS and z/VM operating systems. This work is the only published source of complete RACF algorithm and RACF database parser. In addition, this work implements the only multi-core open-source RACF hash cracking software.

Code 25: RACF hashing algorithm

```python
def process_userid(s):
    while len(s) % 8:
        s.append(0x40)


def process_key(s):
    # replace missing characters in key by EBCDIC spaces (0x40)
    while len(s) % 8 or len(s) == 0:
        s.append(0x40)
    for i in range(0, 8):
        # secret sauce
        s[i] = ((s[i] ^ 0x55) << 1) & 0xff

# truncate password & username and encode to EBCDIC charset
ue = USERNAME[0:8].decode('ascii').encode('EBCDIC-CP-BE')
pe = PASSWORD[0:8].decode('ascii').encode('EBCDIC-CP-BE')
ues = bytearray(ue)
process_userid(ues)
pes = bytearray(pe)
process_key(pes)
pesj = str(pes)
uesj = str(ues)
des = DES.new(pesj, DES.MODE_CBC)

HASH = des.encrypt(uesj)
```

Research into the RACF system was done in collaboration with Nigel Pentland (author of CRACF) and Phil Young.

RACF cracker benchmarks

```
$ ../run/john -fo:racf -t # AMD FX-8120 (1 core)
Benchmarking: RACF DES [32/64]... cdDONE
Many salts: 2024K c/s real, 2024K c/s virtual
Only one salt:  1931K c/s real, 1931K c/s virtual

$ ../run/john -fo:racf -t # AMD FX-8120 (8 cores)
Benchmarking: RACF DES [32/64]... (8xOMP) DONE
Many salts: 8926K c/s real, 1134K c/s virtual
Only one salt:  6408K c/s real, 824801 c/s virtual

$ ../run/john -fo=racf -t # Xeon
Benchmarking: RACF DES [32/64]... DONE
Many salts:     1692K c/s real, 1692K c/s virtual
Only one salt:  1473K c/s real, 1473K c/s virtua

$ ../run/john -fo=racf -t # 2 x Xeon E5420 (8 cores)
Benchmarking: RACF DES [32/64]... (8xOMP) DONE
Many salts:     11213K c/s real, 1401K c/s virtual
Only one salt:  5664K c/s real, 709870 c/s virtual
```

4.2 Analysis of Django 1.4 password hashing algorithm

Earlier versions (< 1.4) of Django didn't use key-stretched hashing algorithms, instead they used single rounds of either SHA1, MD5 or DES crypt algorithms. Hence older Django hashes were vulnerable to brute-forcing at high speeds. Django 1.4 introduces a new flexible password storage system and uses PBKDF2 with SHA256 hash, a password stretching mechanism. By default 10, 000 iterations are used for key stretching.

Django Benchmarks

```
$ john -fo:raw-sha1 -t # AMD FX-8120 (1 core)
Benchmarking: Raw SHA-1 [128/128 SSE2 intrinsics 8x]... DONE
Raw:  17957K c/s real, 17957K c/s virtual

$ john -fo:django -t # AMD FX-8120
Benchmarking: Django PBKDF2-HMAC-SHA-256 (x10000) [32/64]... DONE
Raw:  46.0 c/s real, 46.0 c/s virtual

$ john -fo:django -t # AMD FX-8120 (all cores)
Benchmarking: Django PBKDF2-HMAC-SHA-256 (x10000) [32/64]... (8
    xOMP) DONE
Raw:  203 c/s real, 26.9 c/s virtual

$ ../run/john -fo=django -t # Xeon E5420 (1 core)
Benchmarking: Django PBKDF2-HMAC-SHA-256 (x10000) [32/64]... DONE
Raw:  34.3 c/s real, 34.3 c/s virtual

$ ../run/john -fo=django -t # 2 x Xeon E5420 (8 cores)
Benchmarking: Django PBKDF2-HMAC-SHA-256 (x10000) [32/64]... (8
    xOMP) DONE
Raw:  160 c/s real, 20.2 c/s virtual
```

Our single-core implementation of Django 1.4 achieves only 46 c/s on AMD FX-8120 CPU while the multi-core version achieves 203 c/s for a speedup of 4.7x. Overall cracking speed and mutli-core speedup factor can further be improved by using a custom implementation of PBKDF2-HMAC-SHA-256 algorithm instead of using high-level OpenSSL interfaces. One side-effect of using CPU intensive password hashing algorithms on servers (e.g. bcrypt, ph-pass, scrypt) is that it becomes trivial to mount a DoS (denial of service) attack on them. Since Django run on Python (which effectively uses a single CPU core for running Python code, due to GIL), such DoS attacks become even more trivial to mount against servers running Django.To avoid such attacks DoS attacks, care must be taken to implement policies which deny connection attempts after an IP has failed login process X number of times. This can be done using softwares like fail2ban. etc etc. (benchmark Django implementation and estimate the number of connections needed to DoS the site). Online attacks is to limit both per-IP attempts per second, and per-username attempts per second, with the limit being tripped causing an "automatic reject."

5 Conclusions

6 Related work (not described in this paper)

Some other JtR plug-in that were written (but not described in this paper) are RAdmin, SybaseASE, GOST, SIP, IKE PSK, Nuked Clan, MSSQL 12, wbb3, vms, WebEdition CMS

7 Future Work

Implement DES on GPU, this will benefit RAC format. Implement AES on GPU for KeePass format. GPU implementation of PBKDF-HMAC-WHIRLPOOL etc.

8 Acknowledgements

I would like to thanks Solar Designer for mentoring my GSoC 2011 (Google Summer of Code) work, Robert for collaborating on Mac OS X Keychain work, Lukas for GPU implementation of Password Safe format, Nigel for collaborating on RACF work, magnum for maintaing JtR-jumbo, Milen for collaborating on FileVault work.

References

[1] John the Ripper password cracker, http://www.openwall.com/john/

[2] Ettercap, http://ettercap.sourceforge.net/

[3] Nmap, http://nmap.org/

[4] Metasploit, http://www.metasploit.com/

[5] http://passwordsafe.sourceforge.net/

[6] http://passwordsafe.svn.sourceforge.net/viewvc/passwordsafe/trunk/pwsafe/pwsafe/docs/formatV3.txt?revision=49

[7] http://keybox.rubyforge.org/password-safe-db-format.html

[8] Password Gorilla, http://www.fpx.de/fp/Software/Gorilla/help.html#V3Format

[9] Pasaffe password manager, https://launchpad.net/pasaffe

[10] Key files in TrueCrypt, http://www.truecrypt.org/docs/?s=keyfiles

[11] Key files in KeePass, http://keepass.info/help/base/keys.html#keyfiles

[12] YubiKey Hardware, http://www.yubico.com/products/yubikey-hardware/

[13] Keychain (Mac OS), http://en.wikipedia.org/wiki/Keychain_%28Mac_OS%29

[14] http://www.opensource.apple.com/source/securityd/securityd-55111/doc/BLOBFORMAT

[15] https://matt.ucc.asn.au/src/extractkeychain-0.1/

[16] agilekeychain, https://bitbucket.org/gwik/agilekeychain

[1] http://learn.agilebits.com/1Password4/Security/keychain-design.html

[17] Forensic analysis of the Mac OS X keychain, https://www.dropbox.com/s/uge4ush72oqrrjq/diplomski.pdf

[18] B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Informational), September 2000. URL http://www.ietf.org/rfc/rfc2898.txt.

[19] R. Housley. Cryptographic Message Syntax. RFC 2630 (Proposed Standard), June 1999. URL http://www.ietf.org/rfc/rfc2630.txt. Obsoleted by RFCs 3369, 3370.

[20] http://help.agilebits.com/1Password3/agile_keychain_design.html

[21] http://blog.agilebits.com/2012/07/31/1password-is-ready-for-john-the-ripper/

[22] https://live.gnome.org/GnomeKeyring

[23] https://github.com/kholia/gkcrack

[24] http://fts.ifac.cnr.it/cgi-bin/dwww/usr/share/doc/gnome-keyring/file-format.txt

[26] KDE Wallet Managerhttp://utils.kde.org/projects/kwalletmanager/

[27] KWallet - The KDE Wallet System, http://events.kde.org/info/kastle/presentations/kwallet-kastle-2003.ps

[28] kwalletcrack, https://github.com/kholia/kwalletcrack

[2] http://en.wikipedia.org/wiki/Pretty_Good_Privacy

[3] Kerberos, http://en.wikipedia.org/wiki/Kerberos_%28protocol%29

[29] Schneier, B. 1994. Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish). In Fast Software Encryption, Cambridge Security Workshop (December 09 - 11, 1993). R. J. Anderson, Ed. Lecture Notes In Computer Science, vol. 809. Springer-Verlag, London, 191-204.

[29] Making a Faster Cryptanalytic Time-Memory Trade-Off, Philippe Oechslin, http://dblp.uni-trier.de/db/conf/crypto/crypto2003.html#Oechslin03

[30] KeePass Password Safe, http://keepass.info/

[31] KeePass's types of keys, http://keepass.info/help/base/keys.html

[32] http://keybox.rubyforge.org/keepassx-db-format.html

[33] KeePass's protection against dictionary attacks, http://keepass.info/help/base/security.html#secdictprotect

[34] P-ppk-crack, http://neophob.com/2007/10/putty-private-key-cracker/

[35] FileVault, http://en.wikipedia.org/wiki/FileVault

[36] VileFault, http://code.google.com/p/vilefault/

[37] Hashkill, http://www.gat3way.eu/hashkill/

[4] pgpry - PGP private key recovery, https://github.com/kholia/pgpry

[5] Fabrizio Milo, Massimo Bernaschi, and Mauro Bisson. 2011. A fast, GPU based, dictionary attack to OpenPGP secret keyrings. J. Syst. Softw. 84, 12 (December 2011), 2088-2096. DOI=10.1016/j.jss.2011.05.027 http://dx.doi.org/10.1016/j.jss.2011.05.027

[6] http://en.wikipedia.org/wiki/Personal_Storage_Table

[7] AES Encryption Information, http://www.winzip.com/aes_info.htm

[8] http://www.pkware.com/documents/casestudies/APPNOTE.TXT

[9] iSCSI, http://en.wikipedia.org/wiki/ISCSI

[10] PPP Challenge Handshake Authentication Protocol (CHAP), http://tools.ietf.org/html/rfc1994

[11] Storage Security, http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-Dwivedi-update.pdf

[12] LastPass, http://lastpass.com/

[13] http://media.blackhat.com/bh-eu-12/Belenko/bh-eu-12-Belenko-Password_Encryption-Slides.pdf

[14] Clipperz, http://www.clipperz.com/

[15] https://github.com/clipperz

[16] http://www.clipperz.com/users/marco/blog/2007/01/04/interview_tom_wu_inventor_srp_protocol

[17] http://en.wikipedia.org/wiki/Secure_remote_password_protocol

[18] http://srp.stanford.edu/doc.html

[19] http://packages.python.org/srp/srp.html

[20] http://www.opine.me/blizzards-battle-net-hack/

[21] Rivest, R. 1992 The MD5 Message-Digest Algorithm. RFC. RFC Editor.

[22] Analysis of leaked RockYou's password database (containing 32 million credentials) by Matt Weir, http://reusablesec.blogspot.com/2009/12/rockyou-32-million-password-list-top.html

[23] John the Ripper benchmarks, http://openwall.info/wiki/john/benchmarks

[24] ElcomSoft Distributed Password Recovery, http://www.elcomsoft.com/edpr.html

[13] Ivan Golubev's blog : http://www.golubev.com/blog/?p=94

[25]

[26] https://docs.djangoproject.com/en/1.4/topics/auth/#passwords

[27] http://www.truecrypt.org/docs/?s=volume-format-specification

[28] http://www.truecrypt.org/docs/?s=encryption-scheme

[29] See http://revelation.olasagasti.info/download.php for LUKS implementation in Python.