Project Zero

News and updates from the Project Zero team at Google

Wednesday, December 19, 2018

On VBScript

Posted by Ivan Fratric, Google Project Zero

Introduction

Vulnerabilities in the VBScript scripting engine are a well known way to attack Microsoft Windows. In order to reduce this attack surface, in Windows 10 Fall Creators Update, Microsoft disabled VBScript execution in Internet Explorer in the Internet Zone and the Restricted Sites Zone by default. Yet this did not deter attackers from using it - in 2018 alone, there have been at least two instances of 0day attacks using vulnerabilities in VBScript: CVE-2018-8174 and CVE-2018-8373. In both of these cases, the delivery method for the exploit were Microsoft Office files with an embedded object which caused malicious VBScript code to be processed using the Internet Explorer engine. For a more detailed analysis of the techniques used in these exploits please refer to their analysis by the original discoverers here and here.

Because of this dubious popularity of VBScript, multiple security researchers took up the challenge of finding (and reporting) other instances of VBScript vulnerabilities, including a number of variants of those vulnerabilities used in the wild. Notably, researchers working with the Zero day initiative discovered multiple instances of vulnerabilities relying on VBScript Class_Terminate callback and Yuki Chen of Qihoo 360 Vulcan Team discovered multiple variants of CVE-2018-8174 (one of the exploits used in the wild).

Search This Blog

Search

Pages

- Working at Project Zero

Archives

2019

- Windows Exploitation Tricks: Abusing the User-Mode... (Apr)
- Virtually Unlimited Memory: Escaping the Chrome Sa... (Apr)
- Splitting atoms in XNU (Apr)
- Windows Kernel Logic Bug Class: Access Mode Mismat... (Mar)
- Android Messaging: A Few Bugs Short of a Chain (Mar)
- The Curious Case of Convexity Confusion (Feb)

As a follow up to those events, this blog post tries to answer the following question: Despite all of the existing efforts from Microsoft and the security community, how easy is it to still discover new VBScript vulnerabilities? And how strong are Windows policies intended to stop these vulnerabilities from being exploited?

Even more VBScript vulnerabilities

The approach we used to find VBScript vulnerabilities was quite straightforward: We used the already published Domato grammar fuzzing engine and wrote a grammar that describes the built-in VBScript functions, various callbacks and other common patterns. This is the same approach we used successfully previously to find multiple vulnerabilities in the JScript scripting engine and it was relatively straightforward to do the same for VBScript. The grammar and the generator script can be found here.

This approach resulted in uncovering three new VBScript vulnerabilities that we reported to Microsoft and are now fixed. The vulnerabilities are interesting, not because they are complex, but precisely for the opposite reason: they are pretty straightforward (yet, somehow, still survived to this day). Additionally, in several cases, there are parallels that can be drawn between the vulnerabilities used in the wild and the ones we found.

To demonstrate this, before taking a look at the first vulnerability the fuzzer found, let's take a look at a PoC for the latest VBScript 0day found in the wild:

```
Class MyClass
 Dim array

 Private Sub Class_Initialize
   ReDim array(2)
 End Sub

 Public Default Property Get P
   ReDim preserve array(1)
```

- Examining Pointer Authentication on the iPhone XS (Feb)
- voucher_swap: Exploiting MIG reference counting in... (Jan)
- Taking a page from the kernel&#39;s book: A TLB issue ... (Jan)

2018

- On VBScript (Dec)
- Searching statically-linked vulnerable library fun... (Dec)
- Adventures in Video Conferencing Part 5: Where Do ... (Dec)
- Adventures in Video Conferencing Part 4: What Didn... (Dec)
- Adventures in Video Conferencing Part 3: The Even ... (Dec)
- Adventures in Video Conferencing Part 2: Fun with ... (Dec)
- Adventures in Video Conferencing Part 1: The Wild ... (Dec)
- Injecting Code into Windows Protected Processes us... (Nov)
- Heap Feng Shader: Exploiting SwiftShader in Chrome... (Oct)
- Deja-XNU (Oct)
- Injecting Code into Windows Protected Processes us... (Oct)
- 365 Days Later: Finding and Exploiting Safari Bugs... (Oct)

```
 End Property
End Class

Set cls = new MyClass
cls.array(2) = cls
```

Trend Micro has a more detailed analysis, but in short, the most interesting line is

```
cls.array(2) = cls
```

In it, the left side is evaluated first and the address of variable at `cls.array(2)` is computed. Then, the right side is evaluated, and because cls is an object of type `MyClass` which has a default property getter, it triggers a callback. Inside the callback, the array is resized and the address of the variable computed previously is no longer valid - it now points to the freed memory. This results in writing to a freed memory when the line above gets executed.

Now, let’s compare this sample to the PoC for the first issue we found:

```
Class MyClass
 Private Sub Class_Terminate()
   dict.RemoveAll
 End Sub
End Class

Set dict = CreateObject("Scripting.Dictionary")
Set dict.Item("foo") = new MyClass
dict.Item("foo") = 1
```

On the first glance, this might not appear all that similar, but in reality they are. The line that triggers the issue is

- A cache invalidation bug in Linux memory managemen... (Sep)
- OATmeal on the Universal Cereal Bus: Exploiting An... (Sep)
- The Problems and Promise of WebAssembly (Aug)
- Windows Exploitation Tricks: Exploiting Arbitrary ... (Aug)
- Adventures in vulnerability reporting (Aug)
- Drawing Outside the Box: Precision Issues in Graph... (Jul)
- Detecting Kernel Memory Disclosure – Whitepaper (Jun)
- Bypassing Mitigations by Attacking JIT Server in M... (May)
- Windows Exploitation Tricks: Exploiting Arbitrary ... (Apr)
- Reading privileged memory with a side-channel (Jan)

2017

- aPAColypse now: Exploiting Windows 10 in a Local N... (Dec)
- Over The Air - Vol. 2, Pt. 3: Exploiting The Wi-Fi... (Oct)
- Using Binary Diffing to Discover Windows Kernel Me... (Oct)
- Over The Air - Vol. 2, Pt. 2: Exploiting The Wi-Fi... (Oct)

```
dict.Item("foo") = 1
```

In it, once again, the left side is allocated first and the address of `dict.Item("foo")` is computed. Then, a value is assigned to it, but because there is already a value there it needs to be cleared first. Since the existing value is of the type `MyClass`, this results in a Class_Terminate() callback, in which the `dict` is cleared. This, once again, causes that the address computed when evaluating the left side of the expression now points to a freed memory.

In both of these cases, the pattern is:

1. Compute the address of a member variable of some container object
2. Assign a value to it
3. Assignment causes a callback in which the container storage is freed
4. Assignment causes writing to a freed memory

The two differences between these two samples are that:

1. In the first case, the container used was an array and in the second it was a dictionary
2. In the first case, the callback used was a default property getter, and in the second case, the callback was Class_Terminate.

Perhaps it was because this similarity with a publicly known sample that this variant was also independently discovered by a researcher working with Trend Micro's Zero Day Initiative and Yuki Chen of Qihoo 360 Vulcan Team. Given this similarity, it would not be surprising if the author of the 0day that was used in the wild also knew about this variant.

The second bug we found wasn't directly related to any 0days found in the wild (that we know about), however it is a classic example of a scripting engine vulnerability:

```
Class class1
 Public Default Property Get x
   ReDim arr(1)
 End Property
End Class
```

- Over The Air - Vol. 2, Pt. 1: Exploiting The Wi-Fi... (Sep)
- The Great DOM Fuzz-off of 2017 (Sep)
- Bypassing VirtualBox Process Hardening on Windows (Aug)
- Windows Exploitation Tricks: Arbitrary Directory C... (Aug)
- Trust Issues: Exploiting TrustZone TEEs (Jul)
- Exploiting the Linux kernel via packet sockets (May)
- Exploiting .NET Managed DCOM (Apr)
- Exception-oriented exploitation on iOS (Apr)
- Over The Air: Exploiting Broadcom's Wi-Fi Stack (P... (Apr)
- Notes on Windows Uniscribe Fuzzing (Apr)
- Pandavirtualization: Exploiting the Xen hypervisor... (Apr)
- Over The Air: Exploiting Broadcom's Wi-Fi Stack (P... (Apr)
- Project Zero Prize Conclusion (Mar)
- Attacking the Windows NVIDIA Driver (Feb)
- Lifting the (Hyper) Visor: Bypassing Samsung's Rea... (Feb)

```
set c = new class1
arr = Array("b", "b", "a", "a", c)
Call Filter(arr, "a")
```

In it, a `Filter` function gets called on an array. The `Filter` function walks the array and returns another array containing just the elements that match the specified substring (`"a"` in this case). Because one of the members of the input array is an object with a default property getter, this causes a callback, and in the callback the input array is resized. This results in reading variables out-of-bounds once we return from the callback into the implementation of the `Filter` function.

A possible reason why this bug survived this long could be that the implementation of the Filter function tried to prevent bugs like this by checking if the array size is larger (or equal) than the number of matching objects at every iteration of the algorithm. However, this check fails to account for array members that *do not* match the given substring (such as elements with the value of `"b"` in the PoC).

In their advisory, Microsoft (initially) incorrectly classified the impact of this issue as an infoleak. While the bug results in an out-of-bounds read, what is read out-of-bounds (and subsequently returned to the user) is a VBScript variable. If an attacker-controlled data is interpreted as a VBScript variable, this can result in a lot more than just infoleak and can easily be converted into a code execution. This issue is a good example of why, in general, an out-of-bounds read can be more than an infoleak: it always depends on precisely what kind of data is being read and how it is used.

The third bug we found is interesting because it is in the code that was already heavily worked on in order to address CVE-2018-8174 and the variants found by the Qihoo 360 Vulcan Team. In fact, it is possible that the bug we found was introduced when fixing one of the previous issues.

We initially became aware of the problem when the fuzzer generated a sample that resulted in a NULL-pointer dereference with the following (minimized) PoC:

```
Dim a, r
```

2016

- Chrome OS exploit: one byte overflow and symlinks (Dec)
- BitUnmap: Attacking Android Ashmem (Dec)
- Breaking the Chain (Nov)
- task_t considered harmful (Oct)
- Announcing the Project Zero Prize (Sep)
- Return to libstagefright: exploiting libutils on A... (Sep)
- A Shadow of our Former Self (Aug)
- A year of Windows kernel font fuzzing #2: the tech... (Jul)
- How to Compromise the Enterprise Endpoint (Jun)
- A year of Windows kernel font fuzzing #1: the resu... (Jun)
- Exploiting Recursion in the Linux Kernel (Jun)
- Life After the Isolated Heap (Mar)
- Race you to the kernel! (Mar)
- Exploiting a Leaked Thread Handle (Mar)
- The Definitive Guide on Win32 to NT Path Conversio... (Feb)
- Racing MIDI messages in Chrome (Feb)

```
Class class1
End Class

Class class2
 Private Sub Class_Terminate()
   set a = New class1
 End Sub
End Class

a = Array(0)
set a(0) = new class2
Erase a
set r = New RegExp
x = r.Replace("a", a)
```

Why does this result in a NULL-pointer dereference? This is what happens:

1. An array `a` is created. At this point, the type of `a` is an array.
2. An object of type `class2` is set as the only member of the array
3. The array `a` is deleted using the `Erase` function. This also clears all array elements.
4. Since `class2` defines a custom destructor, it gets called *during* `Erase` function call.
5. In the callback, we change the value of `a` to an object of type `class1`. The type of `a` is now an object.
6. Before `Erase` returns, it sets the *value* of variable `a` to NULL. Now, `a` is a variable with the type *object* and the value NULL.
7. In some cases, when `a` gets used, this leads to a NULL-pointer dereference.

But, can this scenario be used for more than a NULL-pointer dereference. To answer this question, let's look at step 5. In it, the value of `a` is set to an object of type `class1`. This assignment necessarily increases the reference count of a class1 object. However, later, the value of `a` is going to be set to NULL *without* decrementing the reference count. When the PoC above finishes executing, there will be an object of type

- Raising the Dead (Jan)

2015

- FireEye Exploitation: Project Zero's Vulnerability... (Dec)
- Between a Rock and a Hard Link (Dec)
- Windows Sandbox Attack Surface Analysis (Nov)
- Hack The Galaxy: Hunting Bugs in the Samsung Galax... (Nov)
- Windows Drivers are True'ly Tricky (Oct)
- Revisiting Apple IPC: (1) Distributed Objects (Sep)
- Kaspersky: Mo Unpackers, Mo Problems. (Sep)
- Stagefrightened? (Sep)
- Enabling QR codes in Internet Explorer, or a story... (Sep)
- Windows 10^H^H Symbolic Link Mitigations (Aug)
- One font vulnerability to rule them all #4: Window... (Aug)
- Three bypasses and a fix for one of Flash&#39;s Vector... (Aug)
- Attacking ECMAScript Engines with Redefinition (Aug)
- One font vulnerability to rule them all #3: Window... (Aug)

class1 somewhere in memory with a reference count of 1, but no variable will actually point to it. This leads us to a reference leak scenario. For example, consider the following PoC:

```
Dim a, c, i

Class class1
End Class

Class class2
 Private Sub Class_Terminate()
   set a = c
 End Sub
End Class

Set c = New class1
For i = 1 To 1000
 a = Array(0)
 set a(0) = new class2
 Erase a
Next
```

Using the principle described above, the PoC above will increase the reference count for variable `c` to 1000 when in reality only one object (variable `c`) will hold a reference to it. Since a reference count in VBScript is a 32-bit integer, if we increase it sufficient amount of times, it is going to overflow and the object might get freed when there are still references to it.

The above is not exactly true, because custom classes in VBScript have protection against reference count overflows, however this is not the case for built-in classes, such as RegExp. So, we can just use an object of type `RegExp` instead of `class1` and the reference count will overflow eventually. As every reference count increase requires a callback, “eventually” here could mean several hours, so the only realistic exploitation scenario would be someone opening a tab/window and forgetting to close it - not really an APT-

- One font vulnerability to rule them all #2: Adobe ... (Aug)
- One font vulnerability to rule them all #1: Introd... (Jul)
- One Perfect Bug: Exploiting Type Confusion in Flas... (Jul)
- Significant Flash exploit mitigations are live in ... (Jul)
- From inter to intra: gaining reliability (Jul)
- When ‘int’ is the new ‘short’ (Jul)
- What is a &quot;good&quot; memory corruption vulnerability? (Jun)
- Analysis and Exploitation of an ESET Vulnerability... (Jun)
- Owning Internet Printing - A Case Study in Modern ... (Jun)
- Dude, where’s my heap? (Jun)
- In-Console-Able (May)
- A Tale of Two Exploits (Apr)
- Taming the wild copy: Parallel Thread Corruption (Mar)
- Exploiting the DRAM rowhammer bug to gain kernel p... (Mar)
- Feedback and data-driven updates to Google’s discl... (Feb)
- (^Exploiting)\s*(CVE-2015-0318)\s*(in)\s*(Flash$) (Feb)
- A Token’s Tale (Feb)

style attack (unlike the previous bugs discussed) but still a good example how the design of VBScript makes it very difficult to fix the object lifetime issues.

Hunting for reference leaks

In an attempt to find more reference leaks issues, a simple modification was made to the fuzzer: A counter was added and, every time a custom object was created, in the class constructor, this counter was increased. Similarly, every time an object was deleted, this counter was decreased in the class destructor. When a sample finishes executing and all variables are clear, if this counter is larger than 0, this means there was a reference leak somewhere.

This approach immediately resulted in a variant to the previously described reference leak, which is almost identical but uses `ReDim` instead of `Erase`. Microsoft responded that they are considering this a duplicate of the `Erase` issue.

Unfortunately there is a problem with this approach that prevents it from discovering more interesting reference leak issues: The approach can't distinguish between "pure" reference leak issues and reference leak issues that are also memory leak issues and thus don't necessarily have the same security impact. One example of issues this approach gets stuck on are circular references (imagine that object A has a reference to object B and object B also has reference to object A). However, we still believe that finding reference leaks can be automated as described later in this blog post.

Bypassing VBScript execution policy

As mentioned in the introduction, in Windows 10 Fall Creators Update, Microsoft disabled VBScript execution in Internet Explorer in the Internet Zone and the Restricted Sites Zone by default. This is certainly a step in the right direction. However, let's also examine the weaknesses in this approach and its implementation.

Firstly, note that, by default, this policy only applies to the Internet Zone and the Restricted Sites Zone. If a script runs (or an attacker can make it run) in the Local Intranet Zone or the Trusted Sites Zone, the policy simply does not apply. Presumably this is to strike a balance between the security for the home users and

- Exploiting NVMAP to escape the Chrome sandbox - CV... (Jan)
- Finding and exploiting ntpd vulnerabilities (Jan)

2014

- Internet Explorer EPM Sandbox Escape CVE-2014-6350... (Dec)
- pwn4fun Spring 2014 - Safari - Part II (Nov)
- Project Zero Patch Tuesday roundup, November 2014 (Nov)
- Did the "Man With No Name" Feel Insecure? (Oct)
- More Mac OS X and iPhone sandbox escapes and kerne... (Oct)
- Exploiting CVE-2014-0556 in Flash (Sep)
- The poisoned NUL byte, 2014 edition (Aug)
- What does a pointer look like, anyway? (Aug)
- Mac OS X and iPhone sandbox escapes (Jul)
- pwn4fun Spring 2014 - Safari - Part I (Jul)
- Announcing Project Zero (Jul)

business users that still rely on VBScript on their local intranet. However, it is somewhat debatable whether leaving potential gaps in the end-user security vs. having (behind-the-times) businesses that still rely on VBScript change a default setting strikes the right balance. In the future, we would prefer to see VBScript completely disabled by default in the Internet Explorer engine.

Secondly, when implementing this policy, Microsoft forgot to account for some places where VBScript code can be executed in Internet Explorer. Specifically, Internet Explorer supports MSXML object that has the ability to run VBScript code in XSL Transformations, for example like in the code below.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns:msxsl="urn:schemas-microsoft-com:xslt"
     xmlns:user="http://mycompany.com/mynamespace">

<msxsl:script language="vbscript" implements-prefix="user">
Function xml(str)
          a = Array("Hello", "from", "VBscript")
          xml = Join(a)
End Function
</msxsl:script>

<xsl:template match="/">
  <xsl:value-of select="user:xml(.)"/>
</xsl:template>

</xsl:stylesheet>
```

Microsoft did not disable VBScript execution for MSXML, even for websites running in the Internet Zone. This issue was reported to Microsoft and fixed at the time of publishing this blog post.

You might think that all of these issues are avoidable if Internet Explorer isn't used for web browsing, but unfortunately the problem with VBScript (and IE in general) runs deeper than that. Most Windows applications that render web content do it using the Internet Explorer engine, as is the case with Microsoft Office that was used in the recent 0days. It should be said that, earlier this year, Microsoft disabled VBScript creation in Microsoft Office (at least the most recent version), so this popular vector has been blocked. However, there are other applications, including those from third parties, that also use IE engine for rendering web content.

Future research ideas

During this research, some ideas came up that we didn't get around to implement. Rather than sitting on them, we'll list them here in case a reader looking for a light research project wants to pick one of them up:

- Combine VBScript fuzzer with the JScript fuzzer in a way that allows VBScript to access JScript objects/functions and vice-versa. Perhaps issues can be found in the interaction of these two engines. Possibly callbacks from one engine (e.g. default property getter from VBScript) can be triggered in unexpected places in the other engine.

- Create a better tool for finding reference leaks. This could be accomplished by running IE in the debugger and setting breakpoints on object creation/deletion to track addresses of live objects. Afterwards, memory could be scanned similarly to how it was done here to find if there are any objects alive (with reference count >0) that are not actually referenced from anywhere else in the memory (note: Page Heap should be used to ensure there are no stale references from freed memory).

- Other objects. During the previous year, a number of bugs were found that rely on `Scripting.Dictionary` object. `Scripting.Dictionary` is not one of the built-in VBScript objects, but rather needs to be instantiated using CreateObject function. Are there any other objects available from VBScript that would be interesting to fuzz?

Conclusion

VBScript is a scripting engine from a time when a lot of today's security considerations weren't in the forefront of anyone's thoughts. Because of this, it shouldn't be surprising that it is a crowd favorite when it comes to attacking Windows systems. And although it received a lot of attention from the security community recently, new vulnerabilities are still straightforward to find.

Microsoft made some good steps in attack surface reduction recently. However in combination with an execution policy bypass and various applications relying on Internet Explorer engine to render web content, these bugs could still endanger even users using best practices on up-to-date systems. We hope that, in the future, Microsoft is going to take further steps to more comprehensively remove VBScript from all security-relevant contexts.

Posted by Ben at 10:41 AM No comments:

Tuesday, December 18, 2018

Searching statically-linked vulnerable library functions in executable code

Helping researchers find 0ld days

Posted by Thomas Dullien, Project Zero

Executive summary

Software supply chains are increasingly complicated, and it can be hard to detect statically-linked copies of vulnerable third-party libraries in executables. This blog post discusses the technical details of an Apache-licensed open-source library to detect code from other open-source libraries in executables, along with some real-world findings of forked open-source libraries in real-world software.

Technical blog post

Permissive open-source licenses have been a great benefit for the IT industry: Most common problems can be addressed using high-quality libraries under permissive licenses (such as BSD or Apache) which can be easily re-used and integrated.

This has been great for everybody.

Nevertheless, statically linking third-party code requires special care: Vulnerabilities in third-party code are discovered regularly, and this implies updating your binaries. Things get harder when changes in upstream libraries need to be merged into local forks of the code. Not all organisations get this right, and even companies with mature secure development processes fumble sometimes.

For the reverse engineer and offensive researcher, identifying vulnerable statically linked software libraries is both an opportunity and a challenge:

- An opportunity since it can provide a way to obtain a vulnerability in a target without having to do the hard work of identifying a new one.
- A challenge since the available tooling for performing this task is exceptionally poor: The standard way is usually a combination of “searching for known strings”, “educated guess”, and sometimes the use of BinDiff (a tool that was designed for a very different purpose).

The technical problem can be phrased as “efficiently performing a fuzzy search into a relatively large space of possible functions”. Fuzzy search is a requirement because compiler differences, optimization changes, and code changes contribute to add “noise” to the code in question.

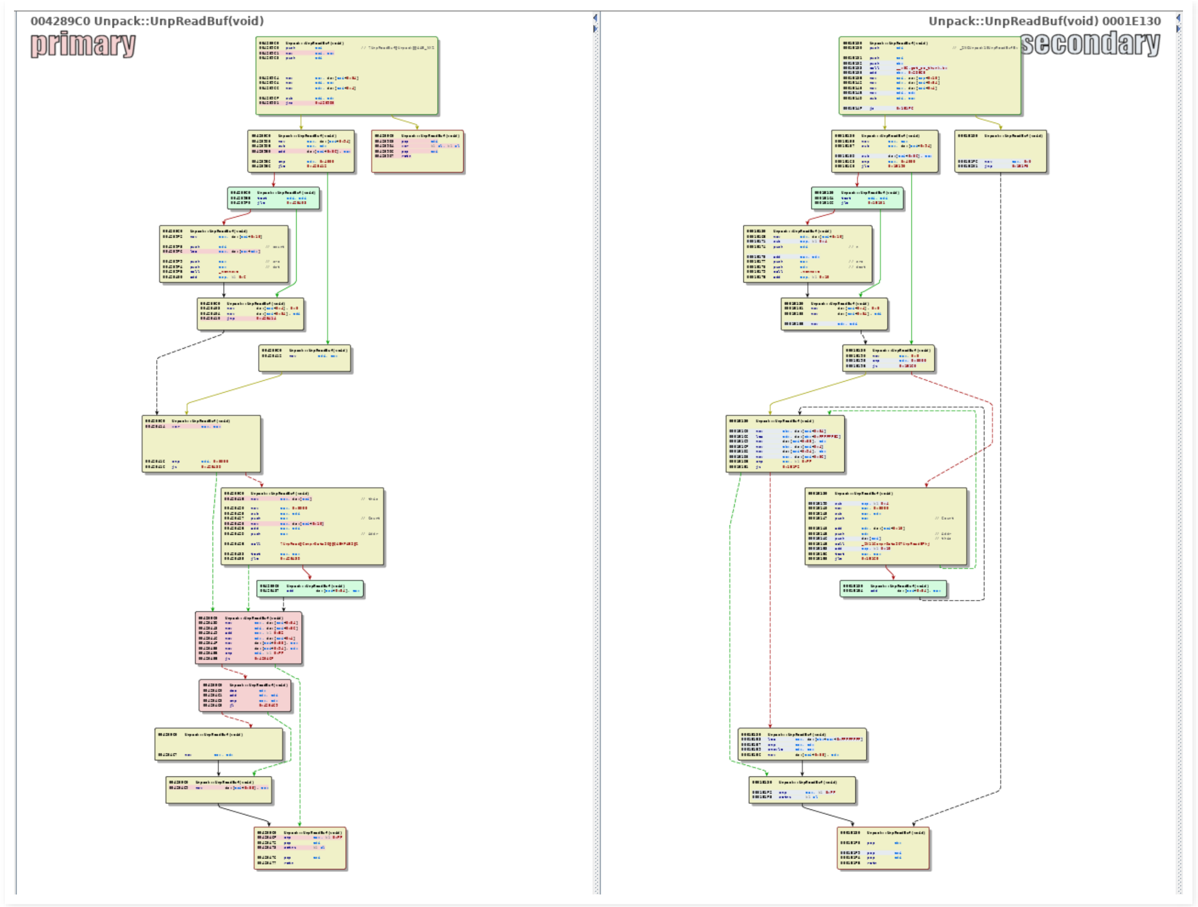
On the side of academic research, several interesting papers (CCS ‘16, CCS ‘17 have proposed sophisticated machine-learning-based methods to combine code embeddings with approximate nearest neighbor searches. They calculate a representation of code in R^n, and then search for nearby points to identify good candidates. While these approaches look powerful and sophisticated, public implementations do not exist, and adoption among practitioners has not happened. On the practical side, real-world use has been derived from CFG-focused algorithms such as MACHOC - but with the downside of being not tolerant to structural changes and not allowing for any “learning” of distances. Recently at (SSTIC ‘18) a neural-network based approach has been presented, with an announcement of making the code available in the next months.

This file describes FunctionSimSearch - an Apache-licensed C++ toolkit with Python bindings which provides three things:

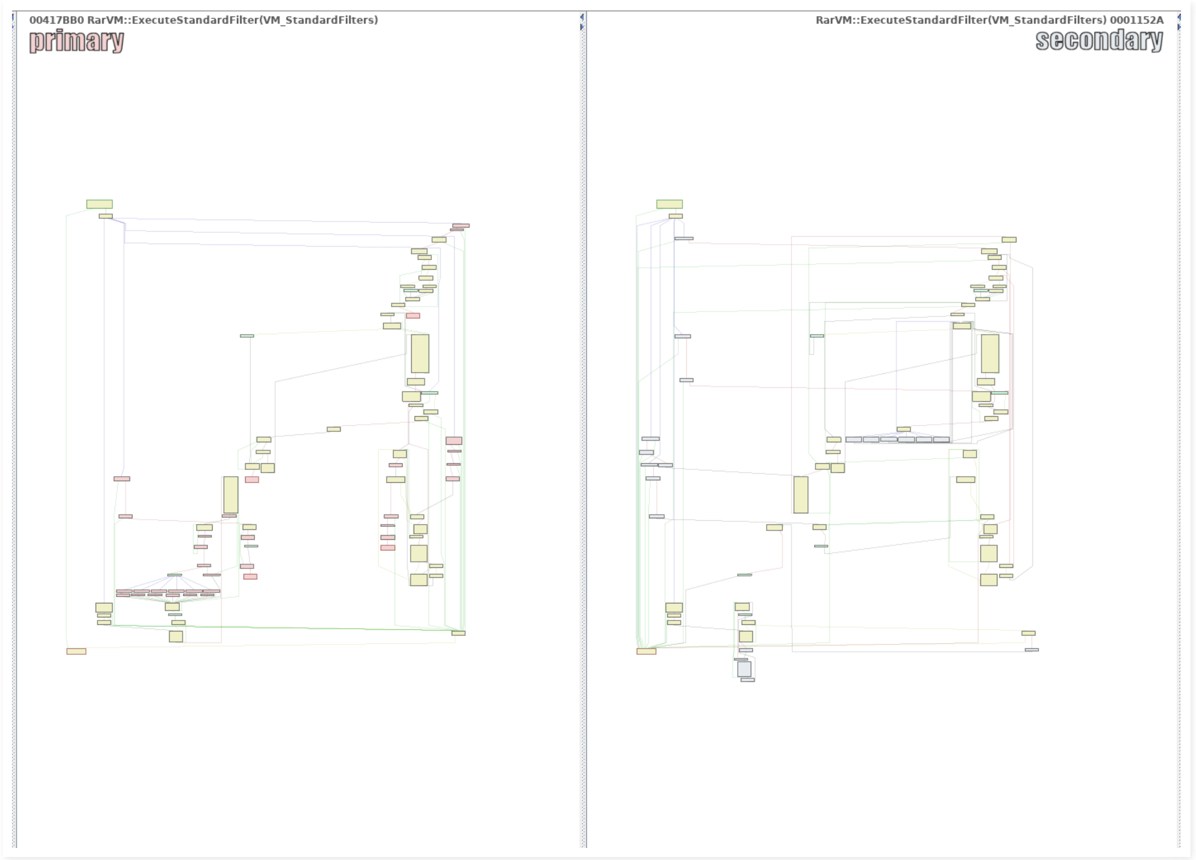
1. An efficient implementation of a hash function (based on SimHashing) which calculates a 128-bit hash from disassembled functions - and which preserves similarity (e.g. “distance” of two functions can be calculated by simply calculating the hamming distance between two hashes - which translates to two XOR and two POPCNT instructions on x64).
2. An efficient search index to allow approximate nearest neighbor search for 128-bit hashes.
3. Some supervised machine learning code that can be used to “learn” a good hash function from human-provided examples - given a set of functions that should be similar according to the hash, and a set of functions that should be dissimilar, the code learns a hash function that attempts to respect these examples.

The need for good fuzzy matching

Every reverse engineer has encountered that different compilers and compiler settings can generate drastically different pieces of assembly. Compilers can alter the CFG significantly, move code around, and even when they do not inline aggressively or unroll loops, decisions about code duplication, instruction movement and scheduling etc. lead to very different disassemblies:


004289C0 Unpack::UnpReadBuf(void)
primary
Unpack::UnpReadBuf(void) 0001E130
secondary

(Visual Studio 2015 unrar code vs. gcc 6.4.0 O1 optimized code)


00417BB0 RarVM::ExecuteStandardFilter(VM_StandardFilters)
primary
RarVM::ExecuteStandardFilter(VM_StandardFilters) 0001152A
secondary

(Visual Studio 2015 unrar code vs. gcc 6.4.0 O1 optimized code)

It is obvious that a good method to identify a function in the presence of changes is needed, and that both instruction-level and graph-level changes need to be dealt with.

Understanding the SimHash algorithm and what it provides

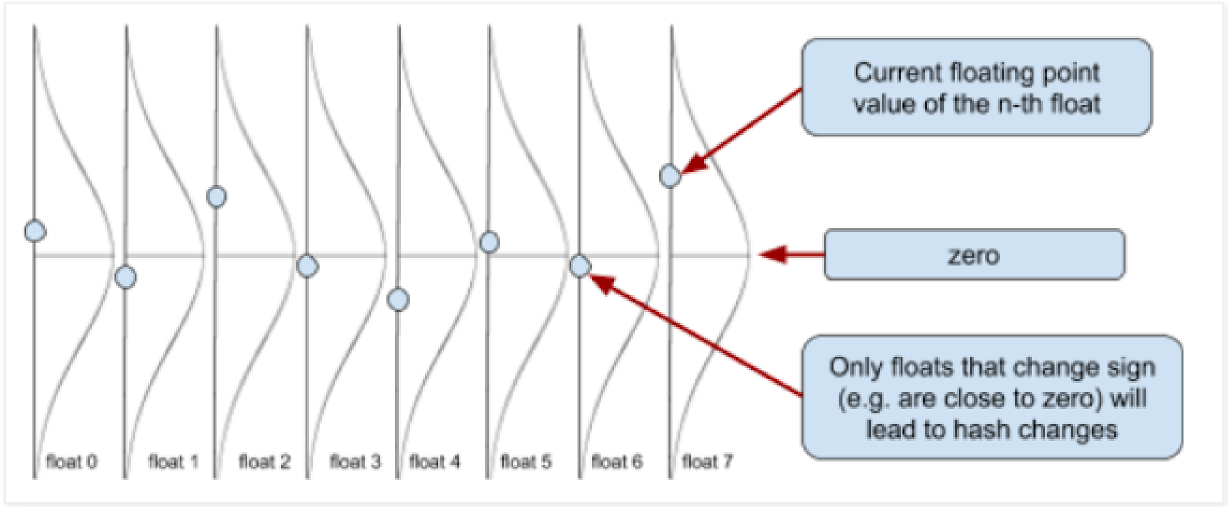
SimHashing was introduced in a paper by Moses Charikar, originally in the context of web page de-duplication. It is part of a family of algorithms and concepts called “locality-sensitive hashing”; a concept we will return to later.

The algorithm itself helps condense a set of values into a hash, with the property that the Hamming distance between the two hashes approximates the set similarity between the original sets. This makes estimating the set similarity between two sets blazingly fast (just an XOR and a POPCOUNT operation).

How does it work? Given a set of features as input (which are random 128-bit vectors themselves - if they are not, hash them), calculate a final hash output as follows:

1. Initialize a vector of 128 floating-point values to all zeroes.
2. For each feature in the input set, do:
    a. If bit n of the feature is 0, subtract 1 from the n-th floating-point value
    b. If bit n of the feature is 1, add 1 to the n-th floating point value
3. Convert the vector of floats to a 128-bit vector by mapping positive values to 1, negative values to 0.

Why does this produce a similarity-preserving hash? The intuition can be obtained by imagining what a minor change in the input set would do to the vector of floats: The values of these vectors will be approximately normally distributed with mean 0 and variance ¼ times the number of features.


Current floating point value of the n-th float
zero
Only floats that change sign (e.g. are close to zero) will lead to hash changes
float 0
float 1
float 2
float 3
float 4
float 5
float 6
float 7

Some of the values will be close to zero, either negative or positive. By changing the sets slightly (adding or removing a few features), there is some probability of individual values crossing over from positive into negative territory or vice versa. This probability goes up as more of the set changes; for small changes, the odds of many bits flipping is comparatively low.

It is important to note that the result will always be a 128-bit vector, irrespective of the size of the set-of-features from which it was calculated.

What is a good set of input features for our comparison? Ideally we would want to extract features that are representative of what we deem “similar”; e.g. two functions that are compiled from the same source code should have similar (overlapping) sets of features. It is somewhat involved to algorithmically design such features, so for the moment, the features in question are extremely simple: Subgraphs of the control-flow graph, n-grams of mnemonics of disassembled instructions, and constants from operands. In a naive implementation, all features have unit weight - e.g. every feature contributes the same to the final hash. This is clearly not ideal - a function prologue is not very indicative of the similarity between two functions - and we will improve this later in this post. Other non-implemented ideas for more features will be discussed at the end of the document.

A simple approximate nearest-neighbor search for hashes

With a way of calculating a similarity-preserving hash for a given input function, how do we search non-trivially sized corpora of such hashes for the “most similar” hash?

The answer lies in a second application of locality-sensitive hashing. If one can construct a family of hash functions so that the probability of two nearby points getting hashed into the same hash bucket is higher than the probability of two distant points getting hashed into the same bucket, one can construct a relatively efficient nearest-neighbor search: Simply use k different hash functions of the family to map inputs to buckets of candidates and process the candidates.

Choosing random bits as locality-sensitive hashes

Since our inputs are bit vectors, the easiest way to build such a hash function is to simply subsample bits from the vector. This has the nice property that a single random bit-level permutation of the input is enough to construct a hash family: In order to construct k different hashes, apply the bit-level permutation k times to your input and take the first few bits. Bitwise permutations on 128 bits are cheap-ish in software and close to free in hardware; the permutation chosen in the codebase should execute in ~65 cycles on a modern CPU.

Choice of data structures

The underlying data structure is an ordered collection of tuples of the form:

<PermutationIndex, k-th-permutation-of-input-hash, result-id>

Performing a binary search using the tuple <k, perm_k(input) & (0xFFL << 56), 0> will give us the hash bucket for a given permutation index and input value. We perform k such searches, and for each hash bucket we add all elements to a candidate list. The hamming distance between each candidate and the input hash is calculated, and the results can be returned in the order of their hamming distance.

A maximally memory- and cache-efficient version of this would simply use a sorted flat array / vector of such tuples; for our purposes (and for efficient insertion) the existing C++ code uses the equivalent of a std::set container, made persistent using a memory-mapped file as storage.

Learning a SimHash from examples

One of the problems with the described approach can immediately be identified: Every feature in the input set is treated with equal importance. In reality, though, features have vastly different importance. Luckily, it is easy to incorporate the importance of individual features into the calculation of a SimHash: Instead of adding +1 or -1 into the vector of floats, one could add or subtract a feature-specific weight.

But how does one infer good weights from the training data? Can we automatically “learn” what features will be preserved across compiler changes, with some predictive power?

Using the cheap gradient principle

The workhorse of modern machine learning is automatic differentiation. In simple terms, automatic differentiation provides the “cheap gradient principle” -- which can be paraphrased as “if you can calculate a function from R^n to R, you can calculate the gradient of this function with moderate overhead”. This means that if we can specify a loss function involving our weights, we can try to minimize this loss function. While we won’t have any guarantees of convergence, odds are we can learn weights from examples.

So what we need is a bunch of labelled data (ideally pairs of functions labelled “same” or “not the same”), and a good loss function.

Choosing a loss function for the SimHash distance

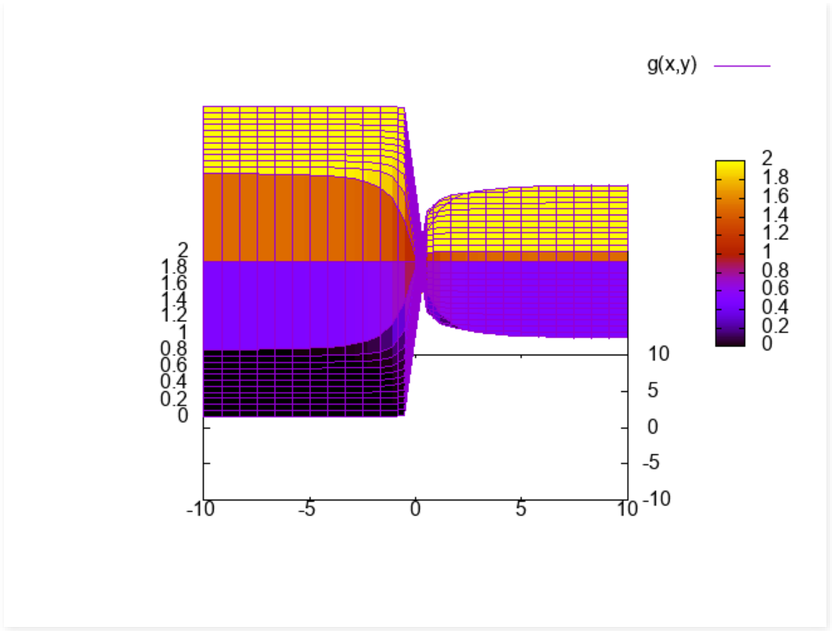
Building a loss function for our distance requires a slight bit of care. Since our final distance is a Hamming distance between two bit vectors, the gradient of this distance is likely to be zero - stepwise functions have many “flat sections” for which we cannot get a useful gradient.

The simplest idea would be to remove the last step of the hash calculation - instead of comparing the hashes that we derive from the vector-of-floats, one could measure the Euclidian distance on the final vectors-of-floats. Unfortunately, this creates “perverse incentives” for the optimizer: The simplest way to make two “similar” functions close to each other would be to shrink weights that occur in both to zero.

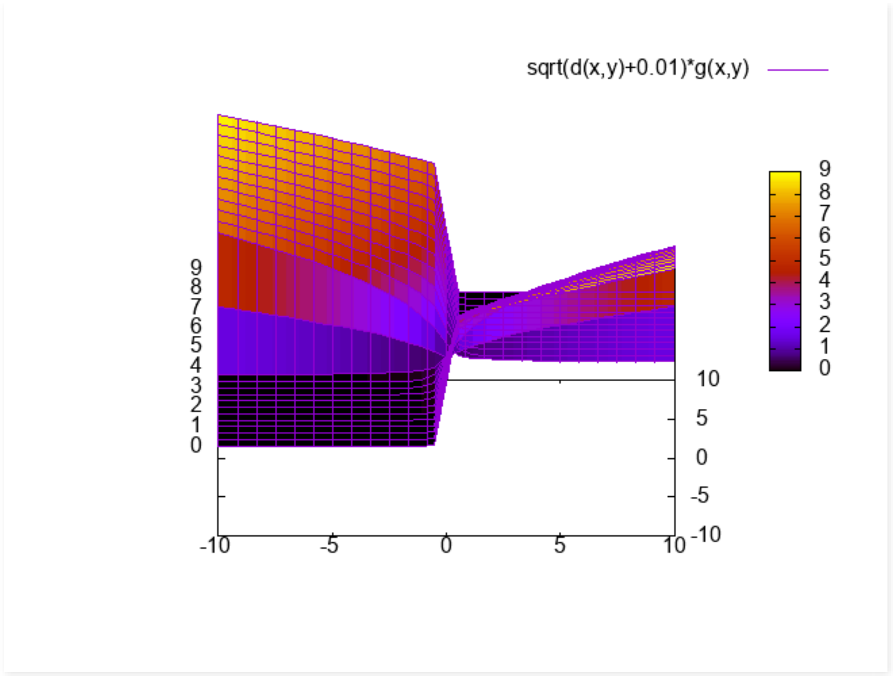
So ideally we want something that “penalizes” when pairs of similar functions with large distance and pairs of dissimilar functions with low distance.

We need a function that is positive when two real values do not have the same sign, and zero (or negative) if the two real values that have the same sign. Ideally, it should also provide a slope / incentive to move inputs in the direction of “same sign”.

We start with a simple smoothed step function g(x,y) := - xyx2y2+1.0+1.0:



This function has high loss when the sign of x and y is different, and zero loss when it is the same. Unfortunately, it is also flat on most of the surface, so we need to somehow skew the flat regions to point into the right direction. So we multiply with d(x,y) :=(x-y)2+0.01



This function satisfies our requirements: It provides a way to move parameters in the desired direction, punishes unequal signs, and has zero loss if x and y have equal sign.

In summary: For a given pair of real vectors (each obtained by calculating the hash function without the last step of converting to a binary hash) we can simply sum the loss for each vector entry. We now have a loss function that we can use to adjust our parameters from examples.

Generating training data

Generating training data should - at least in theory - be simple. It should be sufficient to compile some open-source code with a number of different compilers and compiler settings, and then parse the symbol information to create groups of “function variants” - e.g. multiple different compiler outputs for the same C/C++ function. Similarly, known-dissimilar-pairs can be generated by simply taking two random functions with different symbols.

Unfortunately, theory is not practice, and a number of grimy implementation issues come up, mostly around symbol parsing and CFG reconstruction.

Real-world problems: Symbols

One problem arises from the non-availability of good cross-platform tooling for parsing different versions of the PDB file format - which naturally arise when many different versions of Visual Studio are used - and the difficulty of reliably building the same open-source codebase for many different compilers. While GCC and CLANG are often drop-in-replaceable, projects that build without intervention on both Visual Studio, GCC, and CLANG are much more rare.

The (unfortunate) solution to the PDB parsing issue is “giving up” - the codebase expects the PDB information to have been dumped to a text file. More on this below.

The (equally unfortunate) solution to the issue of building the same codebase reliably with Visual Studio and GCC is also “giving up” - it is up to the user of FunctionSimSearch to get things built.

Other problems arise by different mangling conventions for C++ code, and different conventions in different compilers affecting how exactly a function is named. This is solved by a hackish small tool that removes type information from symbols and tries to “unify” between GCC/CLANG and Visual Studio notation.

Real-world problems: Reliably generating CFGs, and polluted data sets

Obtaining CFGs for functions should be simple. In practice, none of the tested disassemblers correctly disassembles switch statements across different compilers and platforms: Functions get truncated, basic blocks mis-assigned etc. The results particularly dire for GCC binaries compiled using -fPIC of -fPIE, which, due to ASLR, is the default on modern Linux systems.

The net result is polluted training data and polluted search indices, leading to false positives, false negatives, and general frustration for the practitioner.
While the ideal fix would be more reliable disassembly, in practice the fix is careful investigation of extreme size discrepancies between functions that should be the same, and ensuring that training examples are compiled without PIC and PIE (`-fno-pie -fno-PIE -fno-pic -fno-PIC` is a useful set of build flags).

Data generation in practice

In practice, training data can be generated by doing:

```
cd ./testdata
     ./generate_training_data.py --work_directory=/mnt/training_data
```

The script parses all ELF and PE files it can find in the ./testdata/ELF and ./testdata/PE directories. For ELF files with DWARF debug information, it uses objdump to extract the names of the relevant functions. For PE files, I was unfortunately unable to find a good and reliable way of parsing a wide variety of PDB files from Linux. As a result, the script expects a text file with the format "<executable_filename>.debugdump" to be in the same directory as each PE executable. This text file is expected to contain the output of the DIA2Dump sample file that ships with Visual Studio.

The format of the generated data is as follows:

```
./extracted_symbols_<EXEID>.txt
./functions_<EXEID>.txt
     ./[training|validation]_data_[seen|unseen]/attract.txt
     ./[training|validation]_data_[seen|unseen]/repulse.txt
     ./[training|validation]_data_[seen|unseen]/functions.txt
```

Let's walk through these files to understand what we are operating on:

1. The `./extracted_symbols_<EXEID>.txt` files:
   Every executable is assigned an executable ID - simply the first 64 bit of it's SHA256. Each such file describes the functions in the executable for which symbols are available, in the format:
   `[exe ID] [exe path] [function address] [base64 encoded symbol] false`
2. The `./functions_<EXEID>.txt` files:
   These files contain the hashes of the extracted features for each function in the executable in question. The format of these files is:
   `[exe ID]:[function address] [sequence of 128-bit hashes per feature]`

3. The `./[training|validation]_data_[seen|unseen]/attract.txt` and `./repulse.txt` files:
   These files contain pairs of functions that should repulse / attract, the format is simply `[exe ID]:[function address] [exe ID]:[function address]`
4. The `./[training|validation]_data_[seen|unseen]/functions.txt` files:
   A file in the same format as the `./functions_<EXEID>.txt` files with just the functions referenced in the corresponding `attract.txt` and `repulse.txt`.

Two ways of splitting the training / validation data

What are the mysterious `training_data_seen` and `training_data_unseen` directories? Why does the code generate multiple different training/validation splits? The reason for this is that there are two separate questions we are interested in:

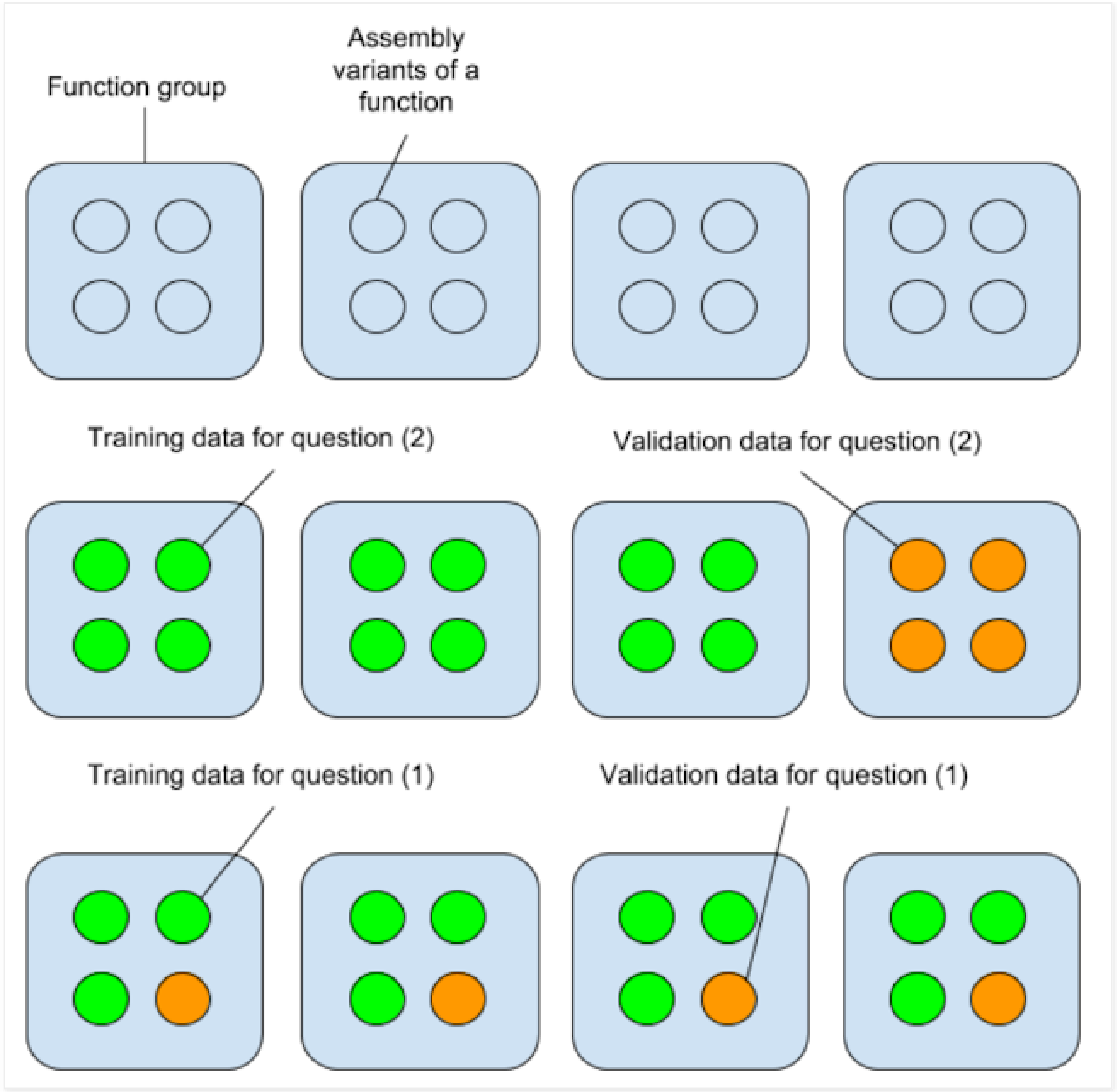
1. Does the learning process improve our ability to detect variants of a function we have trained on?
2. Does the learning process improve our ability to detect variants of a function, even if no version of that function was available at training time?

While (2) would be desirable, it is unlikely that we can achieve this goal. For our purposes (detection of statically linked vulnerable libraries), we can probably live with (1). But in order to answer these questions meaningfully, we need to split our training and validation data differently.

If we wish to check for (2), we need to split our training and validation data along “function group” lines: A “function group” being a set of variant implementations of the same function. We then need to split off a few function groups, train on the others, and use the groups we split off to validate.

On the other hand, if we wish to check for (1), we need to split away random variants of functions, train on the remainder, and then see if we got better at detecting the split-off functions.

The differences in how the training data is split is best illustrated as follows:



Function group
Assembly variants of a function
Training data for question (2)
Validation data for question (2)
Training data for question (1)
Validation data for question (1)

Implementation issues of the training

The industry-standard approaches for performing machine learning are libraries such as TensorFlow or specialized languages such as Julia with AutoDiff packages. These come with many advantages -- most importantly, automated parallelization and offloading of computation to your GPU.
Unfortunately, I am very stubborn -- I wanted to work in C++, and I wanted to keep the dependencies extremely limited; I also wanted to specify my loss function directly in C++. As a result, I chose to use a C++ library called SPII which allows a developer to take an arbitrary C++ function and minimize it. While this offers a very clean and nice programming model, the downside is "CPU-only" training. This works, but is uncomfortably slow, and should be replaced with a GPU-based version.

Running the actual training process

Once the training data is available, running the training process is pretty straightforward:

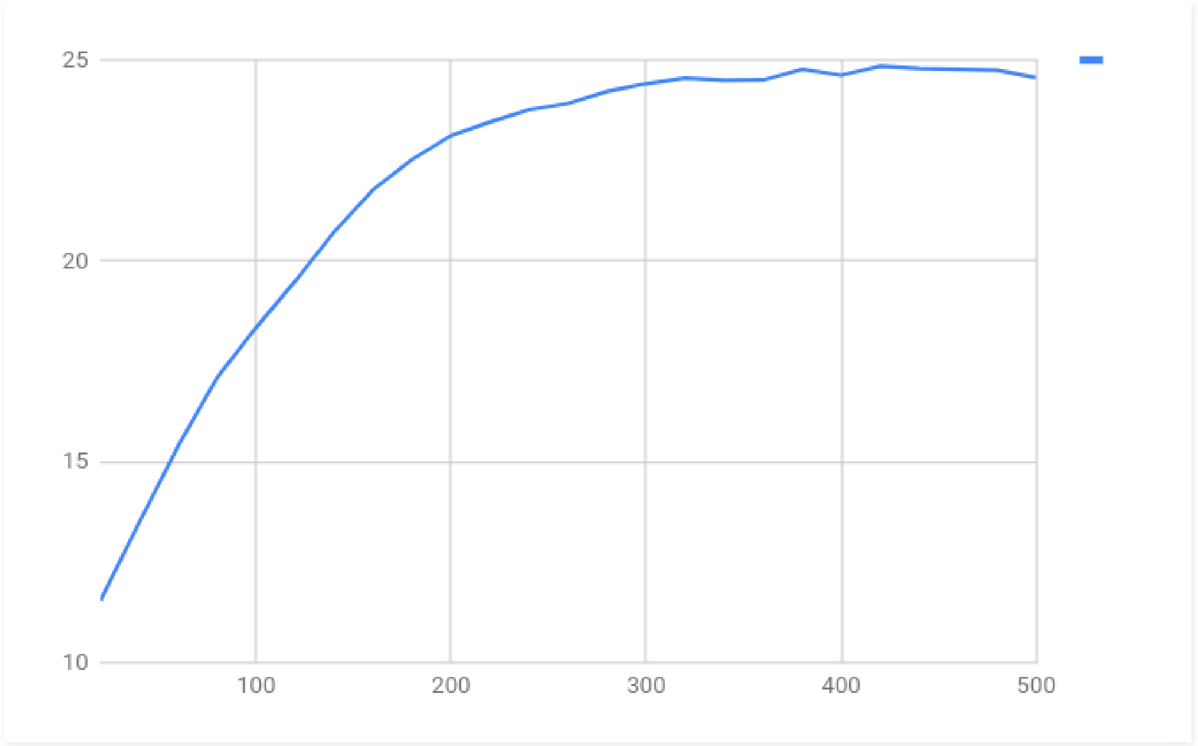
```
thomasdullien@machine-learning-training:~/sources/functionsimsearch/bin$ ./trainsimhashweights -
data=/mnt/training_data/training_data_seen/ --weights=weights_seen.txt
[!] Parsing training data.
[!] Mapping functions.txt
[!] About to count the entire feature set.
[!] Parsed 1000 lines, saw 62601 features ...
[!] Parsed 2000 lines, saw 104280 features ...
(...)
[!] Parsed 12000 lines, saw 270579 features ...
[!] Processed 12268 lines, total features are 271653
[!] Iterating over input data for the 2nd time.
[!] Loaded 12268 functions (271653 unique features)
[!] Attraction-Set: 218460 pairs
[!] Repulsion-Set: 218460 pairs
[!] Training data parsed, beginning the training process.
Itr       f deltaf   max|g_i| alpha     H0 rho
  0 +7.121e+04       nan 4.981e+02 2.991e-06 1.000e+00 0.000e+00
  1 +7.119e+04 2.142e+01 5.058e+02 1.000e+00 1.791e-06 3.114e-01
  2 +7.101e+04 1.792e+02 3.188e+02 1.000e+00 2.608e-05 5.735e-03
  3 +7.080e+04 2.087e+02 2.518e+02 1.000e+00 4.152e-05 4.237e-03
```

```
   4 +7.057e+04 2.271e+02 2.757e+02 1.000e+00 5.517e-05 4.469e-03
    ...
```

A few days later, the training process will have performed 500 iterations of L-BFGS while writing snapshots of the training results every 20 steps into our current directory (20.snapshot … 480.snapshot). We can evaluate the results of our training:

```
$ for i in *.snapshot; do foo=$(./evalsimhashweights --data /mnt/june2/validation_data_seen/ --weights $i
| grep \(trained\)); echo $i $foo; done
```

This provides us with the “difference in average distance between similar and dissimilar pairs” in the validation data: the code calculates the average distance between similar pairs and between dissimilar pairs in the validation data, and shows us the difference between the two. If our training works, the difference should go up.



We can see that somewhere around 420 training steps we begin to over-train - our difference-of-means on the validation set starts inching down again, so it is a good idea to stop the optimization process. We can also see that the difference-in-average-distance between the “similar” and “dissimilar” pairs has gone up from a bit more than 10 bits to almost 25 bits - this seems to imply that our training process is improving our ability to recognize variants of functions that we are training on.

Understanding the results of training

There are multiple ways of understanding the results of the training procedure:

1. Given that we can easily calculate distance matrices for a set of functions, and given that there are popular ways of visualizing high-dimensional distances (t-SNE and MDS), we can see the effects of our training visually.
2. Several performance metrics exist for information-retrieval tasks (Area-under-ROC-curve).

3. Nothing builds confidence like understanding what is going on, and since we obtain per-feature weights, we can manually inspect the feature weights and features to see what exactly the learning algorithm learnt.

The next sections will go through steps 1 and 2. For step 3, please refer to the documentation of the tool.

Using t-SNE as visualisation

A common method to visualize high-dimensional data from pairwise distances is t-SNE -- a method that ingests a matrix of distances and attempts to create a low-dimensional (2d or 3d) embedding of these points that attempts to respect distances. The code comes with a small Python script that can be used to visualize

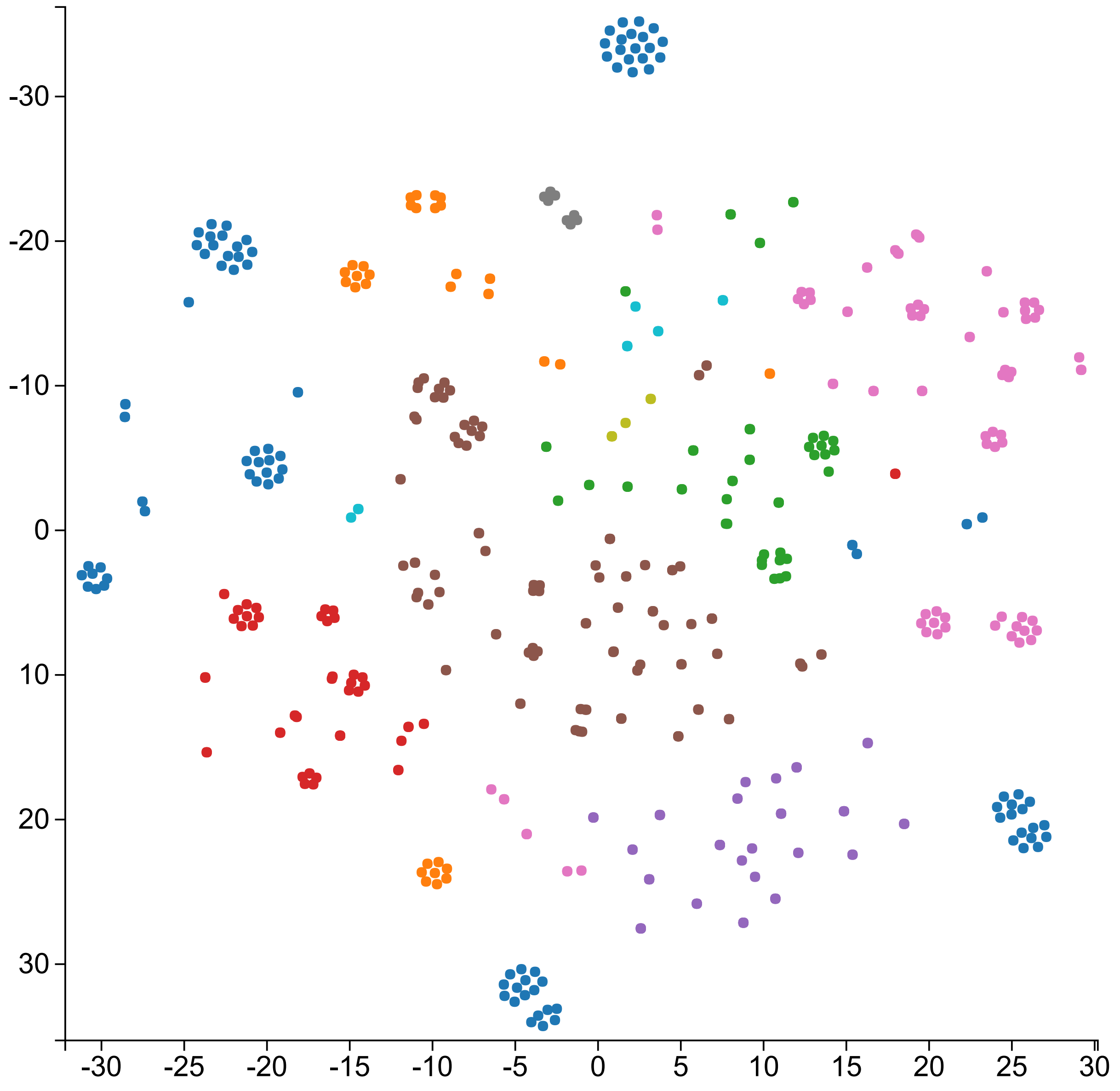
We will create two search indices: One populated with the “learnt feature weights”, and one populated with the “unit feature weight”:

```
# Create and populate an index with the ELF unrar samples with the
# learnt features.
./createfunctionindex --index=learnt_features.index; ./growfunctionindex --index=learnt_features.index --
size_to_grow=256; for i in $(ls ../testdata/ELF/unrar.5.5.3.builds/*); do echo $i; ./addfunctionstoindex -
-weights=420.snapshot --index=learnt_features.index --format=ELF --input=$i; done
# Add the PE files
for i in $(find ../testdata/PE/ -iname *.exe); do echo $i; ./addfunctionstoindex --weights=420.snapshot --
index=learnt_features.index --format=PE --input=$i; done

# Create and populate an index with unit weight features.
./createfunctionindex --index=unit_features.index; ./growfunctionindex --index=unit_features.index --
size_to_grow=256; for i in $(ls ../testdata/ELF/unrar.5.5.3.builds/*); do echo $i; ./addfunctionstoindex -
-index=unit_features.index --format=ELF --input=$i; done
# Add the PE files
for i in $(find ../testdata/PE/ -iname *.exe); do echo $i; ./addfunctionstoindex --
index=unit_features.index --format=PE --input=$i; done

# Dump the contents of the search index into a text file.
```

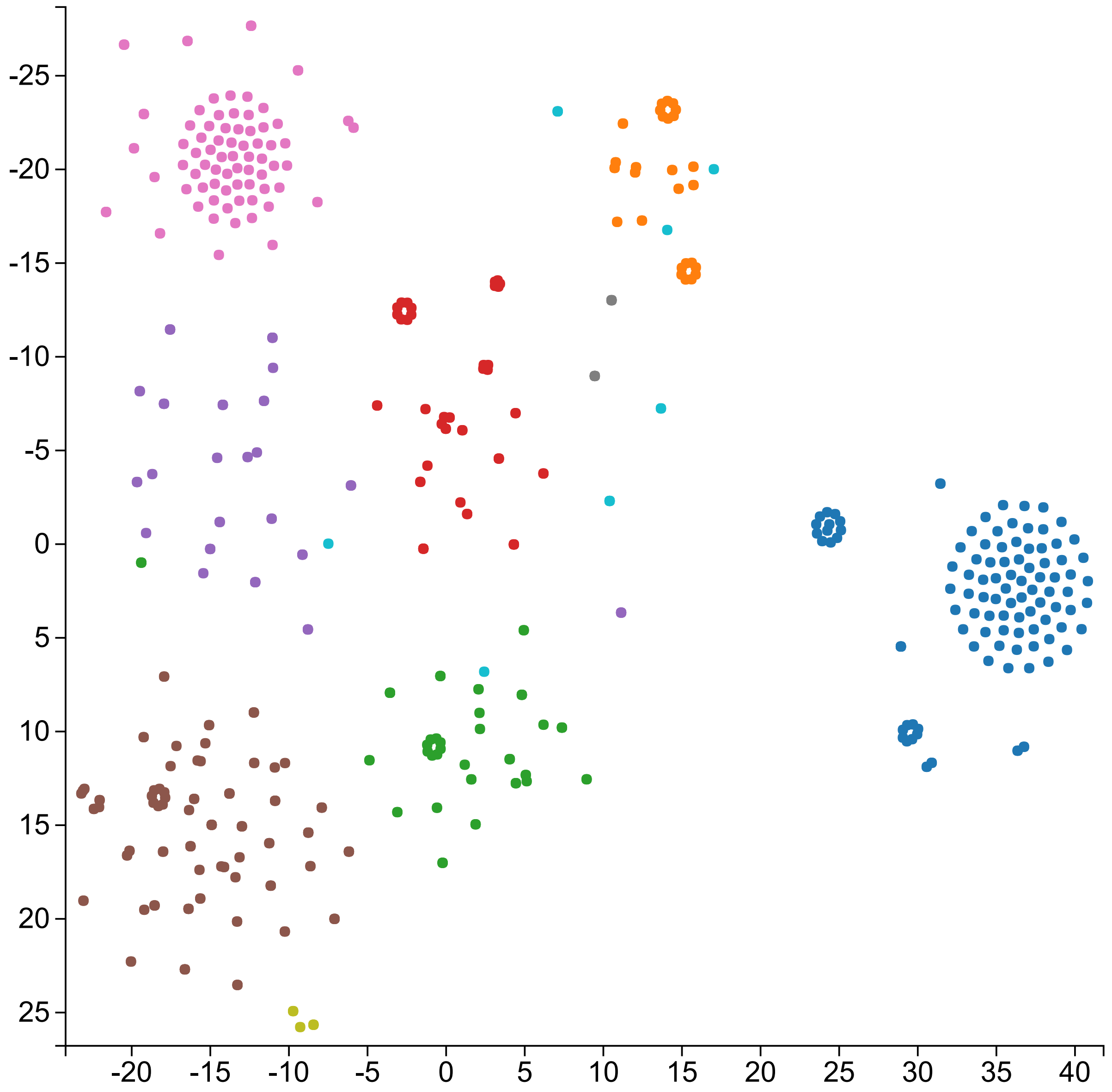
```
./dumpfunctionindex --index=learnt_features.index > learnt_index.txt
./dumpfunctionindex --index=unit_features.index > unit_index.txt

# Process the training data to create a single text file with symbols for
# all functions in the index.
cat /mnt/training_data/extracted_*.txt > ./symbols.txt

# Generate the visualisation
cd ../testdata
./plot_function_groups.py  ../bin/symbols.txt ../bin/learn_index.txt /tmp/learnt_features.html
./plot_function_groups.py  ../bin/symbols.txt ../bin/learnt_index.txt /tmp/learnt_features.html
```

We now have two HTML files that use d3.js to render the results:

Unit weights:



Learned weights:



Mouse-over on a point will display the function symbol and file-of-origin. It is visible to the naked eye that our training had the effect of moving groups of functions “more closely together”.

We can see here that the training does have some effect, but does not produce the same good effect for all functions: Some functions seem to benefit much more from the training than others, and it remains to be investigated why this is the case.

Examining TPR, FPR, IRR, and the ROC-curve

When evaluating information retrieval systems, various metrics are important: The true positive rate (how many of the results we were supposed to find did we find?), the false positive rate (how many of the results we were not supposed to find did we find?), the irrelevant result rate (what percentage of the results we returned were irrelevant? This is the complement to the precision), and the ROC curve (a plot of the TPR against the FPR).

This is helpful in both making informed choices about the right distance threshold, but also in order to quantify how much we are losing by performing approximate vs. precise search. It also helps us choose how many "hash buckets" we want to use for approximate searching.

There is a Python script in the git repository that can be used to generate the data for the ROC curve. The script requires a file with the symbols for all elements of the search index, a textual representation of the search index (obtained with `dumpsearchindex`, and access to the actual search index file.

```
# Create a search index to work with.
./createfunctionindex --index=/media/thomasdullien/roc/search.index
# Make it big enough to contain the data we are adding.
./growfunctionindex --index=/media/thomasdullien/roc/search.index --size_to_grow=1024
# Add all the functions from our training directories to it:
for filename in $(find ../testdata/ELF/ -iname *.ELF); do echo $filename; ./addfunctionstoindex --
format=ELF --input=$filename --index=/media/thomasdullien/roc/search.index; done
for filename in $(find ../testdata/PE/ -iname *.exe); do echo $filename; ./addfunctionstoindex --format=PE
--input=$filename --index=/media/thomasdullien/roc/search.index; done
# Now dump the search index into textual form for the Python script:
```

```
./dumpfunctionindex --index /media/thomasdullien/roc/search.index  >
/media/thomasdullien/roc/search.index.txt
# The file "symbols.txt" is just a concatenation of the symbols extracted during
# the run of the ./generate_training_data.py script.
cat /media/thomasdullien/training_data/extracted_symbols_*.txt > /media/thomasdullien/roc/symbols.txt
```

In order to obtain the data for the curve, we can use the following Python script:

```
testdata/evaluate_ROC_curve.py --symbols=/media/thomasdullien/roc/symbols.txt --
dbdump=/media/thomasdullien/roc/search.index.txt --index=/media/thomasdullien/roc/search.index
```

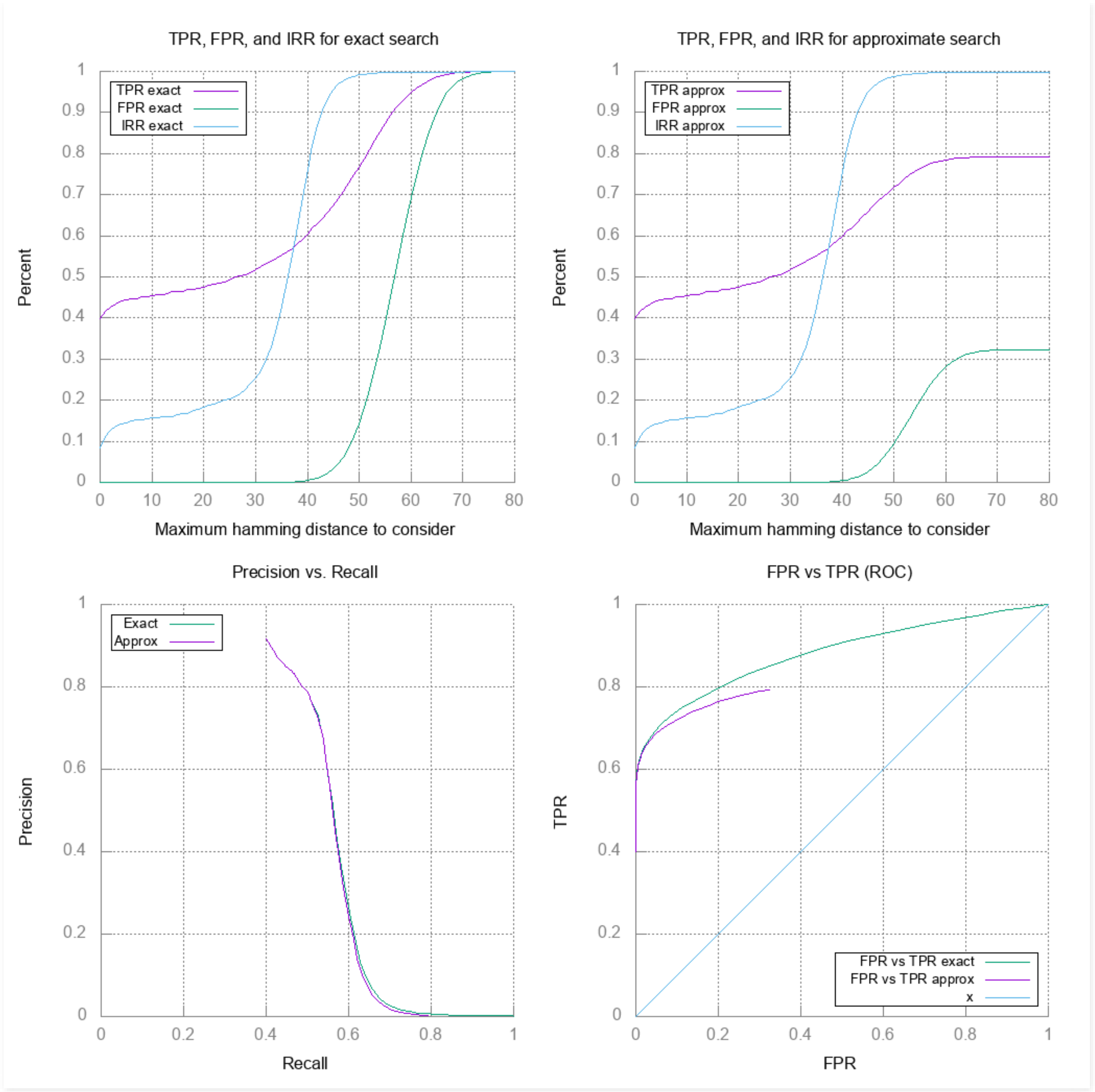
The output of the script is a 7-column output:

1. The maximum distance between two SimHash values to consider.
2. The true positive rate for exact (non-approximate-search-index) search.
3. The false positive rate for exact (non-approximate-search-index) search.
4. The true positive rate for search using the approximate search index.
5. The false positive rate for using the approximate search index.
6. The percentage of irrelevant results returned using exact search.
7. The percentage of irrelevant results returned using approximate search.

We can generate the curves for both the trained and untrained data, and then plot the results using gnuplot:

```
gnuplot -c ./testdata/plot_results_of_evaluate_ROC_curve.gnuplot ./untrained_roc.txt
gnuplot -c ./testdata/tpr_fpr_curve.gnuplot ./untrained_roc.txt ./trained_roc.txt
```

So let us examine this plots for the untrained results first:



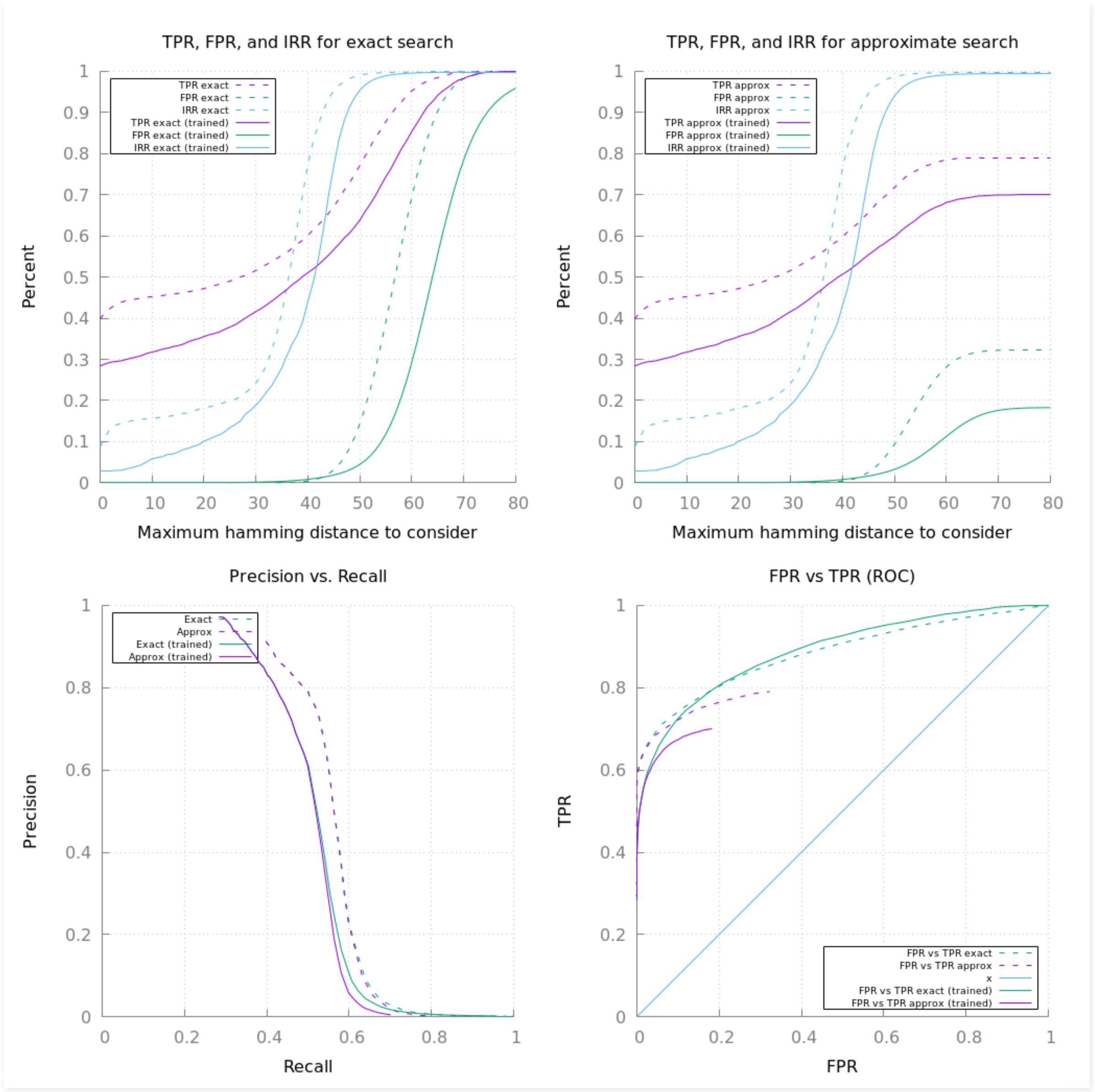
The first diagram shows that if we want a TPR of more than 50%, we will have to incur about 20% of the returned results being irrelevant to our search; the cut-off distance we should take for this is somewhere around 25 bits.

We also see that we will pay a heavy price for increasing the cut-off: At 35 bits, where our TPR hits 55%, half of our results are irrelevant. This is a weakness of the set-up at the moment, and we will see if it can be improved by learning weights.

The second diagram shows that we only pay in TPR for the approximate search for very high cut-offs - the TPR and FPR flatten off, which is a symptom of us missing more and more of the search space as we expand the number of bits we consider relevant.

The lower-left diagram shows how quickly our precision deteriorates as we try to improve the recall.

How are these curves affected by the training process?



So in the top-left curve, we can see that the rate of irrelevant results at 10 bits distance has dropped significantly: Down to approximately 5% from about 15%. Unfortunately, the true-positive-rate has also dropped - instead of about 45% of the results we want to get, we only achieve about 33%. So the training works in the sense that it improves the ratio of good results to irrelevant results significantly, but at the cost of lowering the overall rate of results that we find.

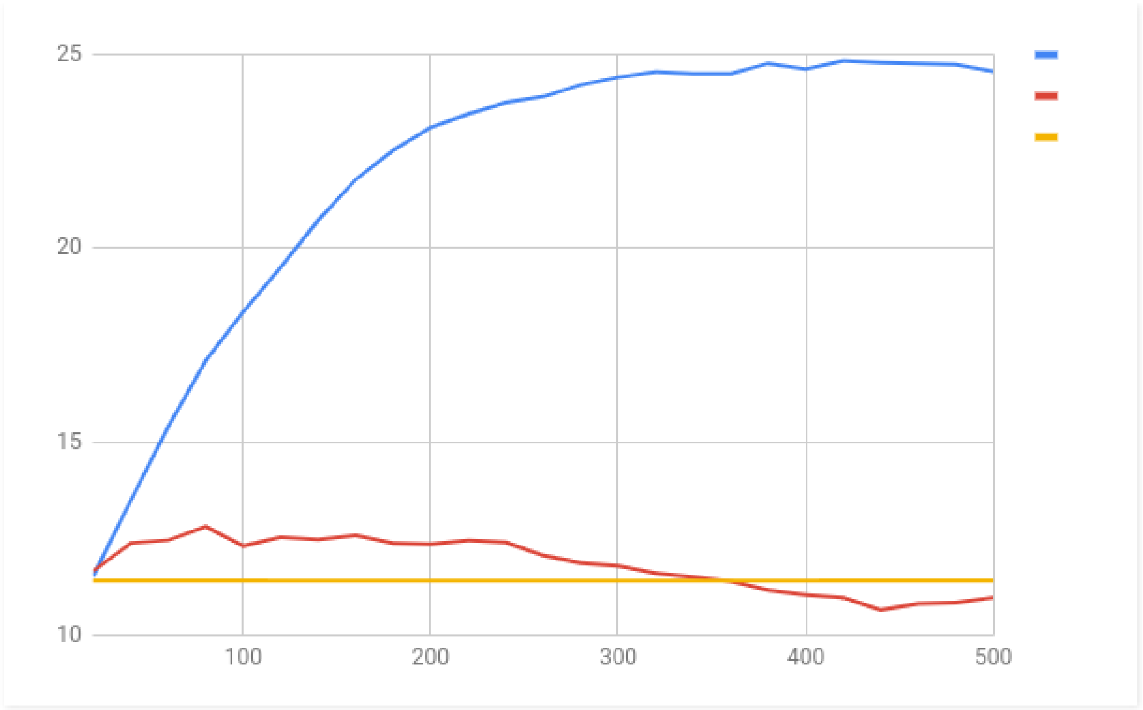
If we are willing to tolerate approximately 15% irrelevant results, we will get about 45% of the results we desire in the non-trained version. Sadly, in the trained version, for the same level of irrelevant results, we only get about 40% of the results we desire.

In summary: In the current form, the training is useful for lowering the irrelevant result rate below what is achievable without training - but for any acceptable rate of irrelevant results that can be achieved without training, the untrained version appears to achieve better results.

Does this generalize to out-of-sample functions?

In the section about splitting our training/validation data, we posed two questions - and the more interesting question is (2). Is there anything we are learning about the *compilers* ?

Plotting the difference-in-mean-distance that we plotted for question (1) also for question (2) yields the following image:



The red curve implies that there is a faint but non-zero signal - after about 80 training steps we have increased the mean-difference-in-means from 11.42 bits to 12.81 bits; overtraining appears to begin shortly thereafter.

It is unclear how much signal could be extracted using more powerful models; the fact that our super-simple linear model extracts *something* is encouraging.

Practical searching

Using FunctionSimSearch from any Python-enabled RE tool

The command-line tools mostly rely on DynInst for disassembly - but reverse engineers work with a bewildering plethora of different tools: IDA, Radare, Binary Ninja, Miasm etc. etc.

Given the development effort to build integration for all these tools, I decided that the simplest thing would be to provide Python bindings -- so any tool that can interact with a Python interpreter can use FunctionSimSearch via the same API. In order to get the tool installed into your Python interpreter, run:

```
python ./setup.py --install user
```

The easiest way to use the API from python is via JSON-based descriptions of flowgraphs:

```
 jsonstring = (... load the JSON ... )
 fg = functionsimsearch.FlowgraphWithInstructions()
 fg.from_json(jsonstring)
 hasher = functionsimsearch.SimHasher("../testdata/weights.txt")
 function_hash = hasher.calculate_hash(fg)
```

This yields a Python tuple with the hash of the function. The JSON graph format used as input looks as follows:

```
{
 "edges": [ { "destination": 1518838580, "source": 1518838565 }, (...) ],
 "name": "CFG",
 "nodes": [
   {
     "address": 1518838565,
     "instructions": [
       { "mnemonic": "xor", "operands": [ "EAX", "EAX" ] },
       { "mnemonic": "cmp", "operands": [ "[ECX + 4]", "EAX" ] },
       { "mnemonic": "jnle", "operands": [ "5a87a334" ] } ]
   }, (...)  ]
}
```

More details on how to use the Python API can be found in this example Python-based IDA Plugin. The plugin registers hotkeys to “save the current function in IDA into the hash database” and hotkeys to “search for similar functions to the current IDA function in the hash database". It also provides hotkeys to save the entire IDB into the Database, and to try to match every single function in a given disassembly against the search index.

For people that prefer using Binary Ninja, a plugin with similar functionality is available (thanks carstein@ :-).

Searching for unrar code in mpengine.dll

As a first use case, we will use IDA to populate a search index with symbols from unrar, and then search through mpengine.dll (also from Binary Ninja) for any functions that we may recognize.

We can populate a search index called "'/var/tmp/ida2/simhash.index"' from a set of existing disassemblies using the following command line:

```
# Create the file for the search index.
/home/thomasdullien/Desktop/sources/functionsimsearch/bin/createfunctionindex --
index=/var/tmp/ida2/simhash.index
# Populate using all 32-bit UnRAR.idb in a given directory.
for i in $(find /media/thomasdullien/unrar.4.2.4.builds.idbs/unrar/ -iname UnRAR.idb); do ./ida -
S"/usr/local/google/home/thomasdullien/sources/functionsimsearch/pybindings/ida_example.py export
/var/tmp/ida2/" $i; done
# Populate using all 64-bit UnRAR.i64 in a given directory.
for i in $(find /media/thomasdullien/unrar.4.2.4.builds.idbs/unrar/ -iname UnRAR.i64); do ./ida64 -
S"/usr/local/google/home/thomasdullien/sources/functionsimsearch/pybindings/ida_example.py export
/var/tmp/ida2/" $i; done
```

Once this is done, we can open mpengine.dll in IDA, go to File->Script File and load `ida_example.py`, then hit "Shift-M".

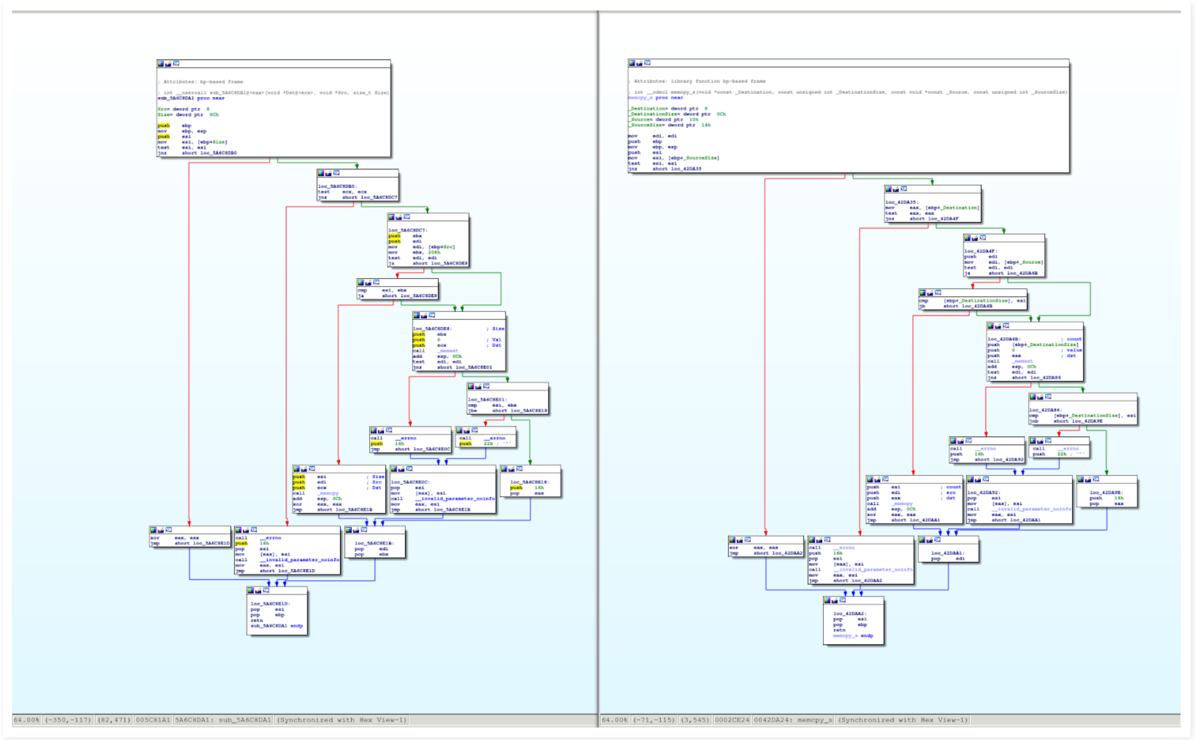
The IDA message window will get flooded with results like the text below:

```
(...)
```

```
6f4466b67afdbf73:5a6c8da1 f3f964313d8c559e-e6196c17e6c230b4 Result is 125.000000 - 72244a754ba4796d:42da24
x:\shared_win\library_sources\unrar\unrarsrc-4.2.4\unrar\build.VS2015\unrar32\Release\UnRAR.exe 'memcpy_s'
(1 in inf searches)
6f4466b67afdbf73:5a6c8da1 f3f964313d8c559e-e6196c17e6c230b4 Result is 125.000000 - ce2a2aa885d1a212:428234
x:\shared_win\library_sources\unrar\unrarsrc-4.2.4\unrar\build.VS2015\unrar32\MinSize\UnRAR.exe 'memcpy_s'
(1 in inf searches)
6f4466b67afdbf73:5a6c8da1 f3f964313d8c559e-e6196c17e6c230b4 Result is 125.000000 - 69c2ca5e6cb8a281:42da88
x:\shared_win\library_sources\unrar\unrarsrc-4.2.4\unrar\build.VS2015\unrar32\FullOpt\UnRAR.exe 'memcpy_s'
(1 in inf searches)
--------------------------------------
6f4466b67afdbf73:5a6f7dee e6af83501a8eedd8-6cdba61793e9a840 Result is 108.000000 - ce2a2aa885d1a212:419301
x:\shared_win\library_sources\unrar\unrarsrc-4.2.4\unrar\build.VS2015\unrar32\MinSize\UnRAR.exe '?
RestartModelRare@ModelPPM@@AAEXXZ' (1 in 12105083908.189119 searches)
6f4466b67afdbf73:5a6f7dee e6af83501a8eedd8-6cdba61793e9a840 Result is 107.000000 - 86bc6fc88e1453e8:41994b
x:\shared_win\library_sources\unrar\unrarsrc-4.2.4\unrar\build.VS2013\unrar32\MinSize\UnRAR.exe '?
RestartModelRare@ModelPPM@@AAEXXZ' (1 in 3026270977.047280 searches)
6f4466b67afdbf73:5a6f7dee e6af83501a8eedd8-6cdba61793e9a840 Result is 107.000000 - eb42e1fc45b05c7e:417030
x:\shared_win\library_sources\unrar\unrarsrc-4.2.4\unrar\build.VS2010\unrar32\MinSize\UnRAR.exe '?
RestartModelRare@ModelPPM@@AAEXXZ' (1 in 3026270977.047280 searches)
--------------------------------------
6f4466b67afdbf73:5a6fa46b f0b5a76c7eee2882-62d6c234a16c5b68 Result is 106.000000 - d4f4aa5dd49097be:414580
x:\shared_win\library_sources\unrar\unrarsrc-4.2.4\unrar\build.VS2010\unrar32\Release\UnRAR.exe '?
Execute@RarVM@@QAEXPAUVM_PreparedProgram@@@Z' (1 in 784038800.726675 searches)
6f4466b67afdbf73:5a6fa46b f0b5a76c7eee2882-62d6c234a16c5b68 Result is 106.000000 - 50bbba3fc643b153:4145c0
x:\shared_win\library_sources\unrar\unrarsrc-4.2.4\unrar\build.VS2010\unrar32\FullOpt\UnRAR.exe '?
Execute@RarVM@@QAEXPAUVM_PreparedProgram@@@Z' (1 in 784038800.726675 searches)
6f4466b67afdbf73:5a6fa46b f0b5a76c7eee2882-62d6c234a16c5b68 Result is 105.000000 - eb42e1fc45b05c7e:410717
x:\shared_win\library_sources\unrar\unrarsrc-4.2.4\unrar\build.VS2010\unrar32\MinSize\UnRAR.exe '?
Execute@RarVM@@QAEXPAUVM_PreparedProgram@@@Z' (1 in 209474446.235050 searches)
--------------------------------------
6f4466b67afdbf73:5a6fa59a c0ddbe744a832340-d7d062fe42fd5a60 Result is 106.000000 - eb42e1fc45b05c7e:40fd39
x:\shared_win\library_sources\unrar\unrarsrc-4.2.4\unrar\build.VS2010\unrar32\MinSize\UnRAR.exe '?
ExecuteCode@RarVM@@AAE_NPAUVM_PreparedCommand@@I@Z' (1 in 784038800.726675 searches)
```

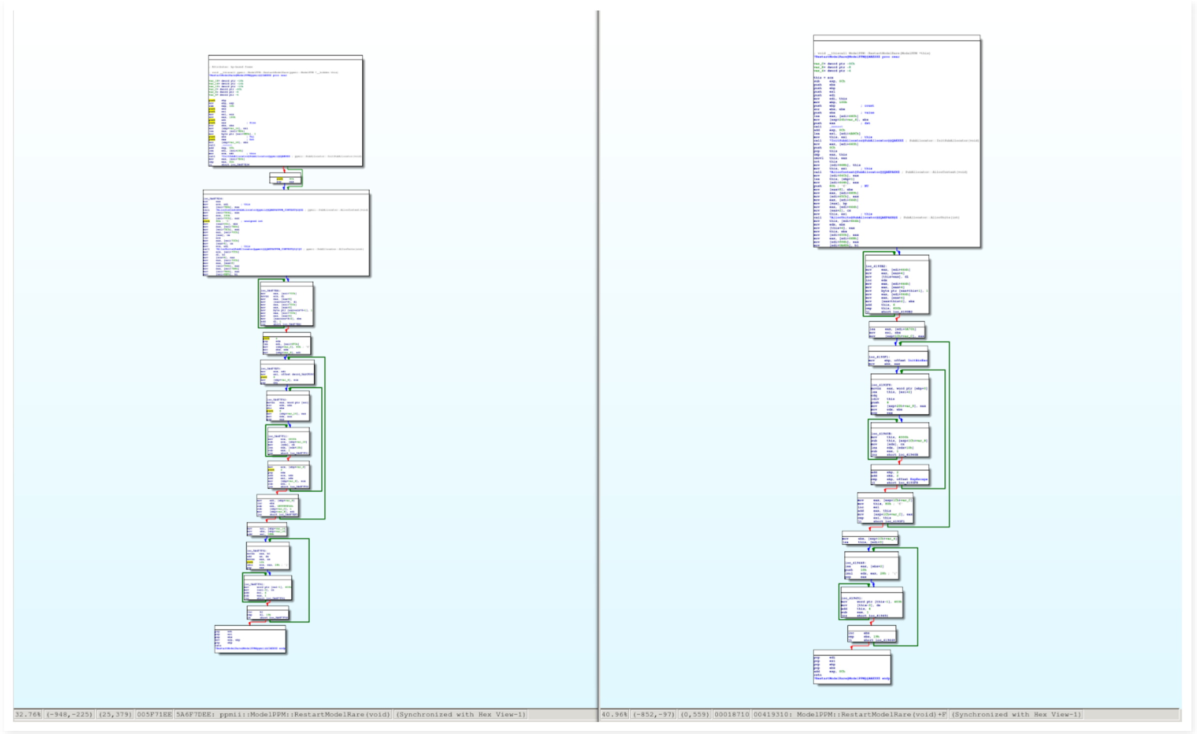
```
---------------------------------------
6f4466b67afdbf73:5a7ac980 c03968c6fad84480-2b8a2911b1ba1e40 Result is 105.000000 -
17052ba379b56077:140069170 x:\shared_win\library_sources\unrar\unrarsrc-
4.2.4\unrar\build.VS2015\unrar64\Debug\UnRAR.exe 'strrchr' (1 in 209474446.235050 searches)
6f4466b67afdbf73:5a7ac980 c03968c6fad84480-2b8a2911b1ba1e40 Result is 105.000000 -
4e07df225c1cf59c:140064590 x:\shared_win\library_sources\unrar\unrarsrc-
4.2.4\unrar\build.VS2013\unrar64\Debug\UnRAR.exe 'strrchr' (1 in 209474446.235050 searches)
6f4466b67afdbf73:5a7ac980 c03968c6fad84480-2b8a2911b1ba1e40 Result is 105.000000 -
a754eed77d0059ed:1400638f0 x:\shared_win\library_sources\unrar\unrarsrc-
4.2.4\unrar\build.VS2012\unrar64\Debug\UnRAR.exe 'strrchr' (1 in 209474446.235050 searches)
---------------------------------------
```

Let us examine some of these results a bit more in-depth. The first result claims to have found a version of `memcpy_s`, with a 125 out of 128 bits matching. This implies a very close match. The corresponding disassemblies are:



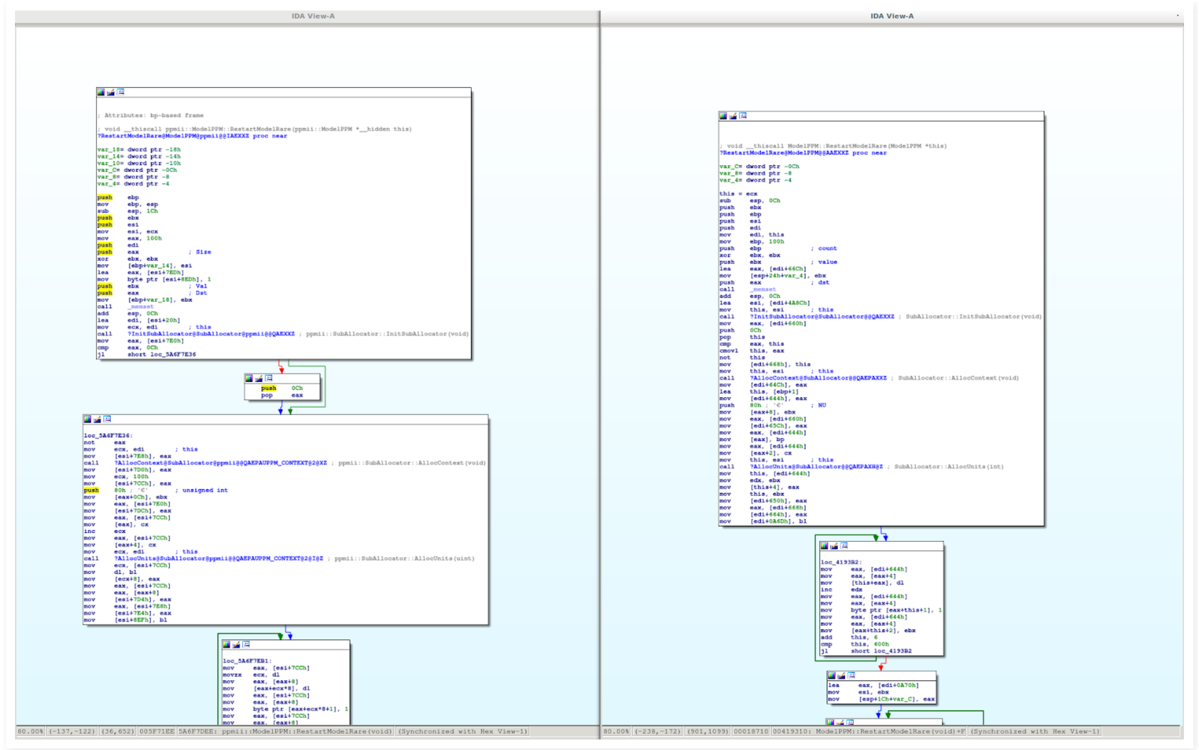
Aside from a few minor changes on the instruction-level, the two functions are clearly the same - even the CFG structure stayed identical.

The next result claims to have found a variant `ppmii::ModelPPM::RestartModelRare` with 108 of the 128 bits matching.



```
32.76% (-948,-225) (25,379) 005F71EE 5A6F7DEE: ppmii::ModelPPM::RestartModelRare(void) (Synchronized with Hex View-1)
```

```
40.96% (-852,-97) (0,559) 00018710 00419310: ModelPPM::RestartModelRare(void)+F (Synchronized with Hex View-1)
```



```
; Attributes: bp-based frame

; void __thiscall ppmii::ModelPPM::RestartModelRare(ppmii::ModelPPM *__hidden this)
?RestartModelRare@ModelPPM@ppmii@@IAEXXZ proc near

var_18= dword ptr -18h
var_14= dword ptr -14h
var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 1Ch
push    ebx
push    esi
mov     esi, ecx
mov     eax, 100h
push    edi
push    eax             ; Size
xor     ebx, ebx
mov     [ebp+var_14], esi
lea     eax, [esi+7EDh]
mov     byte ptr [esi+8EDh], 1
push    ebx             ; Val
push    eax             ; Dst
mov     [ebp+var_18], ebx
call    _memset
add     esp, 0Ch
lea     edi, [esi+20h]
mov     ecx, edi        ; this
call    ?InitSubAllocator@SubAllocator@ppmii@@QAEXXZ ; ppmii::SubAllocator::InitSubAllocator(void)
mov     eax, [esi+7E0h]
cmp     eax, 0Ch
jl      short loc_5A6F7E36

        push    0Ch
        pop     eax

loc_5A6F7E36:
not     eax
mov     ecx, edi        ; this
mov     [esi+7E8h], eax
call    ?AllocContext@SubAllocator@ppmii@@QAEPAUPPM_CONTEXT@2@XZ ; ppmii::SubAllocator::AllocContext(void)
mov     [esi+7D0h], eax
mov     ecx, 100h
mov     [esi+7CCh], eax
push    80h ; '€'       ; unsigned int
mov     [eax+0Ch], ebx
mov     eax, [esi+7E0h]
mov     [esi+7DCh], eax
mov     eax, [esi+7CCh]
mov     [eax], cx
inc     ecx
mov     eax, [esi+7CCh]
mov     [eax+4], cx
mov     ecx, edi        ; this
call    ?AllocUnits@SubAllocator@ppmii@@QAEPAUPPM_CONTEXT@2@I@Z ; ppmii::SubAllocator::AllocUnits(uint)
mov     ecx, [esi+7CCh]
mov     dl, bl
mov     [ecx+8], eax
mov     eax, [esi+7CCh]
mov     eax, [eax+8]
mov     [esi+7D4h], eax
mov     eax, [esi+7E8h]
mov     [esi+7E4h], eax
mov     [esi+8EFh], bl

loc_5A6F7EB1:
mov     eax, [esi+7CCh]
movzx   ecx, dl
mov     eax, [eax+8]
mov     [eax+ecx*8], dl
mov     eax, [esi+7CCh]
mov     eax, [eax+8]
mov     byte ptr [eax+ecx*8+1], 1
mov     eax, [esi+7CCh]
mov     eax, [eax+8]
```

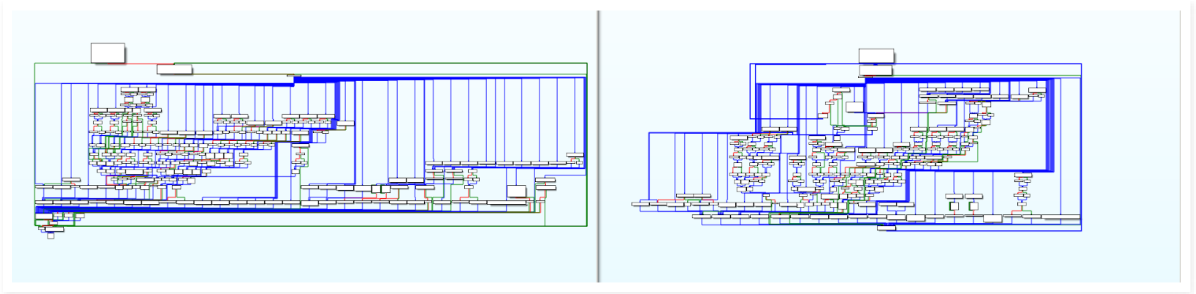
```
80.00% (-137,-122) (36,652) 005F71EE 5A6F7DEE: ppmii::ModelPPM::RestartModelRare(void) (Synchronized with Hex View-1)
```

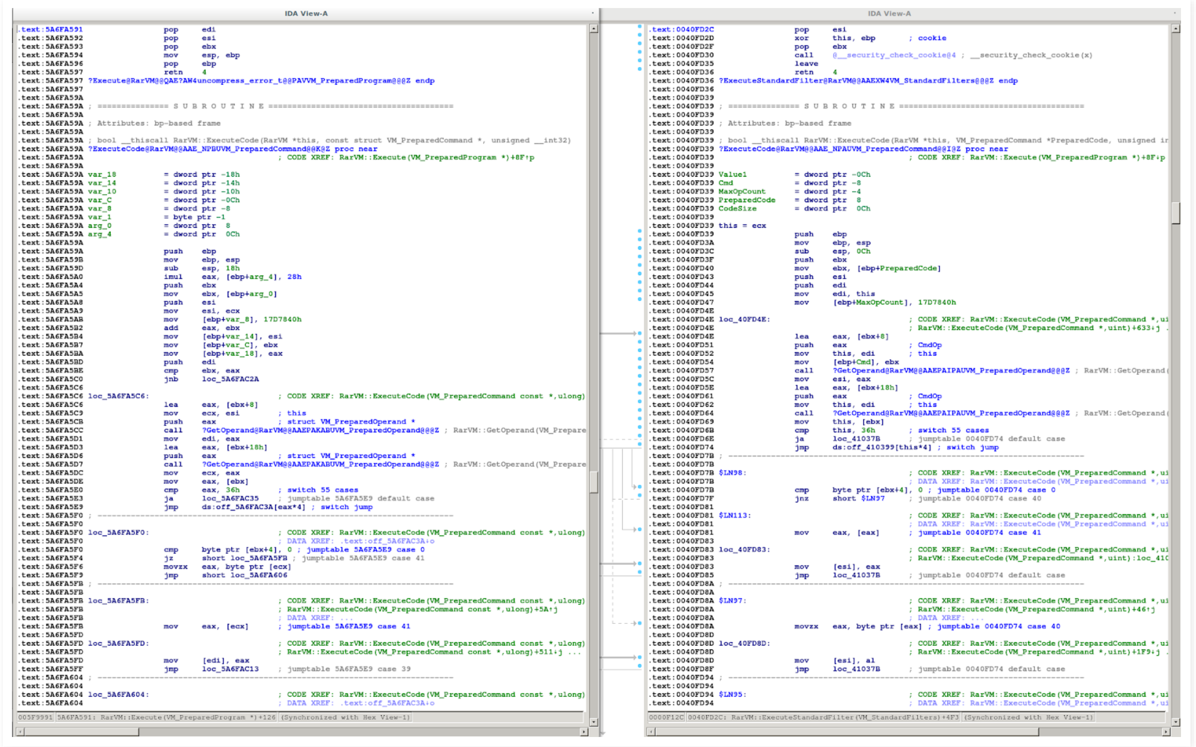
```
; void __thiscall ModelPPM::RestartModelRare(ModelPPM *this)
?RestartModelRare@ModelPPM@@AAEXXZ proc near

var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4

this = ecx
sub     esp, 0Ch
push    ebx
push    ebp
push    esi
push    edi
mov     edi, this
mov     ebp, 100h
push    ebp             ; count
xor     ebx, ebx
push    ebx             ; value
lea     eax, [edi+66Ch]
mov     [esp+24h+var_4], ebx
push    eax             ; dst
call    _memset
add     esp, 0Ch
lea     esi, [edi+4A8Ch]
mov     this, esi       ; this
call    ?InitSubAllocator@SubAllocator@@QAEXXZ ; SubAllocator::InitSubAllocator(void)
mov     eax, [edi+660h]
push    0Ch
pop     this
cmp     eax, this
cmovl   this, eax
not     this
mov     [edi+668h], this
mov     this, esi       ; this
call    ?AllocContext@SubAllocator@@QAEPAXXZ ; SubAllocator::AllocContext(void)
mov     [edi+64Ch], eax
lea     this, [ebp+1]
mov     [edi+644h], eax
push    80h ; '€'       ; NU
mov     [eax+8], ebx
mov     eax, [edi+660h]
mov     [edi+65Ch], eax
mov     eax, [edi+644h]
mov     [eax], bp
mov     eax, [edi+644h]
mov     [eax+2], cx
mov     this, esi       ; this
call    ?AllocUnits@SubAllocator@@QAEPAXH@Z ; SubAllocator::AllocUnits(int)
mov     this, [edi+644h]
mov     edx, ebx
mov     [this+4], eax
mov     this, ebx
mov     [edi+650h], eax
mov     eax, [edi+668h]
mov     [edi+664h], eax
mov     [edi+0A6Dh], bl

loc_4193B2:
mov     eax, [edi+644h]
mov     eax, [eax+4]
mov     [this+eax], dl
inc     edx
mov     eax, [edi+644h]
mov     eax, [eax+4]
mov     byte ptr [eax+this+1], 1
mov     eax, [edi+644h]
mov     eax, [eax+4]
mov     [eax+this+2], ebx
add     this, 6
cmp     this, 600h
jl      short loc_4193B2

lea     eax, [edi+0A70h]
mov     esi, ebx
mov     [esp+1Ch+var_C], eax
```

```
80.00% (-238,-172) (901,1099) 00018710 00419310: ModelPPM::RestartModelRare(void)+F (Synchronized with Hex View-1)
```

The disassembly (and all structure offsets in the code) seems to have changed quite a bit, but the overall CFG structure is mostly intact: The first large basic block was broken up by the compiler, so the graphs are not identical, but they are definitely still highly similar.

The next example is a much larger function - the result claims to have identified `RarVM::ExecuteCode` with 106 of 128 bits of the hash matching. What does the graph (and the disassembly) look like?



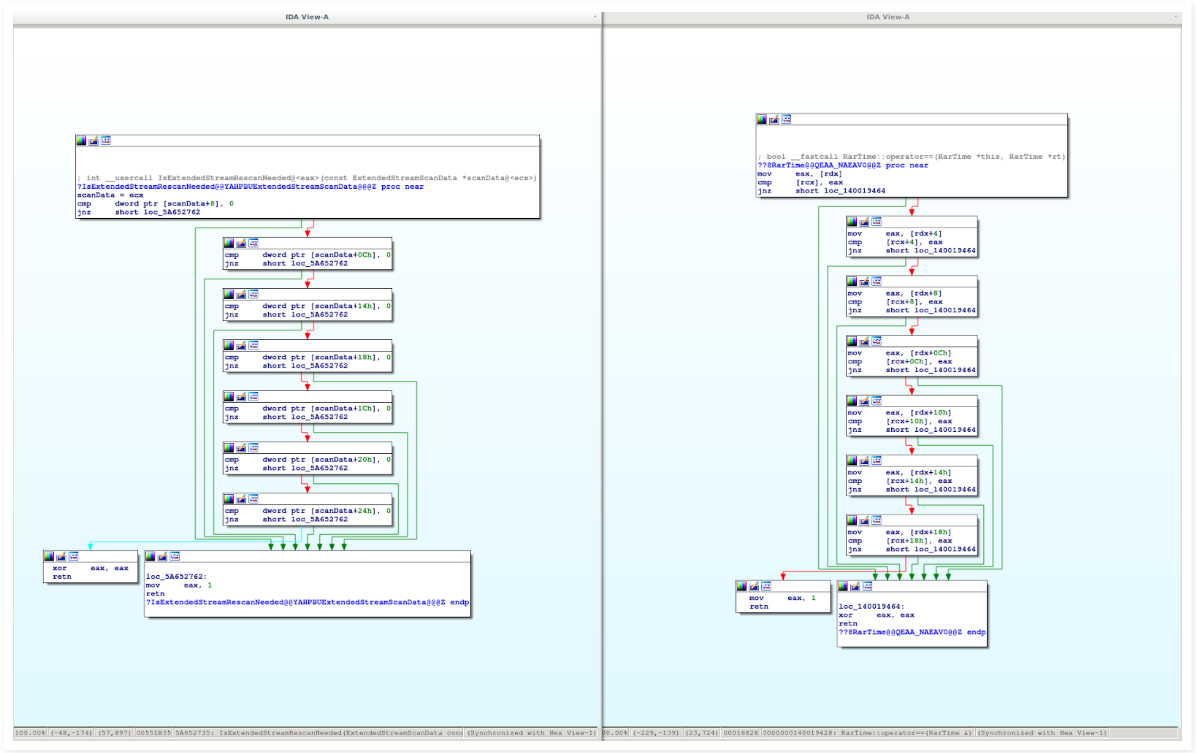


```
IDA View-A
.text:5A6FA591                 pop     edi
.text:5A6FA592                 pop     esi
.text:5A6FA593                 pop     ebx
.text:5A6FA594                 mov     esp, ebp
.text:5A6FA596                 pop     ebp
.text:5A6FA597                 retn    4
.text:5A6FA597 ?Execute@RarVM@@QAE?AW4uncompress_error_t@@PAVVM_PreparedProgram@@@Z endp
.text:5A6FA597
.text:5A6FA59A
.text:5A6FA59A ; =============== S U B R O U T I N E =======================================
.text:5A6FA59A
.text:5A6FA59A ; Attributes: bp-based frame
.text:5A6FA59A
.text:5A6FA59A ; bool __thiscall RarVM::ExecuteCode(RarVM *this, const struct VM_PreparedCommand *, unsigned __int32)
.text:5A6FA59A ?ExecuteCode@RarVM@@AAE_NPBUVM_PreparedCommand@@K@Z proc near
.text:5A6FA59A                                         ; CODE XREF: RarVM::Execute(VM_PreparedProgram *)+8F↑p
.text:5A6FA59A
.text:5A6FA59A var_18          = dword ptr -18h
.text:5A6FA59A var_14          = dword ptr -14h
.text:5A6FA59A var_10          = dword ptr -10h
.text:5A6FA59A var_C           = dword ptr -0Ch
.text:5A6FA59A var_8           = dword ptr -8
.text:5A6FA59A var_1           = byte ptr -1
.text:5A6FA59A arg_0           = dword ptr  8
.text:5A6FA59A arg_4           = dword ptr  0Ch
.text:5A6FA59A
.text:5A6FA59A                 push    ebp
.text:5A6FA59B                 mov     ebp, esp
.text:5A6FA59D                 sub     esp, 18h
.text:5A6FA5A0                 imul    eax, [ebp+arg_4], 28h
.text:5A6FA5A4                 push    ebx
.text:5A6FA5A5                 mov     ebx, [ebp+arg_0]
.text:5A6FA5A8                 push    esi
.text:5A6FA5A9                 mov     esi, ecx
.text:5A6FA5AB                 mov     [ebp+var_8], 17D7840h
.text:5A6FA5B2                 add     eax, ebx
.text:5A6FA5B4                 mov     [ebp+var_14], esi
.text:5A6FA5B7                 mov     [ebp+var_C], ebx
.text:5A6FA5BA                 mov     [ebp+var_18], eax
.text:5A6FA5BD                 push    edi
.text:5A6FA5BE                 cmp     ebx, eax
.text:5A6FA5C0                 jnb     loc_5A6FAC2A
.text:5A6FA5C6
.text:5A6FA5C6 loc_5A6FA5C6:                           ; CODE XREF: RarVM::ExecuteCode(VM_PreparedCommand const *,ulong)
.text:5A6FA5C6                 lea     eax, [ebx+8]
.text:5A6FA5C9                 mov     ecx, esi        ; this
.text:5A6FA5CB                 push    eax             ; struct VM_PreparedOperand *
.text:5A6FA5CC                 call    ?GetOperand@RarVM@@AAEPAKABUVM_PreparedOperand@@@Z ; RarVM::GetOperand(VM_Prepare
.text:5A6FA5D1                 mov     edi, eax
.text:5A6FA5D3                 lea     eax, [ebx+18h]
.text:5A6FA5D6                 push    eax             ; struct VM_PreparedOperand *
.text:5A6FA5D7                 call    ?GetOperand@RarVM@@AAEPAKABUVM_PreparedOperand@@@Z ; RarVM::GetOperand(VM_Prepare
.text:5A6FA5DC                 mov     ecx, eax
.text:5A6FA5DE                 mov     eax, [ebx]
.text:5A6FA5E0                 cmp     eax, 36h        ; switch 55 cases
.text:5A6FA5E3                 ja      loc_5A6FAC35    ; jumptable 5A6FA5E9 default case
.text:5A6FA5E9                 jmp     ds:off_5A6FAC3A[eax*4] ; switch jump
.text:5A6FA5F0 ; ---------------------------------------------------------------------------
.text:5A6FA5F0
.text:5A6FA5F0 loc_5A6FA5F0:                           ; CODE XREF: RarVM::ExecuteCode(VM_PreparedCommand const *,ulong)
.text:5A6FA5F0                                         ; DATA XREF: .text:off_5A6FAC3A↓o
.text:5A6FA5F0                 cmp     byte ptr [ebx+4], 0 ; jumptable 5A6FA5E9 case 0
.text:5A6FA5F4                 jz      short loc_5A6FA5FB ; jumptable 5A6FA5E9 case 41
.text:5A6FA5F6                 movzx   eax, byte ptr [ecx]
.text:5A6FA5F9                 jmp     short loc_5A6FA606
.text:5A6FA5FB ; ---------------------------------------------------------------------------
.text:5A6FA5FB
.text:5A6FA5FB loc_5A6FA5FB:                           ; CODE XREF: RarVM::ExecuteCode(VM_PreparedCommand const *,ulong)
.text:5A6FA5FB                                         ; RarVM::ExecuteCode(VM_PreparedCommand const *,ulong)+5A↑j
.text:5A6FA5FB                                         ; DATA XREF: ...
.text:5A6FA5FB                 mov     eax, [ecx]      ; jumptable 5A6FA5E9 case 41
.text:5A6FA5FD
.text:5A6FA5FD loc_5A6FA5FD:                           ; CODE XREF: RarVM::ExecuteCode(VM_PreparedCommand const *,ulong)
.text:5A6FA5FD                                         ; RarVM::ExecuteCode(VM_PreparedCommand const *,ulong)+511↓j ...
.text:5A6FA5FD                 mov     [edi], eax
.text:5A6FA5FF                 jmp     loc_5A6FAC13    ; jumptable 5A6FA5E9 case 39
.text:5A6FA604 ; ---------------------------------------------------------------------------
.text:5A6FA604
.text:5A6FA604 loc_5A6FA604:                           ; CODE XREF: RarVM::ExecuteCode(VM_PreparedCommand const *,ulong)
.text:5A6FA604                                         ; DATA XREF: .text:off_5A6FAC3A↓o
005F9991 5A6FA591: RarVM::Execute(VM_PreparedProgram *)+126 (Synchronized with Hex View-1)
```

```
IDA View-A
.text:0040FD2C                 pop     esi
.text:0040FD2D                 xor     this, ebp       ; cookie
.text:0040FD2F                 pop     ebx
.text:0040FD30                 call    @__security_check_cookie@4 ; __security_check_cookie(x)
.text:0040FD35                 leave
.text:0040FD36                 retn    4
.text:0040FD36 ?ExecuteStandardFilter@RarVM@@AAEXW4VM_StandardFilters@@@Z endp
.text:0040FD36
.text:0040FD39
.text:0040FD39 ; =============== S U B R O U T I N E =======================================
.text:0040FD39
.text:0040FD39 ; Attributes: bp-based frame
.text:0040FD39
.text:0040FD39 ; bool __thiscall RarVM::ExecuteCode(RarVM *this, VM_PreparedCommand *PreparedCode, unsigned ir
.text:0040FD39 ?ExecuteCode@RarVM@@AAE_NPAUVM_PreparedCommand@@I@Z proc near
.text:0040FD39                                         ; CODE XREF: RarVM::Execute(VM_PreparedProgram *)+8F↓p
.text:0040FD39
.text:0040FD39 Value1          = dword ptr -0Ch
.text:0040FD39 Cmd             = dword ptr -8
.text:0040FD39 MaxOpCount      = dword ptr -4
.text:0040FD39 PreparedCode    = dword ptr  8
.text:0040FD39 CodeSize        = dword ptr  0Ch
.text:0040FD39
.text:0040FD39 this = ecx
.text:0040FD39                 push    ebp
.text:0040FD3A                 mov     ebp, esp
.text:0040FD3C                 sub     esp, 0Ch
.text:0040FD3F                 push    ebx
.text:0040FD40                 mov     ebx, [ebp+PreparedCode]
.text:0040FD43                 push    esi
.text:0040FD44                 push    edi
.text:0040FD45                 mov     edi, this
.text:0040FD47                 mov     [ebp+MaxOpCount], 17D7840h
.text:0040FD4E
.text:0040FD4E loc_40FD4E:                             ; CODE XREF: RarVM::ExecuteCode(VM_PreparedCommand *,ui
.text:0040FD4E                                         ; RarVM::ExecuteCode(VM_PreparedCommand *,uint)+633↓j .
.text:0040FD4E                 lea     eax, [ebx+8]
.text:0040FD51                 push    eax             ; CmdOp
.text:0040FD52                 mov     this, edi       ; this
.text:0040FD54                 mov     [ebp+Cmd], ebx
.text:0040FD57                 call    ?GetOperand@RarVM@@AAEPAIPAUVM_PreparedOperand@@@Z ; RarVM::GetOperand(
.text:0040FD5C                 mov     esi, eax
.text:0040FD5E                 lea     eax, [ebx+18h]
.text:0040FD61                 push    eax             ; CmdOp
.text:0040FD62                 mov     this, edi       ; this
.text:0040FD64                 call    ?GetOperand@RarVM@@AAEPAIPAUVM_PreparedOperand@@@Z ; RarVM::GetOperand(
.text:0040FD69                 mov     this, [ebx]
.text:0040FD6B                 cmp     this, 36h       ; switch 55 cases
.text:0040FD6E                 ja      loc_41037B      ; jumptable 0040FD74 default case
.text:0040FD74                 jmp     ds:off_410399[this*4] ; switch jump
.text:0040FD7B ; ---------------------------------------------------------------------------
.text:0040FD7B
.text:0040FD7B $LN98:                                  ; CODE XREF: RarVM::ExecuteCode(VM_PreparedCommand *,ui
.text:0040FD7B                                         ; DATA XREF: RarVM::ExecuteCode(VM_PreparedCommand *,ui
.text:0040FD7B                 cmp     byte ptr [ebx+4], 0 ; jumptable 0040FD74 case 0
.text:0040FD7F                 jnz     short $LN97     ; jumptable 0040FD74 case 40
.text:0040FD81
.text:0040FD81 $LN113:                                 ; CODE XREF: RarVM::ExecuteCode(VM_PreparedCommand *,ui
.text:0040FD81                                         ; DATA XREF: RarVM::ExecuteCode(VM_PreparedCommand *,ui
.text:0040FD81                 mov     eax, [eax]      ; jumptable 0040FD74 case 41
.text:0040FD83
.text:0040FD83 loc_40FD83:                             ; CODE XREF: RarVM::ExecuteCode(VM_PreparedCommand *,ui
.text:0040FD83                                         ; RarVM::ExecuteCode(VM_PreparedCommand *,uint):loc_41C
.text:0040FD83                 mov     [esi], eax
.text:0040FD85                 jmp     loc_41037B      ; jumptable 0040FD74 default case
.text:0040FD8A ; ---------------------------------------------------------------------------
.text:0040FD8A
.text:0040FD8A $LN97:                                  ; CODE XREF: RarVM::ExecuteCode(VM_PreparedCommand *,ui
.text:0040FD8A                                         ; RarVM::ExecuteCode(VM_PreparedCommand *,uint)+46↑j
.text:0040FD8A                                         ; DATA XREF: ...
.text:0040FD8A                 movzx   eax, byte ptr [eax] ; jumptable 0040FD74 case 40
.text:0040FD8D
.text:0040FD8D loc_40FD8D:                             ; CODE XREF: RarVM::ExecuteCode(VM_PreparedCommand *,ui
.text:0040FD8D                                         ; RarVM::ExecuteCode(VM_PreparedCommand *,uint)+1F9↓j .
.text:0040FD8D                 mov     [esi], al
.text:0040FD8F                 jmp     loc_41037B      ; jumptable 0040FD74 default case
.text:0040FD94 ; ---------------------------------------------------------------------------
.text:0040FD94
.text:0040FD94 $LN95:                                  ; CODE XREF: RarVM::ExecuteCode(VM_PreparedCommand *,ui
.text:0040FD94                                         ; DATA XREF: RarVM::ExecuteCode(VM_PreparedCommand *,ui
0000F12C 0040FD2C: RarVM::ExecuteStandardFilter(VM_StandardFilters)+4F3 (Synchronized with Hex View-1)
```

In this example, the graph has changed substantially, but a few subgraphs seem to have remained stable. Furthermore, the code contains magic constants (such as 0x17D7840, or the 0x36) that will have factored into the overall hash. This is a nontrivial find, so ... yay!

This blog post would not be complete without showing an example of a false positive: Our search also brings up a match for ``RarTime::operator==``. The match is very high-confidence -- 125 out of 128 bits match, but it turns out that - while the code is very similar - the functions do not actually have any relationship on the source-code level.



```
; int __usercall IsExtendedStreamRescanNeeded@<eax>(const ExtendedStreamScanData *scanData@<ecx>)
?IsExtendedStreamRescanNeeded@@YAHPBUExtendedStreamScanData@@@Z proc near
scanData = ecx
cmp     dword ptr [scanData+8], 0
jnz     short loc_5A652762

cmp     dword ptr [scanData+0Ch], 0
jnz     short loc_5A652762

cmp     dword ptr [scanData+14h], 0
jnz     short loc_5A652762

cmp     dword ptr [scanData+18h], 0
jnz     short loc_5A652762

cmp     dword ptr [scanData+1Ch], 0
jnz     short loc_5A652762

cmp     dword ptr [scanData+20h], 0
jnz     short loc_5A652762

cmp     dword ptr [scanData+24h], 0
jnz     short loc_5A652762

  xor     eax, eax
  retn

loc_5A652762:
mov     eax, 1
retn
?IsExtendedStreamRescanNeeded@@YAHPBUExtendedStreamScanData@@@Z endp
```

```
100.00% (-48,-174) (57,897) 00551B35 5A652735: IsExtendedStreamRescanNeeded(ExtendedStreamScanData cons (Synchronized with Hex View-1)
```

```
; bool __fastcall RarTime::operator==(RarTime *this, RarTime *rt)
??8RarTime@@QEAA_NAEAV0@@Z proc near
mov     eax, [rdx]
cmp     [rcx], eax
jnz     short loc_140019464

mov     eax, [rdx+4]
cmp     [rcx+4], eax
jnz     short loc_140019464

mov     eax, [rdx+8]
cmp     [rcx+8], eax
jnz     short loc_140019464

mov     eax, [rdx+0Ch]
cmp     [rcx+0Ch], eax
jnz     short loc_140019464

mov     eax, [rdx+10h]
cmp     [rcx+10h], eax
jnz     short loc_140019464

mov     eax, [rdx+14h]
cmp     [rcx+14h], eax
jnz     short loc_140019464

mov     eax, [rdx+18h]
cmp     [rcx+18h], eax
jnz     short loc_140019464

  mov     eax, 1
  retn

loc_140019464:
xor     eax, eax
retn
??8RarTime@@QEAA_NAEAV0@@Z endp
```

```
00.00% (-229,-139) (23,724) 00018828 0000000140019428: RarTime::operator==(RarTime &) (Synchronized with Hex View-1)
```

Both functions check a number of data members of a structure, and return either true or false if all the values are as expected. Such a construct can arise easily - especially in operator==-style constructs.

Searching for libtiff code in Adobe Reader

It is well-documented that Adobe Reader has been bitten by using outdated versions of libtiff in the past. This means that running a search through AcroForm.dll should provide us with a number of good hits from libtiff, and failure to achieve this should raise some eyebrows.

We populate a database with a variety of libtiff builds, run the plugin as we did previously, and examine the results. In comparison to the mpengine case, we get dozens of high-likelihood-results -- the codebase inside AcroForm.dll has not diverged from upstream quite as heavily as the Unrar fork inside mpengine.

Searching for libtiff code through all my Windows DLLs

Searching for code that we already know is present is not terribly interesting. How about searching for traces of libtiff across an entire harddisk with Windows 10 installed?

In order to do this from the command line (e.g. without any real third-party disassembler), we need a few things:

1. A directory in which we have compiled `libtiff` with a variety of different versions of Visual Studio and a variety of different compiler settings.
2. Debug information from the PDB files in a format we can easily parse. The current tooling expects a `.debugdump` file in the same directory as the PDB file, obtained by using Microsofts `DIA2Dump` tool and redirecting the output to a text file.

Let's create a new search index and populate it:

```
# Create the file for the search index.
/home/thomasdullien/Desktop/sources/functionsimsearch/bin/createfunctionindex --
index=/var/tmp/work/simhash.index
# Populate it.
for i in $(find /media/thomasdullien/storage/libtiff/PE/ -name tiff.dll); do
./addfunctionstoindex --input=$i --format=PE --index=/var/tmp/work/simhash.index; done
```

We also want some metadata so we know the symbols of the files in the search index.

We can generate a metadata file to be used with a search index by running the same script that generates training data:

```
~/Desktop/sources/functionsimsearch/testdata/generate_training_data.py --
work_directory=/var/tmp/work/ --
executable_directory=/media/thomasdullien/storage/libtiff/ --generate_fingerprints=True -
-generate_json_data=False
```

```
cat /var/tmp/work/extracted_symbols* > /var/tmp/work/simhash.index.meta
```

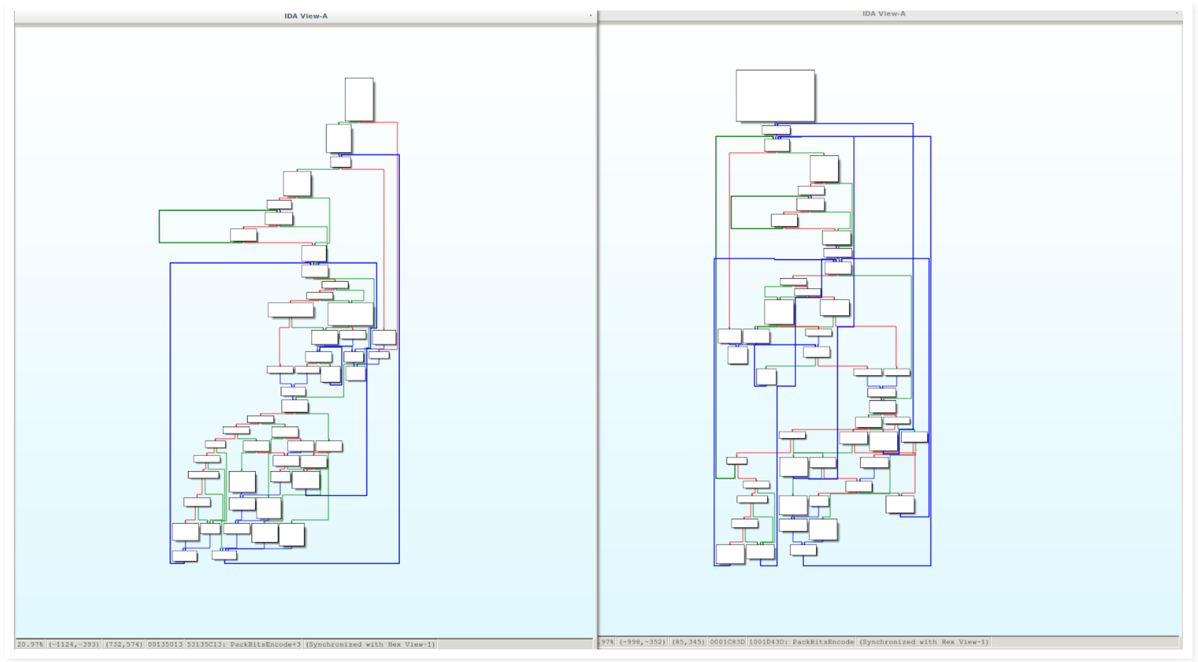
Allright, finally we can scan through the DLLs in a directory:

```
for i in $(find /media/DLLs -iname ./*.dll); do echo $i; ./matchfunctionsindex --
index=/var/tmp/work/simhash.index --input $i; done
```

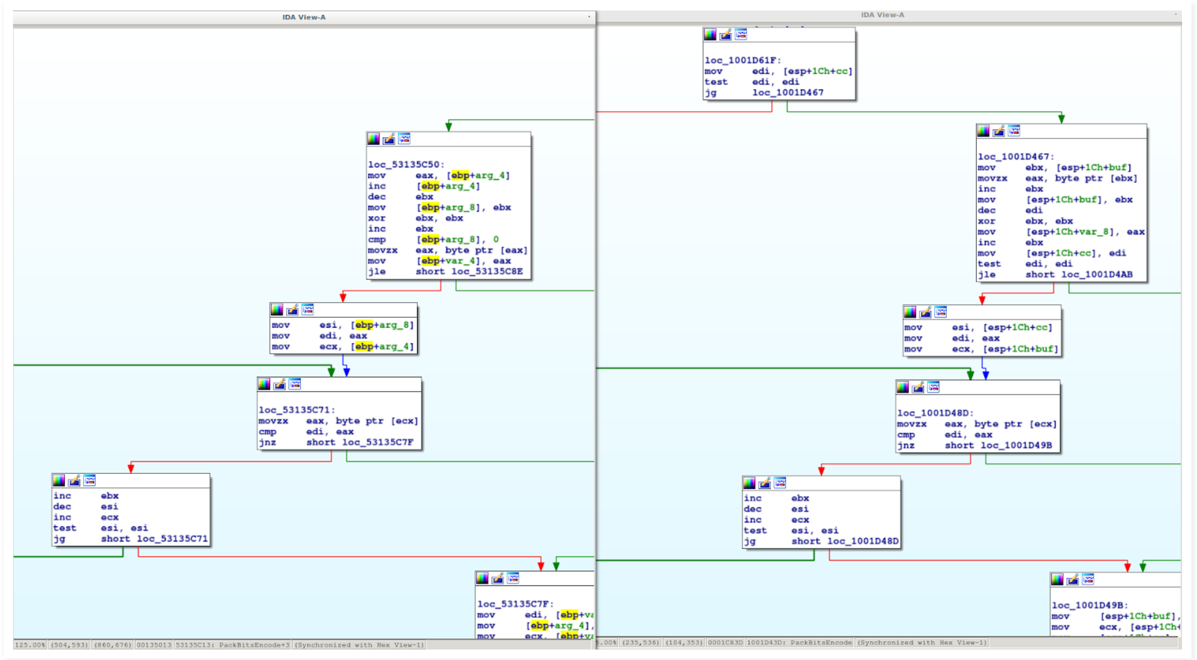
We will get commandline output similar to the following:

```
/home/thomasdullien/Desktop/sources/adobe/binaries/AGM.dll
[!] Executable id is 8ce0e5a0e1324b15
[!] Loaded search index, starting disassembly.
[!] Done disassembling, beginning search.
[!] (1231/7803 - 8 branching nodes) 0.843750: 8ce0e5a0e1324b15.608033d matches
36978e7b9d396c8d.10021978 /home/thomasdullien/Desktop/tiff-3.9.5-
builds/PE/vs2013.32bits.O1/libtiff.dll std::basic_string<char, std::char_traits<char>,
std::allocator<char> >::_Copy(unsigned int, unsigned int)
[!] (1231/7803 - 8 branching nodes) 0.820312: 8ce0e5a0e1324b15.608033d matches
53de1ce877c8fedd.10020e8b /home/thomasdullien/Desktop/tiff-3.9.5-
builds/PE/vs2012.32bits.O1/libtiff.dll std::basic_string<char, std::char_traits<char>,
std::allocator<char> >::_Copy(unsigned int, unsigned int)
[!] (1236/7803 - 7 branching nodes) 0.828125: 8ce0e5a0e13i24b15.608056e matches
36978e7b9d396c8d.100220d4 /home/thomasdullien/Desktop/tiff-3.9.5-
builds/PE/vs2013.32bits.O1/libtiff.dll std::basic_string<char, std::char_traits<char>,
std::allocator<char> >::assign( std::basic_string<char, std::char_traits<char>,
std::allocator<char> > const&, unsigned int, unsigned int)
(...)
/home/thomasdullien/Desktop/sources/adobe/binaries/BIBUtils.dll
[!] Executable id is d7cc3ee987ba897f
[!] Loaded search index, starting disassembly.
[!] Done disassembling, beginning search.
(...)
/media/dlls/Windows/SysWOW64/WindowsCodecs.dll
[!] Executable id is cf1cc98bead49abf
[!] Loaded search index, starting disassembly.
[!] Done disassembling, beginning search.
[!] (3191/3788 - 23 branching nodes) 0.851562: cf1cc98bead49abf.53135c10 matches
39dd1e8a79a9f2bc.1001d43d /home/thomasdullien/Desktop/tiff-3.9.5-
builds/PE/vs2015.32bits.O1/libtiff.dll PackBitsEncode( tiff*, unsigned char*, int,
unsigned short)
[!] (3192/3788 - 23 branching nodes) 0.804688: cf1cc98bead49abf.53135c12 matches
4614edc967480a0d.1002329a /home/thomasdullien/Desktop/tiff-3.9.5-
builds/PE/vs2013.32bits.O2/libtiff.dll
[!] (3192/3788 - 23 branching nodes) 0.804688: cf1cc98bead49abf.53135c12 matches
af5e68a627daeb0.1002355a /home/thomasdullien/Desktop/tiff-3.9.5-
builds/PE/vs2013.32bits.Ox/libtiff.dll
[!] (3192/3788 - 23 branching nodes) 0.804688: cf1cc98bead49abf.53135c12 matches
a5f4285c1a0af9d9.10017048 /home/thomasdullien/Desktop/tiff-3.9.5-
builds/PE/vs2017.32bits.O1/libtiff.dll PackBitsEncode( tiff*, unsigned char*, int,
unsigned short)
[!] (3277/3788 - 13 branching nodes) 0.828125: cf1cc98bead49abf.5313b08e matches
a5f4285c1a0af9d9.10014477 /home/thomasdullien/Desktop/tiff-3.9.5-
builds/PE/vs2017.32bits.O1/libtiff.dll
```

This is pretty interesting. Let's load `WindowsCodecs.dll` and the `libtiff.dll` with the best match into IDA, and examine the results:


IDA View-A
IDA View-A
20.97% (-1124,-393) (732,574) 00135013 53135C13: PackBitsEncode+3 (Synchronized with Hex View-1)
.97% (-998,-352) (85,345) 0001C83D 1001D43D: PackBitsEncode (Synchronized with Hex View-1)

At this zoom level, the two functions do not necessarily look terribly similar, but zooming in, it becomes apparent that they do share a lot of similarities, both structural and in terms of instruction sequences:



```
loc_53135C50:
mov     eax, [ebp+arg_4]
inc     [ebp+arg_4]
dec     ebx
mov     [ebp+arg_8], ebx
xor     ebx, ebx
inc     ebx
cmp     [ebp+arg_8], 0
movzx   eax, byte ptr [eax]
mov     [ebp+var_4], eax
jle     short loc_53135C8E

mov     esi, [ebp+arg_8]
mov     edi, eax
mov     ecx, [ebp+arg_4]

loc_53135C71:
movzx   eax, byte ptr [ecx]
cmp     edi, eax
jnz     short loc_53135C7F

inc     ebx
dec     esi
inc     ecx
test    esi, esi
jg      short loc_53135C71

loc_53135C7F:
mov     edi, [ebp+va
mov     [ebp+arg_4],
mov     ecx, [ebp+va
```

```
125.00% (504,593) (860,676) 00135013 53135C13: PackBitsEncode+3 (Synchronized with Hex View-1)
```

```
loc_1001D61F:
mov     edi, [esp+1Ch+cc]
test    edi, edi
jg      loc_1001D467

loc_1001D467:
mov     ebx, [esp+1Ch+buf]
movzx   eax, byte ptr [ebx]
inc     ebx
mov     [esp+1Ch+buf], ebx
dec     edi
xor     ebx, ebx
mov     [esp+1Ch+var_8], eax
inc     ebx
mov     [esp+1Ch+cc], edi
test    edi, edi
jle     short loc_1001D4AB

mov     esi, [esp+1Ch+cc]
mov     edi, eax
mov     ecx, [esp+1Ch+buf]

loc_1001D48D:
movzx   eax, byte ptr [ecx]
cmp     edi, eax
jnz     short loc_1001D49B

inc     ebx
dec     esi
inc     ecx
test    esi, esi
jg      short loc_1001D48D

loc_1001D49B:
mov     [esp+1Ch+buf],
mov     ecx, [esp+1Ch
```

```
.00% (235,536) (104,353) 0001C83D 1001D43D: PackBitsEncode (Synchronized with Hex View-1)
```

What really gives us confidence in the non-spuriousness of the result is (...drumroll...) the name that IDA obtained for this function from the Microsoft-provided PDB debug symbols: `PackBitsEncode`.

Closer examination of `WindowsCodecs.dll` reveals that it contains a fork of `libtiff` version 3.9.5, which Microsoft changed significantly. We have not investigated how Microsoft deals with backporting security and reliability fixes from upstream. Since `libtiff` links against libjpeg, it should perhaps not surprise us that the same DLL also contains a modified libjpeg fork.

Summary, Future Directions, Next Steps

What has been learnt on this little adventure? Aside from many details about building similarity-preserving hashes and search indices for them, I learnt a few interesting lessons:

Lessons Learnt

The search index vs linear sweep - modern CPUs are fast at XOR

It turns out that modern CPUs are extremely fast at simply performing a linear sweep through large areas of memory. A small C program with a tight inner loop which loads a hash, XORs it against a value, counts the resulting bits, and remembers the index of the "closest" value will search through hundreds of millions of hashes on a single core.

The algorithmic break-even for the locality-sensitive-hashing index is not reached until way north of a few hundred million hashes; it is unclear how many people will even have that many hashes to compare against.

It is possible that the clever search index was over-engineered, and a simple linear sweep would do just as well (and be more storage-efficient).

Competing with simple string search

For the stated problem of finding statically linked libraries, it turns out that in the vast majority of cases (personally guess 90%+), searching for particularly expressive strings which are part of the library will be the most effective method: Compilers generally do not change strings, and if the string is sufficiently unusual, one will obtain a classifier with almost zero irrelevant results and a reasonably high true positive rate.

The heavy machinery that we explored here is hence most useful in situations where individual snippets of code have been cut & pasted between open-source libraries. Of the real-world cases we examined, only mpengine.dll fits the bill; it is an open question how prevalent cutting-and-pasting-without-strings is.

An interesting research question with regards to existing published results is also: What added value does the method provide over simple string search?

The problem is still hard

Even with all the engineering performed here, we can only reliably find about 40% of the cases we care about - and likely even fewer if a compiler is involved to which we do not have access. There is a lot of room to improve the method - I optimistically thinks it should be possible to reach a true positive rate of 90%+ with a small number of irrelevant results.

It sounds like an interesting question for the ML and RE community: Can embeddings from disassemblies into Hamming-space be learnt that achieve much better results than the simple linear model here? At what computational cost?

Future directions / next steps

There are a number of directions into which this research could (and should) be expanded:

1. Re-writing the machine learning code in TensorFlow or Julia (or any other setup that allows efficient execution on the GPU). The current code takes days to train with 56-core server machines mainly because my desire to write the loss function directly in C++. While this is elegant in the framework of a single-language codebase, using a language that allows easy parallelization of the training process onto a GPU would make future experimentation much easier.
2. Swapping L-BFGS for the usual SGD variants used in modern machine learning. As the quantity of training data increases, L-BFGS scales poorly; there are good reasons why almost nobody uses it any more for training on massive quantities of data.
3. Triplet and quadruplet training. Various recent papers that deal with learning embeddings from data [Triplet]. From an intuitive perspective this makes sense, and the training code should be adapted to allow such training.
4. Better features. The set of features that are currently used are very poor - mnemonic-tuples, graphlets, and large constants that are not divisible by 4 are all that we consider at the moment; and operands, structure offsets, strings etc. are all ignored. There is clearly a lot of valuable information to be had here.
5. Experiments with Graph-NNs. A lot of work on 'learning on graphs' has been performed and published in the ML community. Exciting results in that area allow learning of (very simple) graph algorithms (such as shortest path) from examples, it is plausible that these models can beat the simple linear model explored here. CCS ‘17 uses such a model, and even if it is

hard for me to judge what part of their performance is due to string matching and what part is the rest of the model, the approach sounds both promising and valid.

6. Using adjacency information. Functions that were taken from another library tend to be adjacent in binaries; a group of functions that come from the same binary should provide much stronger evidence that a third-party library is used than an isolated hit.
7. Replacing the ANN tree data-structure with a flat array. Given the speed of modern CPUs at linearly sweeping through memory, it is likely that the vast majority of users (with less than 100m hashes) does not require the complex data structure for ANN search (and the resulting storage overhead. For the majority of use-cases, a simple linear sweep should be superior to the use of bit-permutations as LSH family.

The end (for now).

If you have questions, recommendations, or (ideally) pull requests: Please do not hesitate to contact the authors on the relevant github repository here.

Posted by Ben at 10:37 AM   No comments:

Newer Posts   Home   Older Posts

Subscribe to: Posts (Atom)