On Code Diversity and Static Malware Similarity

Adam Duby
University of Colorado, Colorado Springs
aduby@uccs.edu

***Abstract*— Automated tools for detecting similar malware variants and code reuse in malware samples save malware analysts time to focus on new and otherwise uncategorized malware. Fuzzy hashing, n-grams, function call signatures, dependency analysis, and code normalization are the most common static techniques for identifying similar code fragments and malware similarity. Code diversification and randomization as a defense has been an active parallel research effort. Code diversification and randomization technology aims to create an unreliable and unpredictable attack surface. Todays software terrain offers developers a plethora of programming languages, compilers, source-code transformers, target architectures, and code randomization algorithms. Thanks to this diversified landscape, malware developers can intentionally create numerous variants of the same malware to avoid static signature detections.**

In this paper, we offer a categorization of existing malware similarity techniques and discuss the impact of code diversity on each class. We identify gaps in existing knowledge and offer insight for further research in malware similarity detection that is resilient to code diversification.

I. Introduction

Combating malware (malicious software) is an evolving and dynamic endeavor in the modern cyber battlefield. Defenders are charged with developing signatures and detection engines for the prolific malware landscape. New malware variants are being introduced frequently, and the defense is charged with maintaining pace to analyze, signature, detect, and mitigate. McAfee Labs observed approximately 80 million new samples in the first half of 2018 [1]. Manually treating each sample in isolation does not scale well for malware analysts. Automated similarity techniques afford analysts the time to focus on new and unknown malware tactics.

Manual malware reverse engineering is time consuming and often does not reveal new valuable threat intelligence if the malware under analysis is simply a variant of an existing malware family. Each sample (a compiled malware binary) is rarely an entirely new malware. Malware authors utilize code reuse, repurposing existing code and modules to provide a new spin on a known attack vector [2], [3], [4]. Malware can also vary depending on the target [5] and the campaign [6]. Similarity detection aims to identify code reuse by identifying semantic and syntactic similarities between samples. This lets researchers identify malware trends, evolution, and rapidly deploy new signatures.

Malware attempts to address the code reuse detection problem through packing, obfuscation, and binary diversification. Packed malware utilizes small wrapper code to decompress the malware at runtime. This limits the exposure of the semantics to manual and static analysis. Detection engines are fairly effective at classifying packed malware as malicious due to the high entropy in the code and the unusual code section headers [7], [8].

Code and data obfuscation aims to generate software samples that are challenging to reverse engineer [9]. This creates difficulty for malware analysts to easily derive semantics from obfuscated malware [10], [11]. This increased overhead on the analyst can also thwart similarity detection efforts due to the code modifications required during deobfuscation. Automated and static code deobuscation efforts have been proposed in [12], [13], [14], [15].

Similar to obfuscation is malware diversity [16] and software randomization [17], [18]. Software randomization and diversification research originally stemmed from a defensive perspective to create uncertainty in the attack surface. Uncertainty is created by generating a unique, yet semantically equivalent, compiled binary for each compilation. Larsen et al. [18] provide a systemization for compiler-generated software randomization techniques from a software defensive perspective.

While software randomization via the compiler is explicit, there are also implicit sources of diversification. Choice of source code, compiler options, target architecture, and linkage options can vastly impact the compiled output. Variations in these options can create syntactically unique yet semantically equivalent malware variants. In this paper, we refer to intentional compiler generated software and code randomization as discussed in [18] as software randomization. We refer to the implicit differences introduced by architecture and compiler toolchains as software diversification.

Malware similarity research (or code similarity in the general case) needs to address these various sources of randomization and diversity in the modern dynamic landscape. Similarity detection aims to identify similar code fragments in a malware corpus to classify malware into families or categories. Microsoft sparked a renewed research interest in this problem with its malware classification challenge announced in 2015 [19]. We can classify similarity approaches into two broad categories: dynamic analysis and static analysis. Dynamic analysis involves the execution of the malware sample to classify the malware based on its observed runtime behavior. Static analysis does not involve code execution and only relies on the compiled sample in its native format. Our paper focuses on static analysis techniques for several reasons. First, static analysis reduces the overhead required

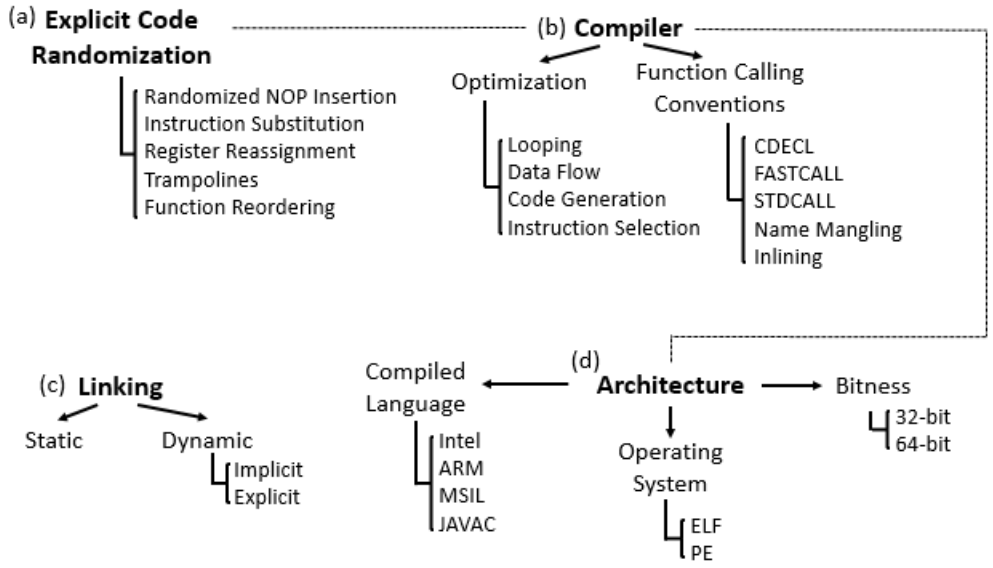

Fig. 1. Taxonomy of software diversity sources introduced by (a) explicit and intentional software randomization, (b) compiler variations, (c) linking methods, and (d) the target architecture. Solid lines represent hierarchical relationships while dotted lines represent more loosely coupled relationships.

to execute the sample. Organizations may not have the computational resources to execute every sample. Second, dynamic analysis is vulnerable to deviations in the execution environment. For example, malware may behave differently based on conditionals and may take varying execution paths. Thus, we scope our discussion on the static techniques and discuss potential issues introduced by code randomization and diversification.

With this systematization of knowledge paper, we make the following contributions:

- categorize sources of code diversity into a general taxonomy shown in Figure 1;
- categorize existing static malware similarity detection approaches into a general taxonomy shown in Figure 5;
- discuss the advantages and disadvantages of each malware similarity approach under malware diversification;
- and propose the necessary criteria for a new static solution that is resilient to diversification.

Our discussion is scoped with the assumptions that the samples under analysis are:

- deobfuscated and unpacked;
- not encrypted;
- and in compiled executable format.

II. Code Diversity and Randomization

Diversity in malware samples can be introduced by various phases of the compilation tool chain. The choice of compiler, compiler flags, linking options, and target architecture can drastically alter the compiled output. Further, explicit malware diversity [16] intentionally introduces randomness into the compiled output. In theory, a malware sample X of the some source code can produce N-variants, where each variant is semantically equivalent but implemented differently.

Source to source code transformers are also available to change high level source code implementations (insert citation here). For example, the same malware sample X written in the C programming language can be converted to C++. The changes in compilation will in turn alter the compiled binary.

We conceptualize the sources of diversity and randomness in Figure 1. Our high level taxonomy provides researchers with a quick reference for understanding where variations in compilation output can originate. We observe that there are four main sources to consider:

- explicit code randomization;
- compiler options;
- linking options;
- and the target architecture.

A. Explicit Code Randomization

Explicit and intentional code randomization can be introduced in the optimization phase of the compiler toolchain, which we conceptualize in Figure 2. One of the simplest techniques that the compiler can utilize to generate diversity is randomized NOP insertion. The NOP instruction (opcode 0x90 in x86), simply consumes a clock cycle, and does not impact any of the flags on the processor status register. It is therefore an intuitive candidate to randomize the binary without impacting semantics. Randomized NOP insertion creates different offsets and relative addresses for each instance of the compiled application and has a large effect on the relocation tables. Each instance of the application will have its code and return addresses at randomized locations.

Instruction substitution is a compiler technique where different instructions are generated. Observing that the x86_64 instruction set is a variable length architecture, instruction level variation will impact the offsets and relocations. To

TABLE I
INSTRUCTION-LEVEL VARIATIONS

Instructions (x86 Intel Syntax)	Object Code (MASM)
`xor eax, eax`	`33C0`
`mov eax, 0`	`B800000000`
`sub eax, eax`	`2BC0`
`mov ebx, eax; xor eax, ebx`	`8BD833C3`

observe the variation in object code with semantically equivalent instructions, we generated some samples using the MASM assembler in Table I. Observe that each instruction performs the same task, but with deviations in object code. Similarly, instruction permutation randomizes the order in which the instructions are generated. This is dependent on pipelining and register dependencies, but can it can introduce large variance in the compiled object code size and instructions.

Garbage code insertion is a technique that inserts dead code and semantically useless code into the compiled binary. This can include superfluous API calls, function calls, and random instructions. This introduces additional overhead for disassembly and reverse engineering.

Function shuffling is another compiler approach in which the nature of function calls are randomized. This includes function calling notations, introducing polymorphic function arguments, and randomizing the function calling order. This can be extended to randomize the order of library API calls.

Another compiler technique was introduced in Readactor [20]. They observed that ROP and JIT-ROP [21] attacks require memory disclosure from code pointers. This includes function pointers, return addresses, and dynamic linker structures (such as the PLT and GOT in ELF binaries). Thus, Readactor implemented an LLVM modification that creates code trampolines to prevent memory disclosure. All pointers, addresses, and control flow information are replaced with a jump instruction into a randomized execute-only trampoline section. The trampoline then simply redirects control to the desired call site. They implemented the execute-only permissions via a thin hyperviser that uses Intel Extended Page Tables (EPT). Therefore, the true layout of the code cannot be determined at runtime due to the added layer of indirection. Readactor also implemented code randomization techniques, such as function reordering and randomized NOP insertion.

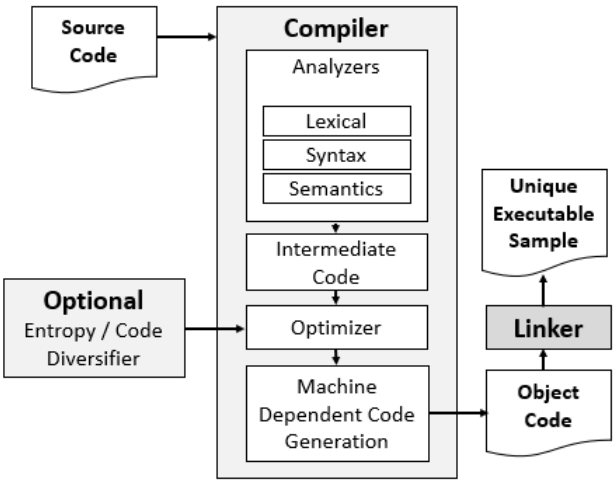

Fig. 2. Compiler Generated Randomization

B. Compiler Variations

Different compilers and compilation options can produce semantically equivalent variations of the same malware. For example, variations in optimization levels and function calling conventions can create different compiled outputs.

Various levels of aggressiveness in code optimization impacts the code size, instruction selection, and control flow. Optimization that impact code layout include loop unwinding, function inlining, unused code removal, and register selections.

Function calling convention can be explicitly declared by the programmer, or chosen by the compiler. Consider argument passing to a function call. There are three common ways to pass arguments into a function. The Visual Studios (VS) compiler typically pushes arguments onto the stack before a function call. The GCC compiler typically moves arguments onto the stack. The third alternative is to follow an ABI, where arguments are placed into specific registers. This is common with x64 applications and the FASTCALL function calling convention. Consider the following example function:

```
int foo(int a, int b){return a * b;}
int x = foo(3, 5);
```

Using standard CDECL calling convention in VS, the calling code may look like:

```
push   5
push   3
call   _foo
add    esp, 8
```

Alternatively, the calling code can use the `mov` instruction to move the arguments onto the stack, which is the default in the GCC compiler:

```
mov    [esp-04h], 5
mov    [esp], 3
call   _foo
add    esp, 8
```

Notice in the above examples that the stack restoration code (`add esp, 8`) is performed by the caller. This is in accordance with the CDECL calling convention. Compilers that opt to use the STDCALL calling convention make sure that stack restoration is the responsibility of the callee. In our example, the called function will be compiled to include a `ret 8` instruction at the end of the function, where 8 is the number of bytes to by popped off the stack.

The FASTCALL calling convention assigns registers to arguments to avoid argument passing via the stack altogether. As an example, our calling code can be compiled as:

```
mov    eax, 5
mov    edx, 3
call   @foo@8
```

Also notice how this introduces function name decorations by appending the number of bytes required for the arguments to the function name. Alternatively, the compiler can opt to inline this small function, completely removing the `call` instruction altogether. Inlining not only changes the code layout, but changes control flow of program execution.

Variations in calling convention can

C. Linker Variations

Variations in linking methods create variations in code size, import tables, and control flow. For this discussion, we consider static linking, dynamic linking, and explicit runtime dynamic linking, also known as delay loading.

To understand the mechanics of linking, we consider compiled Windows binaries (.exe and .dll), which adhere to the Portable Executable file structure, or the PE32 for 32-bit binaries and PE32+ for 64-bit binaries. During compilation, a section in the PE32 binary called the `.reloc` section is built to map symbols, certain functions, and direct memory addressing to the appropriate call sites. This enables position independence through relocatable relative virtual addresses (RVAs) and offsets. This also enables ASLR to map the base image of the executable into a random memory page.

The compiler also creates an import directory. The import directory contains an Import Name Table (INT) that contains a list of the names of the dynamically linked DLLs and the exported functions from said DLLs that the program invokes. At compile time, a section is reserved at the end of the program's `.text` section. This special section, called the `.idata` section, will be filled in at load time with the addresses of the call sites for the imported functions. When the program is first loaded, the linker and loader make a pass at the program to examine the DLLs in the INT. Those DLLs then get mapped to the process's address space. Then the RVAs in the INT are examined to find where in the `.idata` section the linker should insert the pointer to the appropriate call site. This list of pointers is called the Import Address Table (IAT). Similarly, DLLs contain an export directory which contains a list of exported functions in an export name table (ENT). The export directory also contains the RVAs of the exported functions' call sites. The load-time linking process resolves a process's IAT by examining the DLL's export directory in order to fill in the correct pointer in the `.idata` section. Linux ELF binaries store this information in the procedure linkage table (PLT) and global offset table (GOT).

Statically linking library files attaches the library's code base to the compiled application. This clearly increases the size and introduces large code blocks opposed to dynamic linking. This also affects the application's import directory. When the library code is linked statically, a library function's call site is located within the image itself, and therefore there are no entries in the import directory.

Applications can also be compiled to use runtime explicit linking, commonly called delay loading in Windows. A delay loaded library is not mapped to the process address space until needed, and therefore the compile-time linker does not include an entry in the import directory. The operating system's dynamic loader will load the library when explicitly called to do so through the `LoadLibrary` API call. In the below example, `myDLL.dll` is delay loaded, and will not appear in the application's import table.

```
typedef int (*foo)(void);
foo _foo;
HINSTANCE h = LoadLibrary("myDLL.dll");
_foo = (foo)GetProcAddress(h, "foo");
_foo();
```

D. Architectural Differences

The target architecture dictates the machine language, bitness, and operating system API used by the malware. First, we consider the architectural differences between applications and their target operating systems. Linux applications are compiled into ELF format, while Windows applications are compiled into the PE file format. Each specification varies in section names, headers, and sizes. A simple hello-world program compiled into an ELF looks different than from a PE. Although the code constructs may appear very similar, items such as library imports and the construction of the application control flow vary greatly.

The instruction set architecture (ISA) dictates the compiled application's machine code. As an example, consider an application that is compiled for an ARM processor, and again for an x86 or AMD processor. The ARM ISA is a fixed-length RISC architecture, as opposed to the variable length x86 architecture. There are also variations in register options, and of course the actual machine code itself. Thus, semantically equivalent applications compiled on different architectures will appear drastically different at the binary level. As an example, below we show the compiled ARM assembly output from the calling code from our previous `foo(3, 5)` example.

```
mov    r1, #5
mov    r0, #3
bl     foo
str    r0, [fp, #-8]
mov    r3, #0
mov    r0, r3
sub    sp, fp, #4
```

We also consider variations in the operating system API. Suppose a malware sample that interacts with the NTFS file system on a Windows target needs to call the `CreateFile` function. The developer can either invoke the Win32 API `CreateFileA` function or invoke the native system call ABI for `NtCreateFile`. The native API called from user code is less common and not advised for legitimate

```
6A 00           push    0               ; hTemplateFile
68 80 00 00 00  push    80h             ; dwFlagsAndAttributes
6A 02           push    2               ; dwCreationDisposition
6A 00           push    0               ; lpSecurityAttributes
6A 07           push    7               ; dwShareMode
68 00 00 00 C0  push    0C0000000h      ; dwDesiredAccess
8B 4D F0        mov     ecx, [ebp+lpFileName]
51              push    ecx             ; lpFileName
FF 15 08 C0 40 00 call  ds:CreateFileA
89 45 E8        mov     [ebp+var_18], eax
```

(a) x86 Compiled Code

```
48 C7 44 24 30 00 00 00+mov     [rsp+78h+hTemplateFile], 0 ; hTemplateFile
C7 44 24 28 80 00 00 00 mov     [rsp+78h+dwFlagsAndAttributes], 80h ; dwFlagsAndAttributes
C7 44 24 20 02 00 00 00 mov     [rsp+78h+dwCreationDisposition], 2 ; dwCreationDisposition
45 33 C9                xor     r9d, r9d        ; lpSecurityAttributes
41 B8 07 00 00 00       mov     r8d, 7          ; dwShareMode
BA 00 00 00 C0          mov     edx, 0C0000000h ; dwDesiredAccess
48 8B 4C 24 58          mov     rcx, [rsp+78h+lpFileName] ; lpFileName
FF 15 72 9F 00 00       call    cs:CreateFileW
48 89 44 24 60          mov     [rsp+78h+var_18], rax
```

(b) x64 Compiled Code

Fig. 3. Example of Differences in 32-bit and 64-bit Compiled Applications

applications because the API can vary between OS versions. However, it can still be used to achieve the same semantic result. Switching between APIs changes the application's dependencies and import tables. `CreateFile` is called from `kernel32.dll` while `NtCreateFile` is called from `ntdll.dll`. Further, each API also varies in the number of parameters and parameter data types, as shown below.

```
HANDLE CreateFileA(
  LPCSTR           lpFileName,
  DWORD            dwDesiredAccess,
  DWORD            dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD            dwCreationDisposition,
  DWORD            dwFlagsAndAttributes,
  HANDLE           hTemplateFile
);

__kernel_entry NTSTATUS NtCreateFile(
  OUT PHANDLE      FileHandle,
  IN ACCESS_MASK   DesiredAccess,
  IN POBJECT_ATTRIBUTES ObjectAttributes,
  OUT PIO_STATUS_BLOCK IoStatusBlock,
  IN PLARGE_INTEGER AllocationSize,
  IN ULONG         FileAttributes,
  IN ULONG         ShareAccess,
  IN ULONG         CreateDisposition,
  IN ULONG         CreateOptions,
  IN PVOID         EaBuffer,
  IN ULONG         EaLength
);
```

Many Win32 functions also have ASCII and Unicode versions. For example, The Unicode variant for the Win32 `CreateFile` API is `CreateFileW` while the ASCII variant is `CreateFileA`. Both have identical APIs, except that the first argument for `CreateFileW` is of type LPCWSTR for wide characters. ASCII and Unicode strings are also stored differently in the application's data section. Developers also have the choice of using other libraries, such as the standard C library to create file. Switching between native, ASCII, Unicode, and standard C APIs is another example where minor variations impact the application's import table and code layout.

Bitness offers another source of architecturally generated software diversity. 32-bit (x86) applications are compiled very differently from their 64-bit (x64) counterparts. A motivating example shown in figure 3. Here, we have two variations compiled from identical source code. As discussed earlier, the x64 code uses the FASTCALL convention while the x86 variant utilizes CDECL. The x64 code also calls imported functions quite differently. The operand to the `call` instruction is an offset from instruction pointer, indicating the call site. In x86 code, the `call` instruction's operand is the address of the call site. This address is changed at load time to correspond to the actual address in the process address space. The `.reloc` section contains the offsets for the loader to check and change call site addresses to enable ASLR compliance and position independence so the image can be mapped to any base address in memory. The other obvious differences between 32-bit code and 64-bit code are in the size of operands and choice of registers, which drastically impact the compiled object code.

E. Source Code Transformations

Malware authors can also deviate between source code languages to inflict diversity. Source-to-source code transformers such as the ROSE compiler [22] offer developers a platform to transform code between languages. Choice of source code impacts the compilation toolchain and therefore the compiled application. Consider a simple program that prints "Hello" to standard output written in C and again in C++. These semantically equivalent applications have different import dependencies, function calling conventions, control flow, and object code. Figure 4 shows the compiled C++ code, which uses the `std::cout` object instead of the C `printf` function. C++ function names are also mangled by the compiler to support function overloading.

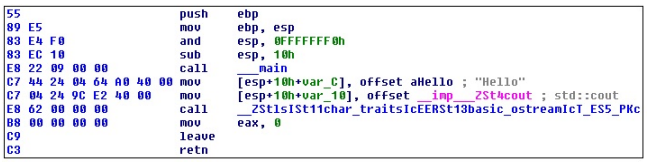

Fig. 4. Compiled C++ Program

Programming languages that use an intermediate representation and a runtime interpreter offer another source of diversity. For example, applications written in Java are compiled into Java bytecode, which is then executed by the JVM at runtime. Microsoft .NET applications use the MSIL intermediate language. Therefore, the same malware can be written in various languages and their compiled counterparts will appear very different.

III. Static Malware Similarity

One of the largest challenges with compiled code and malware similarity is all of the potential diversification discussed in the previous section. Changes in implementation and the compilation tool chain drastically impact the compiled output, rendering semantic static similarity a challenge [23]. In this section, we discuss common techniques for static malware similarity and classification observed in the literature. We propose a taxonomy of techniques in Figure 5. Our classification defines four main techniques for static code similarity techniques:

- hashing;
- extracting metadata;
- code normalization;
- and deriving semantics aware signatures based on functions and graph analysis.

The taxonomy is provided as a guide to classify the techniques in a high level in order to guide our discussion on the implications of code diversity on these techniques.

A. Hashing

Numerous similarity hashing and calculation techniques have been proposed over the years to move away from integrity hashing, such as SHA and MD5. Integrity hashes are obviously vulnerable to the smallest deviations, making them useless for classifying malware. Approximate matching [24] uses similarity metrics to identify a percentage of similarity between samples. This is often called fuzzy hashing, and the ssdeep fuzzy hash has long been the industry adopted standard. Locality sensitive hashing (LSH) algorithms, such as TLSH, attempt to represent the distance between samples, similar to a Hamming distance between data sets.

It has already been shown in previous studies that hashing algorithms are vulnerable to code diversity. [25] discussed bytewise approximate matching and its associated shortfalls and [26] performed an analysis of various fuzzy hashing. Recently, Pagani et al [27] analyzed the effectiveness of various fuzzy hashing techniques to classify malware of similar variants. There study included a discussion on identifying the same source from different compiler flags, but they did not consider explicit compiler generated diversification as discussed previously in section 4.1. Their study was able to shine some light on why certain similarity algorithms work better than others in various situations. We summarize their findings as follows:

- ssdeep only works for files in similar size and with very few modifications.
- tlsh is preferable when dealing with source code changes.
- sdhash is preferable when dealing with compiler toolchain changes.

Although software diversity originated as a defensive strategy, Payer et. al. discussed that this can be re-purposed by adversaries to generate unique malware samples that thwart static similarity and detection [16]. Payer further showed that intentional diversification and code randomization thwarts common approximate matching metrics [4]. For example, they tested the Jaccard Similarity (JS) coefficient, shown in equation 1, which divides the size of the intersection of the two samples under comparison by the size of the union of said samples [28].

$$JS(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{1}$$

[4] concluded that JS can be effective for minor deviations in samples, but is not effective when garbage code and excessive randomization is inserted into the sample.

They discuss how very small changes can throw off similarity detection due to offsets and relocations changing. Their findings are in accordance with [29]. As discussed earlier with compiler diversity, these changes are desired for randomizing an attack surface. The findings are interesting, but we feel there is more work to be done in this field. For example, this study should consider explicit malware diversity as discussed in [16]. Further, the authors did not consider 32-bit versus 64-bit files. The difference in compilation and object code between 32-bit and 64-bit binaries is vast. When we reverse engineered our application in Figure 3, we observed that relocations are not used as heavily in 64-bit application as in 32-bit binaries. That is because x64 introduced the ability to reference call sites and jump destinations as offsets from the instruction pointer. This alleviates a need for the linker to resolve the destination operands from the relocation tables. Further, the function calling conventions vary drastically as previously shown.

Upchurch proposed malware provenance, which incorporated context triggered piecewise hashing in conjunction with operand masking and a sliding window approach [30]. The operand masking only considers the first byte of the object code, conceptually normalizing the n-grams under analysis. Although this is a step towards similarity metrics that are diversity agnostic, it still relies too heavily on object code that is vulnerable to code diversity.

B. Metadata

Compiled applications, malware included, contain a rich amount of metadata in the compiled file structure. Research

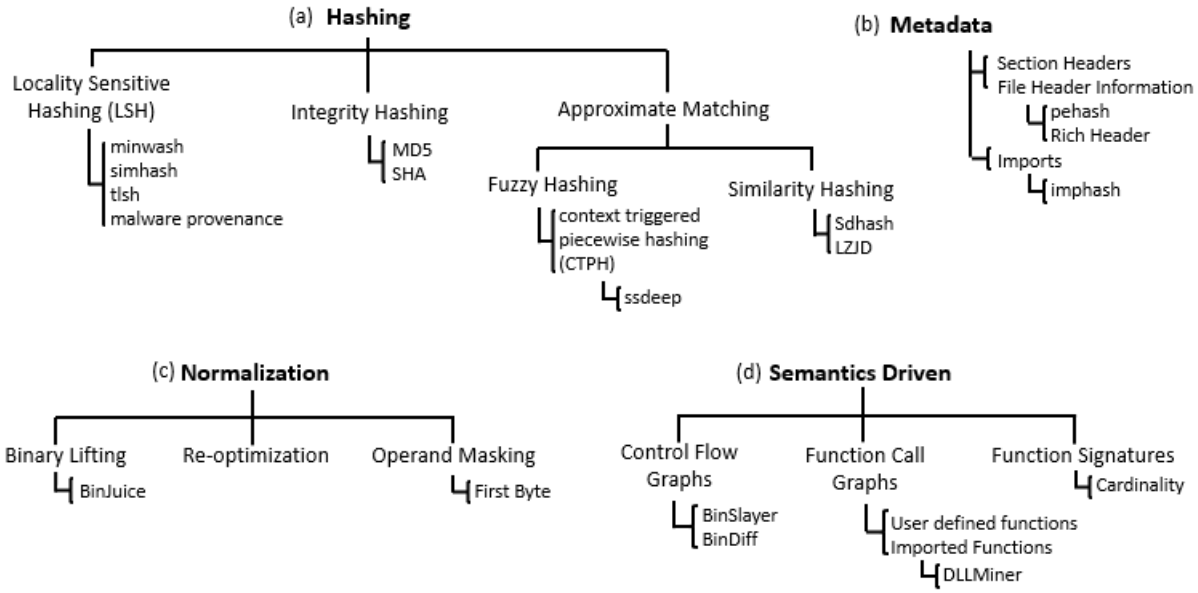

Fig. 5. Taxonomy of static similarity techniques classified into (a) hashing, (b) metadata feature extraction, (c) code normalization, and (d) semantics driven signatures. Literature has proposed techniques that combine two or more of these techniques.

has proposed utilizing this information to find similarity in applications. This approach does not look inside of the object code, and instead leverages the file's metadata and information. PeHash proposed extracting information from the portable executable (PE) file structure to find similar samples [31]. Similarly, Kirda et. at. proposed PE Miner, which uses structural information to classify similar files [32]. Choi et. al. also formalized an approach for malware classification based on generic file information [33]. Feature extraction has also been proposed to train machine learning techniques for similarity and classification [34].

An interesting and novel approach demonstrated by Polychronakis et. al. showed that the PE contains valuable intelligence in the Rich Header [35]. This header information is generated by the visual studios compiler and contains unique application signatures that can be utilized to find variants of the same application.

An application's dependencies in the import directory offer insight into the program's semantics and structure.. Leveraging this information is another technique to find similar malware variants [36]. One popular way to find variants of the same malware is to calculate the hash of the import directory, called the ImpHash [37] [33]. Intuitively, this similarity technique should be effective because it is opcode agnostic and can therefore overcome compiler diversification. However, the ImpHash does not consider delay loading, embedded executable images hidden elsewhere in the PE32 that may contain an import directory, or obfuscated API calls [11] [38] [39].

Metedata, dependencies, and structural information extraction has not been tested against diversified samples to assess their vulnerability to code diversity. Static linking and architectural variations, for example, drastically changes the code layout size, structure, and dependencies. Various language architectures also create very different compiled files whose metadata changes in accordance with the operating system's and language's standards. Delay loaded libraries are also an issue for finding similarities based on dependencies.

C. Code Normalization

With the research community identifying the issue of code variations, code normalization has been an active research area for finding similar code fragments. Early work in code abstraction laid the groundwork for lifting compiled applications into a normalized and canonical layout [40].

Code normalization for malware similarity was first addressed in [41]. However, their technique was not resilient to anti-disassembly tactics and was very susceptible to deviations in control flow. Preda et. al. opted to use program dependencies for normalization and abstraction [42], but this technique is vulnerable to changes in the import tables for the same reasons discussed in the previous section.

Leveraging the LLVM compiler's intermediate representation for binary lifting was proposed in [43]. This interesting approach abstracts the architectural and object code variations by lifting the application into the LLVM's intermediate language. David et. al. took this a step further by lifting and then optimizing the lifted intermediate representation to abstract away any deviations in instructions.

Binary lifting and normalization is a technique that is leaning towards overcoming diversity. David et al. showed that lifting the binary into an intermediate representation, then normalizing the code can abstract away differences presented by compilers and architectures [44]. This approach is object code and architecture agnostic. However, this approach does consider intentional excessive randomization and diversification as discussed in [17] [16].

D. Semantics-Driven Signatures

Research has attempted to address the static similarity problem by extracting semantics from the program's static information. Using only static analysis, control flow graphs, function call graphs, function signatures, and API sequences can be derived that glean insight into a sample's semantics.

Several proposals have extended the dependency extraction technique from a file's metadata to create API sequence alignments [45] [46] [47] [48]. This approach analyzes the order in which the dependencies are invoked to create a coarse grained semantics driven API call sequence. [49] takes this a step further with DLL Miner, which assesses similarity based on the longest common subsequence of DLLs. This approach finds similarity based on applications invoking the same DLLs in the same sequence order.

Other techniques proposed dissembling the object code to find all `call` instructions to create function call graphs [50]. BinSequence [51] implements a fuzzy matching technique for identifying whole function code reuse. This is effective for identifying reused code, but does not address the diversity issue within the function definitions. Further, C++ applications with name mangling and function overloading are not normalized. BinJuice [52] takes a similar approach by signaturing functions to extract the semantic "juice" of an application. [53] takes the call graphs approach a step further by combing call graph analysis with n-grams, similar to the bytewise approximate matching technique.

Upchurch et al. propose a sliding window n-gram approach to detecting code reuse patterns in a large malware corpus [30]. Their approach examines the first opcode in each n-gram block of object code to find similar blocks. It was shown to be effective for fast precision and recall, but this work does not address variations introduced by compilers.

CARDINAL [54] was proposed to address opcode diversity introduced by compilers. Instead of analyzing blocks of object code, they parse the compiled binary for function calls and use call parameter cardinality (CPC) as a similarity metric. They argue that the number of arguments to a function call does not change significantly between compiler generated diversity. However, they do not consider function overloading and calling convention diversity.

Some techniques look at complete control flow instead of just call graphs by looking for all variants of `call` and `jmp` instructions [55]. However, Delay loading, function inlining, function call order shuffling, and variations in equivalent function calls can thwart static control flow graph (CFG) analysis based on API extraction

Current research in this field appears to be leaning towards machine learning approaches that leverage both metadata, API sequences, and control graphs to classify malware and find similarity [56] [57]. It is our hope that future work takes into consideration the sensitivity of applications to all of the sources of code diversity.

IV. Conclusion

Malware code diversity presents a challenge for finding static similarities between semantically equivalent malware samples. Our taxonomy of code diversification sources illustrates that there are numerous sources in compiled code variations. This warrants further discussion and research into the efficiency and legitimacy of static similarity detection mechanisms.

Our discussion leads use to the following conclusions:

- need a compiler and architecture agnostic approach to finding similar code fragments;
- dependency analysis needs to be normalized to account for variations in linking methods;
- metadata from the compiled application's headers and sections is easily manipulated either manually or as a result of diversification;
- more testing is required for code normalization techniques against a large malware corpus;
- and control flow and function call analysis is vulnerable to diversification.

A thorough examination and experimentation on the effectiveness of popular static malware similarity techniques against explicitly diversified samples will be a novel endeavor. We can then examine import and export directories for similarity, similar to ImpHash [37]. Import-based similarities does have shortcomings that need to be addressed, such as library normalization, delay loaded libraries, and explicitly linked API calls at runtime. By examining imports and exports, we can build a coarse-grained control flow integrity graph without need for recompilation. We can further normalize library API calls and relocations to account for differences in versions and compilers.

We suggest that the work of [44] to normalize compiled binaries into an intermediate representation (IR) and performing similarity checks against the normalized IR instead of the compiled executable binary image is promising for malware similarity. Combine this with import dependencies that scan for delay loading, import normalization, and machine learning and we arguably have the best approach for overcoming diversity in malware similarity.

References

[1] "McAfee Labs Threats Report September 2018," p. 21, 2018. [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-sep-2018.pdf

[2] "WannaCry and NotPetya inspiring new attacks." [Online]. Available: https://www.computerweekly.com/news/252449265/WannaCry-and-NotPetya-inspiring-new-attacks

[3] M. Laliberte and I. S. T. A. a. W. Technologies, "Why hackers reuse malware," Nov. 2017. [Online]. Available: https://www.helpnetsecurity.com/2017/11/20/hackers-reuse-malware/

[4] M. Payer, S. Crane, P. Larsen, S. Brunthaler, R. Wartell, and M. Franz, "Similarity-based matching meets malware diversity," *arXiv preprint arXiv:1409.7760*, 2014.

[5] R. Abel. (2018) Uptick in malware designed to size up targets before launching full payload. [Online]. Available: https://www.scmagazine.com/home/news/uptick-in-malware-designed-to-size-up-targets-before-launching-full-payload/

[6] E. M. Hutchins, M. J. Cloppert, and R. M. Amin, "Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains," *Leading Issues in Information Warfare & Security Research*, vol. 1, no. 1, p. 80, 2011.

[7] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," *IEEE Security & Privacy*, vol. 5, no. 2, 2007.

[8] E. O. Osaghae, "Classifying Packed Programs as Malicious Software Detected," *Information Technology*, vol. 5, no. 6, p. 4, 2016.

[9] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

[10] J.-M. Borello and L. Mé, "Code obfuscation techniques for metamorphic viruses," *Journal in Computer Virology*, vol. 4, no. 3, pp. 211–220, 2008.

[11] M. Suenaga, "A museum of api obfuscation on win32," in *Proceedings of 12th Association of Anti-Virus Asia Researchers International Conference, AVAR*, vol. 2009, 2009.

[12] Y. Guillot and A. Gazet, "Automatic binary deobfuscation," *Journal in computer virology*, vol. 6, no. 3, pp. 261–276, 2010.

[13] S. K. Udupa, S. K. Debray, and M. Madou, "Deobfuscation: Reverse engineering obfuscated code," in *Reverse Engineering, 12th Working Conference on*. IEEE, 2005, pp. 10–pp.

[14] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *2015 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2015, pp. 674–691.

[15] J. Raber and E. Laspe, "Deobfuscator: An automated approach to the identification and removal of code obfuscation," in *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*. IEEE, 2007, pp. 275–276.

[16] M. Payer, "Embracing the new threat: Towards automatically self-diversifying malware," in *The Symposium on Security for Asia Network*, 2014.

[17] P. Larsen, S. Brunthaler, L. Davi, A.-R. Sadeghi, and M. Franz, "Automated software diversity," *Synthesis Lectures on Information Security, Privacy, & Trust*, vol. 10, no. 2, pp. 1–88, 2015.

[18] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Automated software diversity," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014.

[19] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi, "Microsoft Malware Classification Challenge," *arXiv:1802.10135 [cs]*, Feb. 2018, arXiv: 1802.10135.

[20] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 763–780.

[21] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 574–588.

[22] L. L. N. L. (LLNL). (2018) Rose compiler. [Online]. Available: http://rosecompiler.org/

[23] A. Walenstein and A. Lakhotia, "The Software Similarity Problem in Malware Analysis," p. 10.

[24] F. Breitinger, B. Guttman, M. McCarrin, V. Roussev, and D. White, "Approximate matching : definition and terminology," National Institute of Standards and Technology, Tech. Rep. NIST SP 800-168, May 2014. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-168.pdf

[25] V. Harichandran, F. Breitinger, and I. Baggili, "Bytewise Approximate Matching: The Good, The Bad, and The Unknown," *Journal of Digital Forensics, Security and Law*, 2016.

[26] N. Sarantinos, C. Benzaid, O. Arabiat, and A. Al-Nemrat, "Forensic Malware Analysis: The Value of Fuzzy Hashing Algorithms in Identifying Similarities," in *2016 IEEE Trustcom/BigDataSE/ISPA*. Tianjin, China: IEEE, Aug. 2016, pp. 1782–1787.

[27] F. Pagani, M. Dell'Amico, and D. Balzarotti, "Beyond Precision and Recall: Understanding Uses (and Misuses) of Similarity Hashes in Binary Analysis," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy - CODASPY '18*. Tempe, AZ, USA: ACM Press, 2018, pp. 354–365.

[28] E. Raff and C. Nicholas, "Lempel-Ziv Jaccard Distance, an effective alternative to ssdeep and sdhash," *Digital Investigation*, vol. 24, pp. 34–49, Mar. 2018.

[29] V. Roussev, "An evaluation of forensic similarity hashes," *Digital Investigation*, vol. 8, pp. S34–S41, Aug. 2011.

[30] J. Upchurch and X. Zhou, "Malware provenance: Code reuse detection in malicious software at scale," in *Malicious and Unwanted Software (MALWARE), 2016 11th International Conference on*. IEEE, 2016, pp. 1–9.

[31] G. Wicherski, "peHash: A Novel Approach to Fast Malware Clustering," p. 8.

[32] M. Z. Shafiq, S. M. Tabish, F. Mirza, and M. Farooq, "PE-Miner: Mining Structural Information to Detect Malicious Executables in Realtime," in *Recent Advances in Intrusion Detection*, E. Kirda, S. Jha, and D. Balzarotti, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, vol. 5758, pp. 121–141.

[33] J. Choi, H. Kim, J. Choi, and J. Song, "A malware classification method based on generic malware information," in *International Conference on Neural Information Processing*. Springer, 2015, pp. 329–336.

[34] K. Raman, "Selecting Features to Classify Malware," p. 5.

[35] G. D. Webster, B. Kolosnjaji, C. von Pentz, J. Kirsch, Z. D. Hanif, A. Zarras, and C. Eckert, "Finding the Needle: A Study of the PE32 Rich Header and Respective Malware Triage," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, M. Polychronakis and M. Meier, Eds. Cham: Springer International Publishing, 2017, pp. 119–138.

[36] J. Choi, Y. Han, S.-j. Cho, H. Yoo, J. Woo, M. Park, Y. Song, and L. Chung, "A Static Birthmark for MS Windows Applications Using Import Address Table," in *2013 Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*. Taichung, Taiwan: IEEE, Jul. 2013, pp. 129–134.

[37] Mandiant, *Tracking Malware with Import Hashing*, 2014, https://www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html.

[38] A. Srivastava, A. Lanzi, and J. Giffin, "System call api obfuscation," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2008, pp. 421–422.

[39] P. O'Kane, S. Sezer, and K. McLaughlin, "Obfuscation: The hidden malware," *IEEE Security & Privacy*, vol. 9, no. 5, pp. 41–47, 2011.

[40] C. Cifuentes and D. Simon, "Procedure abstraction recovery from binary code," in *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*. Zurich, Switzerland: IEEE Comput. Soc, 2000, pp. 55–64.

[41] D. Bruschi, L. Martignoni, and M. Monga, "Code Normalization for Self-Mutating Malware," *IEEE Security and Privacy Magazine*, vol. 5, no. 2, pp. 46–54, Mar. 2007.

[42] M. D. Preda, I. Mastroeni, and R. Giacobazzi, "Analyzing program dependencies for malware detection," in *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014 - PPREW'14*. San Diego, CA, USA: ACM Press, 2014, pp. 1–7.

[43] N. Hasabnis and R. Sekar, "Lifting Assembly to Intermediate Representation: A Novel Approach Leveraging Compilers," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '16*. Atlanta, Georgia, USA: ACM Press, 2016, pp. 311–324.

[44] Y. David, N. Partush, and E. Yahav, "Similarity of binaries through re-optimization," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2017*. Barcelona, Spain: ACM Press, 2017, pp. 79–94.

[45] I. K. Cho, T. Kim, Y. J. Shim, H. Park, B. Choi, and E. G. Im, "Malware Similarity Analysis using API Sequence Alignments," p. 12.

[46] K. Iwamoto and K. Wasaki, "Malware classification based on extracted API sequences using static analysis," in *Proceedings of the Asian Internet Engineeering Conference on - AINTEC '12*. Bangkok, Thailand: ACM Press, 2012, pp. 31–38.

[47] M. Alazab, S. Venkataraman, and P. Watters, "Towards Understanding Malware Behaviour by the Extraction of API Calls," in *2010 Second Cybercrime and Trustworthy Computing Workshop*. Ballarat, Australia: IEEE, Jul. 2010, pp. 52–59.

[48] ——, "Towards understanding malware behaviour by the extraction of api calls," in *Cybercrime and Trustworthy Computing Workshop (CTC), 2010 Second*. IEEE, 2010, pp. 52–59.

[49] M. Narouei, M. Ahmadi, G. Giacinto, H. Takabi, and A. Sami, "DLLMiner: structural mining for malware detection: DLLMiner: structural mining for malware detection," *Security and Communication Networks*, vol. 8, no. 18, pp. 3311–3322, Dec. 2015.

[50] M. Hassen and P. K. Chan, "Scalable Function Call Graph-based Malware Classification," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy - CODASPY '17*. Scottsdale, Arizona, USA: ACM Press, 2017, pp. 239–248.

[51] H. Huang, A. M. Youssef, and M. Debbabi, "BinSequence: Fast, Accurate and Scalable Binary Code Reuse Detection," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications*

Security - ASIA CCS '17. Abu Dhabi, United Arab Emirates: ACM Press, 2017, pp. 155–166.

[52] A. Lakhotia, M. D. Preda, and R. Giacobazzi, "Fast location of similar code fragments using semantic 'juice'," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop on - PPREW '13*. Rome, Italy: ACM Press, 2013, pp. 1–6.

[53] Y. R. Lee, B. Kang, and E. G. Im, "Function matching-based binary-level software similarity calculation," in *Proceedings of the 2013 Research in Adaptive and Convergent Systems on - RACS '13*. Montreal, Quebec, Canada: ACM Press, 2013, pp. 322–327.

[54] L. Jones, A. Sellers, and M. Carlisle, "Cardinal: similarity analysis to defeat malware compiler variations," in *Malicious and Unwanted Software (MALWARE), 2016 11th International Conference on*. IEEE, 2016, pp. 1–8.

[55] M. Bourquin, A. King, and E. Robbins, "BinSlayer: accurate comparison of binary executables," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop on - PPREW '13*. Rome, Italy: ACM Press, 2013, pp. 1–10.

[56] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto, "Novel Feature Extraction, Selection and Fusion for Effective Malware Family Classification," in *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy - CODASPY '16*. New Orleans, Louisiana, USA: ACM Press, 2016, pp. 183–194.

[57] M. Gong, U. Girkar, and B. Xie, "Classifying Windows Malware with Static Analysis," p. 7.