CARDINAL: Similarity Analysis to Defeat Malware Compiler Variations

Luke Jones, Andrew Sellers
Academy Center for Cyberspace Research
US Air Force Academy
{luke.jones.ctr, andrew.sellers}@usafa.edu

Martin Carlisle
Information Networking Institute
Carnegie Mellon University
carlislem@cmu.edu

Abstract

Authors of malicious software, or malware, have a plethora of options when deciding how to protect their code from network defenders and malware analysts. For many static analyses, malware authors do not even need sophisticated obfuscation techniques to bypass detection, simply compiling with different flags or with a different compiler will suffice.

We propose a new static analysis called CARDINAL that is tolerant of the differences in binaries introduced by compiling the same source code with different flags or with different compilers. We accomplished this goal by finding an invariant between these differences. The effective invariant we found is the number of arguments to a call, or call parameter cardinality (CPC). We concatenate all CPC's together per function and input these chains into a Bloom filter.

Signatures constructed in this manner can be quickly compared to each other using a Jaccard index to obtain a similarity score. We empirically tested our algorithm on a large corpus of transformed programs and found that using a threshold value of 0.15 for determining a positive or negative match yielded results with a 11% false negative rate and a 11% false positive rate. Overall, we both demonstrate that CPC's are a telling feature that can increase the efficacy of static malware analyses and point the way forward in static analyses.

1. Introduction

For some years now, malware authors have been flooding the Internet with increasingly large amounts of malware. At this juncture, nearly 300 million new samples were found just last year in 2015 [6]. Much of this malware is quickly made variants made to fool simple signature methods. Once an anti-virus (AV) vendor catches a sample, they then immediately flag any present or future samples exactly like it. Commonly, virus writers use metamorphism to thwart this kind of AV protection, but in most cases, just switching compilers or even changing compile flags with the same compiler is enough to avoid static analysis detection.

So why not switch to dynamic analysis, which negates many obfuscation techniques (as in [1])? We pursue static analysis for the sake of avoiding the overhead of executing code [11, 7] and for the specificity of similar sample identification [8]. Though we make no assertions as to which approach is fundamentally better because each has merits and limitations, as evidenced by the march of progress of both static [14, 7, 10] and dynamic [12, 13, 4] analyses. However, if a static analysis could somehow account for program transformation differences (one of dynamic analyses greatest strengths) then a single analysis could leverage the strengths of both categories. In this work, we create a static analysis that is proven to account for a subset of program transformations, namely variations in compiler configuration.

The program transformation comparisons we consider are the following: where $\mathcal{A}^{\mathcal{X}}$ and $\mathcal{A}^{\mathcal{Y}}$ are the same compiler with different flags passed (i.e. isocompiler modulation where $\mathcal{X}$ and $\mathcal{Y}$ describe distinct, replicable compiler configurations) and $\mathcal{B}^{\mathcal{S}}$ and $\mathcal{C}^{\mathcal{T}}$ are different compilers such that $\mathcal{S}$ and $\mathcal{T}$ encode compiler configurations for compilers $\mathcal{B}$ and $\mathcal{C}$, respectively. Our goal is to be able to consistently determine that $\mathcal{M}^{\mathcal{X}}_{\mathcal{A}}$, $\mathcal{M}^{\mathcal{Y}}_{\mathcal{A}}$, $\mathcal{M}^{\mathcal{S}}_{\mathcal{B}}$, $\mathcal{M}^{\mathcal{T}}_{\mathcal{C}}$ are the same. Our tool is right about this determination 89% of the time. We are the first to test recovering similarity from these types of

List 1: Transformations

1. $\mathcal{M} \rightarrow \mathcal{A}^{\mathcal{X}} \Rightarrow \mathcal{M}^{\mathcal{X}}_{\mathcal{A}}$
   $\mathcal{M} \rightarrow \mathcal{A}^{\mathcal{Y}} \Rightarrow \mathcal{M}^{\mathcal{Y}}_{\mathcal{A}}$
2. $\mathcal{M} \rightarrow \mathcal{B}^{\mathcal{S}} \Rightarrow \mathcal{M}^{\mathcal{S}}_{\mathcal{B}}$
   $\mathcal{M} \rightarrow \mathcal{C}^{\mathcal{T}} \Rightarrow \mathcal{M}^{\mathcal{T}}_{\mathcal{C}}$

transformations.

We focus on call parameter cardinality (CPC)'s and propose CPC Aggregation by Reversing and Dumping in Arrays Lightweight (CARDINAL) which is an IDAPython module and some accompanying Python modules. Our critical insight for this work was that though many aspects of code and syntax change between different compilers and even optimization levels, the number of arguments specified by each function, or CPC's, remain invariant across compilers and flags. Even better, CPC is not a common target of obfuscation right now and naive attempts at obfuscation by adding extraneous arguments to a function might be optimized out by the compiler.

CARDINAL first finds every CPC. All CPC's in the same function are grouped together into a CPC chain. Then, every chain is dumped into a Bloom filter. This is CARDINAL's signature for an executable. We chose Bloom filters over other possible data structures for their ease of comparison. Two signatures are compared using a Jaccard index between the bit arrays of their Bloom filters. This yields a number between 1 and 0, 1 being exactly identical and 0 meaning there are no identical CPC chains.

To test the utility of CARDINAL, we created a corpus of binaries compiled in multiple different ways. This corpus is freely available for download. We compiled with Clang and GCC, and for both compilers, we compiled at the O0, O1, O2 and O3 optimization levels. Our sources were composed of 20 programs from the LLVM test-suite and 70 from GitHub. We largely used C programs, but there are a few C++ programs as well. We tested CARDINAL to see how well it could identify binaries made from the same source, regardless of how they had been compiled. The next paragraph briefly discusses our results.

Contributions. In summary, this paper embodies these contributions to malware similarity analysis:

1. *Call parameter cardinality.* We show the potential of this feature to accurately inform static analyses with 89% accuracy rate even when evaluating code clones that have been compiled using different compilers or with different compiler flags.

2. *CARDINAL.* We implement a working open-source prototype that can accurately recover CPC, or the number of arguments, from about 95% of disassembled functions. The code is available here: <url redacted for peer review>

3. *Compiler variation corpora.* Available for download are the corpora we made to test our tool which includes 90+ sources compiled with two different compilers and four different optimization levels. It is available at this location: <url redacted for peer review>

2. Our Approach

CARDINAL is different from any previous work because it constructs a signature out of the number of arguments to a call. Of utmost importance to our algorithms is the accuracy of the cardinality of these recovered arguments, and critical to accuracy is correct understanding of calling conventions.

2.1. Calling Convention

The calling convention of a compiled program establishes a common agreement between caller and callee on passing arguments, saving registers, and managing the stack to enable its usage for the callee and consistency for the caller. We implement CARDINAL for 64-bit Linux systems, so our algorithms assume the System V AMD64 ABI calling convention. This convention uses **rdi**, **rsi**, **rdx**, **rcx** (or **r10** for system calls), **r8**, **r9**, and all the corresponding registers of different width (**edi**, **r8d**, etc.) for integer/-pointer arguments, and **xmm0** through **xmm7** for floating point arguments. We call these registers the argument registers for the remainder of the paper. Their order is very specific. **rdi** is the first argument, **rsi** is the second, etc. With floating point arguments, **xmm0** is first, **xmm1** is second, and so on. Any additional arguments are pushed on the stack.

2.2. Assembly Case Studies

2.2.1 Silent Arguments

The silent argument case is when a compiler passes arguments that are already in the correct registers to a function's first call, so the compiler adds no code, leaving the argument passing "silent" at the callsite. This posed problems. Take for example Listing 1, taken from the "lua" test of the LLVM test-suite compiled with the O1 flag set.

Listing 1: Silent Argument Example

```
405786 mov  esi , 0FFFFFFFFh
40578B mov  rdi , rbx
40578E call lua_isstring
...
(lua_isstring)
402560 push rax
402561 call lua_type
402566 add  eax , 0FFFFFFFDh
402569 cmp  eax , 2
40256C sbb  eax , eax
40256E and  eax , 1
402571 pop  rdx
402572 retn
```

```
...
(lua_type)
4024D0 push rax
4024D1 call index2adr
...
(index2adr)
4022F0 test   esi, esi
4022F2 jle    short loc_402311
4022F4 dec    esi
4022F6 movsxd rax, esi
4022F9 shl    rax, 4
4022FD add    rax, [rdi+18h]
```

At 405786 and 40578B, two arguments are passed to lua_isstring. However, throughout the entirety of lua_isstring, we do not see these arguments in any code. It is not until we follow both lua_type and index2adr that we find **rdi** and **rsi** being used at lines 4022FD and 4022F0, respectively. This means that **rdi** and **rsi** were passed down through lua_istring and lua_type implicitly, or silently. In Section 2.3.3 we explain how our callee analysis accounts for silent arguments.

2.2.2 Unused Arguments

In unoptimized versions of code, arguments unused by a function will be saved onto the stack regardless if they are used, and thus have some associated code. However, once code is optimized, unused arguments disappear from the callee's disassembly. Listing 2 from "ldecod" from the LLVM test-suite compiled with the O3 flag set demonstrates this.

Listing 2: Unused Argument Example

```
41C8AD mov rcx, cs:erc_errorVar
41C8B4 xor edi, edi
41C8B6 xor edx, edx
41C8B8 xor esi, esi
41C8BA call ercStartSegment
...
(ercStartSegment)
40D040 test   rcx, rcx
40D043 jz     short 40D06B
40D045 mov    eax, [rcx+40h]
40D048 test   eax, eax
40D050 movsxd rsi, esi
40D05A lea    rdx, [rsi+rsi*2]
40D069 mov    [rax], edi
40D06B rep retn
```

At the beginning of this listing, four arguments are passed to ercStartSegment, through the appropriate registers. In ercStartSegment, we can see **rdi** used at 40D069, **rsi** at 40D050, and **rcx** used at 40D040. However, there is no usage of **rdx**. **rdx** is a register that is passed to ercStartSegment, but any code referencing it is optimized away, leaving it as a completely unused argument register. Section 2.3.5 discusses our heuristics that handle unused arguments.

2.3. Algorithms

We built the heart of CARDINAL on top of the popular disassembler, IDA Pro. The two major algorithms in our IDA Pro analysis are the caller sweep and the callee sweep. These two algorithms exist because there are certain cases when determining the number of arguments is more accurate at the callee and some where it is more accurate at the caller. There are also cases when the callee is not available such as with an indirect call or library call, but, on the other hand, caller analysis has a larger scope than callee analysis. In callee analysis, the arguments passed are usually used before the first call instruction which makes them easily found.

2.3.1 Caller Sweep Overview

To recover the CPC's in a program, we start up IDA Pro and disassemble the program. We start with the caller sweep, and mark the name and starting address of every function so that our subsequent analysis can respect function boundaries. The caller sweep walks through every instruction in the code section, doing three main things:

1. Making a list of any targets of calls or jumps to functions. These will be resolved later to the CPC of the given function when CPC aggregation as described in Section 2.3.5 happens.
2. Starting a new callee sweep at the target of any call or jump to a function.
3. Maintaining a caller context of what argument registers are currently being used as parameters at a callsite.

CARDINAL mainly tells if an argument register is being used as a parameter by doing a dataflow analysis to determine what argument registers reach a callsite after being set but without being used prior to the call. We leverage heuristic reasoning to differentiate between argument registers used as parameters and those simply set near a call. CARDINAL also resets the caller context after a fixed number of instructions without seeing an argument register set. This lets the analysis forget about irrelevant argument registers set a long time ago that likely have nothing to do with a callsite. We decided on a window of 15 instructions without an argument register set to reset the caller context. This essentially keeps the caller sweep dataflow analysis fresh.

With the dataflow analysis guided by the above heuristics, the caller sweep determines a CPC for each callsite it encounters. Since there might be multiple callsites to the same function, the caller sweep builds a list of each CPC it calculates for a given function. Later, when CARDINAL is resolving each target from the list of called addresses, it will only consider the caller value if there is above a certain threshold of agreement on at least one CPC of a given function. For our experiments, we decided on a threshold of 0.75. So at least three out of four caller sweep CPC's have to be the same, otherwise CARDINAL will default to the callee sweep determined CPC for the final accepted value of the target. 0.75 was chosen by testing some other threshold values empirically and finding that 0.75 had the best accuracy.

2.3.2 Caller Sweep Algorithm

Algorithm 1 is a formalization of the steps discussed in Section 2.3.1. It requires the following constructs:

- CALLEE_SWEEP($addr$,$next\_func$,n) : starts Algorithm 2 at $addr$. Returns callee context.
- IS_FUNC_CALL($inst$) : checks if instruction is call or jump to a function in the code section.
- cxt : context, an object that holds what registers have been set and not used.
- cxt.update($inst$) : updates context by clearing any set registers that have now been used, and by adding any new set registers.
- callee_dict : dictionary of callee to callee context.
- caller_dict : dictionary of callee to list of caller contexts.

Of particular note is how the algorithm builds dictionaries for contexts discovered by both caller argument analysis and callee argument analysis. Later, the CPC can easily be calculated from the stored contexts. On line 7, the algorithm inserts a separator between target addresses for every function boundary. This creates an address chain composed of every called address in a function. This is to offset the actual construction of a CPC chain aggregate because the caller analysis CPC's are not complete until the entire disassembly has been analyzed. The algorithm in Section 2.3.5 creates the aggregate chain of CPC's by deciding between caller or callee for each called function in addr_chain.

Algorithm 1 checks that the target of the call instruction is not recursive at line 20. Recursive callsites are poor indicators of CPC since their arguments can easily be silently passed like the example in Section 2.2.1. We therefore rely on either callee analysis or non-recursive callsites to determine these functions' CPC's.

Algorithm 1 Sweep Caller for CPC's and Callees

```
1: CALLER_CONTEXT_REFRESH ⇐ 15
2: $funcs$ ⇐ FUNC_BEGIN() {list of starting addresses}
3: $f$ ⇐ 0
4: $h$ ⇐ 0
   {Iterates over every instruction}
5: for all $inst$ in disassembled instructions do
6:   if addr($inst$) >= $funcs[f]$ then
7:     addr_chain.append(sep) {sep is newline character}
8:     cxt.reset()
9:     $f$ ⇐ $f$ + 1
10:  end if
11:  if $h$ > CALLER_CONTEXT_REFRESH then
12:    cxt.reset()
13:    $h$ ⇐ 0
14:  end if
15:  if IS_FUNC_CALL($inst$) then
16:    if target($inst$) not in callee_dict then
17:      $next\_func\_head$ ⇐ NEXT_FUNC($inst$)
18:      callee_dict ⇐ (target($inst$),
            CALLEE_SWEEP( target($inst$),
            $next\_func\_head$, 0))
19:    end if
       {Is not recursive call}
20:    if target($inst$) != $funcs[f$-1] then
21:      caller_dict[target($inst$)].append(cxt)
22:      cxt.reset()
23:    end if
24:    addr_chain.append(target($inst$))
25:  end if
26:  if cxt.update($inst$) then
27:    $h$ ⇐ 0
28:  else
29:    $h$ ⇐ $h$ + 1
30:  end if
31: end for
```

2.3.3 Callee Sweep Overview

The callee sweep works similarly to the caller sweep, although with an inverted perspective. Namely, instead of looking for argument registers that have been set without being used before a callsite, we instead look for argument registers that have been used without being set after execution follows a call. If the register is used without being set, we know that the argument register has been supplied from an external source. We call these argument registers "source." The callee analysis accomplishes three things:

1. Maintaining a callee context that accurately reflects the state of source argument registers.
2. Finding silently passed arguments by recursing down

the first call of a function.

3. Finding stack arguments by textual analysis of used values from the stack.

The callee sweep deals with silent arguments, as discussed in Section 2.2.1, by recursing down its first call. Silent arguments are also only pursued at up to five calls deep to avoid resource over-consumption. The first child analysis to either hit the max recursion depth, or hit the end of its analysis without finding a call passes its context to its parent callee sweep. The parent adds whatever source the child has to its own context, because a source argument register that a child has must come from its parent. However, if the child uses all fourteen argument registers, we do not add those to the parent context because that is usually an artifact of a varargs implementation. Passing every single argument up to the callee analysis would sabotage this heuristic detection of silent arguments, but thanks to the rarity of actually using all fourteen argument registers, we can just ignore the child context whenever it reports all registers being used. Ultimately this algorithm works because although an argument might be passed silently once or twice, eventually a function uses it plainly.

Finding stack arguments is not difficult because IDA Pro conveniently labels them as "arg_x" where x is some offset. The callee analysis simply textually searches for the "arg_" string in stack-based source operands. When CPC is calculated, the number of unique stack arguments stored in the context is added to the cardinality.

2.3.4 Callee Sweep Algorithm

Algorithm 2 is a straight-forward implementation of the previous discussion in Section 2.3.3. It has the following parameters:

- $inst\_start$: where to start analysis of disassembly.
- $next\_func$: starting address of next sequential function in disassembly.
- n : current recursion depth.

The algorithm has the following requirements:

- cxt : context, an object that holds which registers are source and which are set.
- cxt.add_cxt($child\_cxt$) : Adds source registers from $child\_cxt$ to cxt as long as they are not already set in cxt.
- cxt.add_stack_args($inst$) : adds the correct number of stack arguments to the context.

Algorithm 2 Sweep Callee for CPC

```
1:  MAX_DEPTH ⇐ 4
2:  MAX_REG_ARGS ⇐ 14 {Max number of argument registers}
    {Iterate over every instruction starting at passed in address}
3:  for all $inst$ starting at $inst\_start$ do
4:    if $inst$ >= $next\_func$ then
5:      break
6:    end if
7:    if IS_FUNC_REF($inst$) then
8:      if $n$ < MAX_DEPTH then
9:        $child\_cxt$ ⇐ callee_dict[target($inst$)]
10:       if $child\_cxt$ is None then
11:         $next\_func\_inst$ ⇐ NEXT_FUNC($inst$)
12:         $child\_cxt$ ⇐ CALLEE_SWEEP(
              target($inst$), $next\_func\_inst$, $n$+1)
13:         callee_dict ⇐ (target($inst$),$child\_cxt$)
14:       end if
15:       if $child\_cxt$.calc_cac() < MAX_REG_ARGS then
16:         cxt.add_cxt(callee_dict[target($inst$)])
17:       end if
18:     end if
19:     break {Stop callee analysis after first call}
20:   end if
21:   if "arg_" in $inst$ then
22:     cxt.add_stack_args($inst$)
23:   end if
24:   cxt.update($inst$)
25: end for
26: return cxt
```

- cxt.update($inst$) : updates context with source and set registers.

First, at line 4, Algorithm 2 checks that its disassembly is not in the next function. Detecting function boundaries is very important because we do not want to conflate two functions' argument registers and get incorrect CPC's. IDA has good function boundary detection built-in and CARDINAL takes advantage of that. At line 15 we check for the child context using all argument registers which as discussed before is a sign of a function using varargs, and we safely ignore child contexts with this symptom. Lastly, the break at line 19 ensures that Algorithm 2 stops analysis at the first function call it encounters. Note that after a function call, a floating point return value can be passed out through the **xmm0** register, thereby used without being set even though it was set by the called function and not by the caller.

2.3.5 Aggregate Construction

First, CARDINAL makes a list of all the CPC's that the caller analysis believes the function at $addr$ might be based on its previous multiple callsite dataflow analysis. Next it finds the highest frequency CPC and calculates what percentage of all CPC's this particular value constitutes. Next, CARDINAL ensures that the actual majority percentage is greater than or equal to the threshold percentage. If it is, CARDINAL selects it for the caller CPC. On the callee side, first the algorithm checks if all argument registers have been used and invalidates the callee CPC contribution if they have. The case for a function using all 14 argument registers is so rare, that this heuristic increases the accuracy of our analyses appreciably by ignoring erroneous values for varargs functions. Finally, we pick the larger of either the caller or callee CPC for our CPC dictionary value. We pick the larger because both caller and callee analyses err on the side of underestimating. Especially for the unused argument case, the correct caller value will be larger than the callee. Lastly, we construct the CPC chain aggregate by looking up in the dictionary the CPC that CARDINAL selected for each function address. CARDINAL maintains separate chains by printing any separators found in the $addr\_chain$ from Algorithm 1 into the CPC chain aggregate. We can input the completed CPC chain aggregate into a Bloom filter for comparing different binaries.

2.3.6 Similarity Metrics

To assess similarity, we use a Jaccard index, and to calculate this, we use the underlying bit array to calculate set intersections and unions between two different fingerprints of two different binaries. As previously proved [9], the bitwise "or" and "and" of a signature approximates the Jaccard index faithfully. This similarity metric handles reordered functions very well, since the introduction into a Bloom filter ignores any notion of location. However, the feature hashing involved in a Bloom filter means that there is no allowance for close matches, only exact matches. CPC chains that are the same except for an extra call, or slightly different CPC will be treated as completely different. It would be preferable if this metric had some allowance for near matches. The compiler will often inline a call or unroll a loop which changes the CPC chain, sometimes minimally enough that just having a softer metric would handle the issue and still report high similarity between two binaries made from the same source.

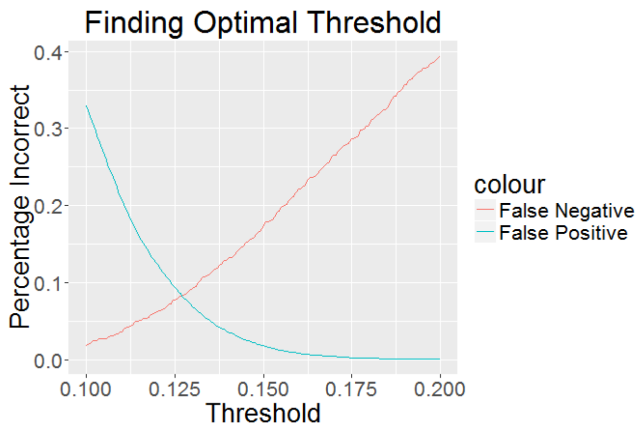

Figure 1

3. Evaluation

3.1. CPC Recovery Accuracy

As mentioned previously, accurately recovering CPC's critically affects the efficacy of CARDINAL as a whole. Therefore, we calculated this accuracy by finding ground truth on a subset of our test set and comparing it to CARDINAL's report. We compiled and linked the test modules into a single LLVM IR file and ran a pass we wrote that found how many arguments each function required. We then ran CARDINAL in a mode that just reported the number of arguments it calculated each function to have. Lastly, we compared ground truth and CARDINAL's results and calculated a percentage. Each test had 85 to 600 functions, and since we had to manually compile and link and LLVM IR module for each test, we only tested eight of our programs. We found that CARDINAL on average was 95% accurate in finding the correct CPC for a function. This proved accurate enough to yield good results in our binary similarity comparison.

3.2. Transformation Tolerance

We applied CARDINAL to our entire corpus to see how correctly it could detect binaries made from the same source but altered with the transformations described in List 1. This did not result in a perfect delineation of related and unrelated binaries, we therefore chose a threshold value above which we concluded that two binaries were strongly related and below which we concluded that two binaries were unrelated. We found that drawing the line at 15% of Bloom filter CPC chains matching balanced the number of false positives and false negatives which is illustrated by Figure 1.

Overall, using a threshold of 15%, CARDINAL misidentified 11% of strongly related binaries as unrelated and 11%

of unrelated binaries as strongly related. The tests with strongly related binaries were the isocompiler modulation and different compiler tests. The unrelated binaries test was the different source tests. With these false positive and false negative rates, CARDINAL has a 89% accuracy rate overall. Next, we look at some of the results in detail, per test.

For the isocompiler modulation test, we compared our test binaries that were compiled with the same compiler, but with different levels of optimization. Every optimization level was compared to every other optimization level.

Both Clang and GCC have similar outcomes. CARDINAL is almost always right when comparing O0 to O1 or O2 to O3. It is often right for O1 to O2 and O1 to O3. We get most false negatives in the O0 to O2 and O0 to O3 category, however, the preponderance of comparisons even in those categories yield the correct result. In four of the six categories, the heaviest cluster of data points are clear of the threshold. Only in O0-O2 and O0-O3 does the heaviest cluster include the threshold, though still the majority of the cluster is above the threshold. We conclude that CARDINAL on the whole is not fooled by tweaking optimization levels.

For the different compiler test, we compared test binaries compiled with the same optimization flag but a different compiler. We tested Clang and GCC.

The different compilers test turned out better than the isocompiler modulation test. The majority of tests are easily above the threshold, and even all the heavy clusters are above the threshold. These results show that CARDINAL can tell that two sources are the same, even when the binaries are compiled with different compilers. This type of differentiation is a novel contribution to the malware similarity analysis field.

However, does CARDINAL faithfully report when two binaries have different sources? We took all of our binaries and compared them against each other to find out. This was our different source test.

CARDINAL is not perfect, but the weight of the cluster of tests fall well below the threshold. CARDINAL misclassifies unrelated binaries as strongly related only 11% of the time. From this data we know that our good results with the isocompiler modulation and different compiler tests are not the product of an overly greedy algorithm.

4. Related Works

As far as malware similarity, CARDINAL falls under the detecting strong relationship between binaries category, and though there are many research papers in this niche, the question that CARDINAL answers is unique. Our question is: can we negate the differences introduced by compiling the same source with different compilers, and compiling the same source with different sets of flags? This has not been investigated before, so the related works presented here cannot be directly compared to CARDINAL, even though we have borrowed various ideas from them.

The Basic Block Comparison Platform (BBCP) [2], increased the accuracy of similarity measure of BitShred [9] by putting sliding windows of normalized assembly instructions into a Bloom filter instead of sliding windows of machine code. CARDINAL borrows the idea of using Bloom filters for efficient set membership testing of features. CARDINAL uses CPC chains as features instead of normalized instructions. BBCP also mapped the way for CARDINAL by first comparing binaries made from the same source but different compilers, however they did not use BBCP on binaries compiled from different sources so the result of their different compilers test is unclear. BBCP and CARDINAL have similar goals of comparing two binaries and returning a similarity score, however BBCP was tested on binaries compiled from subsets of the same source while CARDINAL was tested on binaries compiled from the same source but with variations on the compiler.

Chen et al. [3] investigate how optimization levels affect similarity scores derived from the number of similar functions reported by three IDA-based tools. They found that optimization levels did in fact affect the similarity score, though they did not test how their similarity tests did against binaries from different sources so the severity of the effect of optimization on their methods is unclear.

Choi et al. [5] conduct software similarity research, not malware similarity research specifically, but they implement a similar method as CARDINAL. Their technique uses IDA Pro to disassemble an executable and identify API calls. They calculate each function's set of possible API calls. They find the similarity between two functions by comparing differences in API call sets and they compare whole programs by a matching problem that pairs the most similar functions together. Overall, both the work of Choi et al. and CARDINAL measure the similarity of two binaries by comparing them per function and looking at calls. The difference being CARDINAL looks at number of arguments to non-library calls and the work of Choi et al. looks at Windows API calls specifically. The work would be comparable to ours, except for the divergence in testing methodology. Choi et al. use a small corpus of less than twenty unique sources for most of their tests, and they test the similarity between different versions of the same software, instead of the effects of compiler variations on the same exact software. They have a different compilers test, but the compilers differ in version only.

MetaAware [14] is a static analysis tool that seeks to minimize the effects of metamorphism and program transformation on similarity scores. The authors choose to do this by creating patterns out of a code fragment's library and system calls. They perform data and control flow analysis to make this as accurate as possible. They get good results for their test sets including the SPEC CPU2000 benchmarks transformed with a research obfuscation tool, viruses and their variants from VX Heaven, and various versions of binutils. Their goal is the same as ours, to maintain a high similarity score even with code transformations, however they test against different code transformations than we do, electing for a research obfuscation technique, metamorphic malware variants and software lineage tests.

5. Future Work

One undesirable property of Bloom filters is that they report matches only on exact matches. This creates problems when different optimization levels inline functions and unroll loops. If we could find a data structure or metric that allowed for slight differences between CPC chains, our results may improve. Fuzzy hashing or sequence alignment algorithms should be tried in the future to see if they helpfully ignore small differences in CPC chains without increasing false positives too much.

6. Conclusion

Though malware authors continue to inundate the internet with variations on the same binary, researchers are inventing new ways to detect this malware. One of these new ways is CARDINAL, which aims to negate code transformations and starts with negating the difference between binaries when compiled with different optimization flags or different compilers. We thought that if we could negate these differences, we would have a good base upon which to build and later negate other code transformations such as obfuscation. The main insight with CARDINAL was that the number of arguments passed to a function was invariant across optimization levels and compilers, and so building a signature out of these cardinalities would be more resilient to code transformation than other options.

This insight proved to be viable and useful, letting us bring the following contributions to the malware similarity field: 1. Call parameter cardinality as an accurate, robust feature informing static analyses; 2. our open-source implementation of *CARDINAL.* accurately recovers call parameter cardinality from a high number of compiled functions: and, 3. a public compiler variation corpora we made to test our tool that includes 90+ sources compiled with two different compilers and four different optimization levels.

In conclusion, we have shown previous related work and how their results are similar yet incomparable to ours for various reasons, some challenges in analyzing the assembly to recover the cardinality of arguments at a callsite and how we overcame them, and finally how CARDINAL proved to be a worthwhile tool with its 11% false positive and 11% false negative rate, yielding an 89% accuracy rate for transformations that until now had not been shown to be negated by modern malware similarity methods.

References

[1] R. Abielmona, R. Falcon, N. Zincir-Heywood, and H. Abbass. *Recent Advances in Computational Intelligence in Defense and Security*. Studies in Computational Intelligence. Springer International Publishing, 2016.

[2] F. Adkins, L. Jones, M. Carlisle, and J. Upchurch. Heuristic malware detection via basic block comparison. In *Malicious and Unwanted Software:" The Americas"(MALWARE), 2013 8th International Conference on*, pages 11–18. IEEE, 2013.

[3] H. Chen. The influences of compiler optimization on binary files similarity detection. In *2013 the International Conference on Education Technology and Information System (ICETIS 2013)*. Atlantis Press, 2013.

[4] I. K. Cho, T. Kim, Y. J. Shim, H. Park, B. Choi, and E. G. Im. Malware similarity analysis using api sequence alignments. *Journal of Internet Services and Information Security (JISIS)*, 4(4):103–114, 2014.

[5] S. Choi, H. Park, H.-I. Lim, and T. Han. A static birthmark of binary executables based on api call structure. In *Proceedings of the 12th Asian Computing Science Conference on Advances in Computer Science: Computer and Network Security*, ASIAN'07, pages 2–16, Berlin, Heidelberg, 2007. Springer-Verlag.

[6] A.-T. Institute. Malware.

[7] G. Jacob, P. M. Comparetti, M. Neugschwandtner, C. Kruegel, and G. Vigna. A static, packer-agnostic filter to detect similar malware samples. In *Detection of intrusions and Malware, and vulnerability assessment*, pages 102–122. Springer, 2012.

[8] G. Jacob, H. Debar, and E. Filiol. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in computer Virology*, 4(3):251–266, 2008.

[9] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: Fast, scalable malware triage. *Cylab, Carnegie Mellon University, Pittsburgh, PA, Technical Report CMU-Cylab-10*, 22, 2010.

[10] Y. Li, S. C. Sundaramurthy, A. G. Bardas, X. Ou, D. Caragea, X. Hu, and J. Jang. Experimental study of fuzzy hashing in malware clustering analysis. In *8th Workshop on Cyber Security Experimentation and Test (CSET 15)*, 2015.

[11] M. A. Siddiqui. *Data mining methods for malware detection*. ProQuest, 2008.

[12] G. Wagener, A. Dulaunoy, et al. Malware behaviour analysis. *Journal in computer virology*, 4(4):279–287, 2008.

[13] Y. Yi, Y. Lingyun, W. Rui, S. Purui, and F. Dengguo. Depsim: a dependency-based malware similarity comparison system. In *Information Security and Cryptology*, pages 503–522. Springer, 2010.

[14] Q. Zhang and D. S. Reeves. Metaaware: Identifying metamorphic malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 411–420. IEEE, 2007.