On the effectiveness of code normalization for function identification

Angelos Oikonomopoulos*, Remco Vermeulen*, Cristiano Giuffrida*, Herbert Bos*
*Vrije Universiteit Amsterdam

***Abstract*—Information on the identity of functions is typically removed when translating source code to executable form. Yet being able to recognize specific functions opens up a number of applications. In this paper, we investigate normalization-based approaches for the purposes of aiding the reverse engineer and as an enabler for the rejuvenation of legacy binaries. We iteratively refine our methods and report on their effectiveness. Our results show that a naive approach can be surprisingly effective in both problem domains. Further, our evaluation looks into more advanced normalization techniques and finds that their practicality varies significantly with the problem domain.**

I. Introduction

Program analysis at the binary level is exceedingly challenging: source information is often irreplaceable in reasoning about the properties of a source base. Yet source code is rarely available for malicious software (malware) samples or many commercial programs. Even in the rare case that the source for malicious software leaks, it is unlikely to be the corresponding source for a given sample, leaving the reverse engineer with the arduous task of analyzing a sample manually, with the help of tools such as clone detection software—painstakingly trying to match the binary code to specific functions.

Reverse engineering is a laborious process which is human-time-intensive. Succinctly summarizing the semantics of a piece of code is not possible in general. When it is, it usually concerns standard functions with well-defined semantics. Aside from being more reliable, any automatic identification of an already known function saves significant reverse engineering time, as a person never has to look at this part of the program.

Function identification is also important in production deployments where organizations are often left relying on outdated binaries for which the corresponding source is not known or was not available in the first place (as for commercial off-the-self software). Binary rejuvenation [1] refers to the act of revamping "stale" binaries to take advantage of modern features. For instance, IBM frequently adapts parts of decades-old binary-only software to modern hardware [2]. For a multitude of reasons—including static linking, embedding of custom "smart" implementations, as well as portability and deployment considerations—binaries often end up with copies of standard functions. Yet as such code gets older, these code fragments are often a poor fit for today's cache sizes and microarchitectures and likely fail to take advantage of new, specialized instructions. Worse, embedded versions might also suffer from correctness or reliability issues. Binary rewriters (such as DynInst [3]) can take on the task of replacing a function with a more appropriate counterpart, if only the function can be reliably identified in the first place.

Here, we consider function identification as the problem of declaring a machine code function to be *equivalent* to one of a given set of "known" functions, provided in source form. We call this set the *registry*. In both motivating applications we consider in this paper, reverse engineering and binary rejuvenation, we are only concerned with the *functionality* of the identified functions and not with their other (e.g., code size, performance or power consumption) characteristics. Therefore, we identify functions by *input/output equivalence*, which indicates that two functions compute exactly the same outputs for the same inputs with no additional side effects [4]. However, we do not try to identify completely unrelated reimplementations of a function, which would be undesirable for our target domain.

To make progress on this hard problem, we narrow our focus to code which offers the highest return on investment. For the purposes of reverse engineering, we look at utility functions, as "glue" or novel functions are unlikely to have semantics which can be summarized [5]. Moreover, with regard to binary rejuvenation, we focus on functions which are regularly the target of micro-optimization effort (for instance, primitives such as `memcpy()`, `strcpy()` and `crc32()`). The target sets of functions largely overlap, making for a more attractive set.

Needless to say, techniques which deduce software *similarity* (such as clone detectors at the binary level) are not sufficient for either purpose. Even reliable similarity detection does not eliminate the need for a reverse engineer to carefully examine the piece of code which looks like a known function. More worrying still, replacing a function with one which has, even subtly, different behavior, is likely to have catastrophic results.

In this paper, we propose to tackle function identification in this narrower context using two key insights. First, developing a new compiler or making significant changes to one is not a task undertaken lightly. As a result, we expect executables to have been compiled either by a released version of the popular compilers for the target platform or

by a closely related version (e.g., one slightly modified by a GNU/Linux distribution). Clearly, this assumption does not *always* hold as some code may have been generated by an arcane compiler that does not resemble one of the popular ones at all but, as argued in Section II-A, this is very rare indeed.

Second, and this is the main thrust of this paper, the task of determining equivalence can be made more tractable by applying *normalization* techniques, which transform different expressions of a construct to the same normal form. This is an idea that the software similarity community has made heavy use of (Section IV). Here, we start by evaluating a *minimal* form of normalization, which we find to be surprisingly effective in our target applications. We then investigate how far the idea can be taken, by coming up with increasingly aggressive normalizing transformations.

Concretely, we make the following contributions:

1) We provide an in-depth investigation on the efficacy of code normalization as an approach for identifying functions in binary executables in two application-relevant problems: identifying functions optimized for different CPU microarchitectures (III-B) and functions compiled by different versions of the same compiler (III-C). We show the practicality of minimal normalization for the first and, especially, the second problem.
2) We propose a set of *static* advanced normalization techniques (Section II-C) which, combined with known transformations, are shown to significantly improve on the minimal approach for the first problem. Conversely, we argue that even aggressive use of normalization is an impractical approach for the second problem.
3) We analyze (III) and, when possible, quantify (III-D) the effects of the normalizations and qualitatively discuss the obstacles to potential future work.

Our framework is available under a free software license at https://github.com/vusec/kamino.

II. Normalization

A. Precompilation

Different compilers or even different versions of the same compiler introduce a multitude of differences in the generated code. If we are to have any hope of identifying a function in machine code form by matching it to any function in our registry of known functions, we would do well to begin our attempts from a version which has been through a similar translation process.

For this reason, we first preprocess the implementations of functions in the registry with an array of compilers and compiler versions, at all available optimization levels. Under the assumption that at least one of the compiler versions we use is *related* to the actual compiler version used to compile the function embedded in a binary, this reduces the scope of the problem to verifying the equivalence of two machine code versions of a function—compiled by very similar compiler versions and at the same optimization level. Normalization can then account for the differences stemming from different combinations of the flags which affect code generation and reasonably constrained modifications of the compiler itself. This is still a difficult problem: flags which affect code generation are numerous and render exhaustive precompilation infeasible. As we will see in Section III, clustering can be applied to the results of precompilation to significantly reduce the final number of compilation artifacts.

In the case of malware, one may question whether it is reasonable to assume access to a closely related compiler version. In theory, at least, malware may use modified compilers and custom translations. In practice, Chaki et al. [6] attest that the overwhelming majority of malware samples they were able to examine were compiled with one of three widely-available compiler families. Indeed, concerns about attribution should even *discourage* any sufficiently-funded organization that spreads malware from using a non-standard compiler. Legitimate software vendors also need to choose from a limited set of compilers when translating their code.

B. Function Identification With Minimal Normalization

On 32-bit X86, code is typically compiled to be loaded at a fixed address (position-dependent), so that even function-local jumps will refer to different addresses. This alone makes function identification by means of byte-by-byte comparisons impractical. As a first attempt at function identification, we need to abstract away what are purely *syntactical* differences in the generated machine code. To do so, we lift the machine code to an intermediate representation. For this task, we rely on version 0.6 of the Binary Analysis Platform [7]. We use BAP to obtain the control flow graph (CFG) of every function. Conveniently, the lifting process also eliminates NOP instructions (of either short or long form). An important outcome of this step is that the structure of the CFG has become explicit, instead of being encoded in the addresses referenced by the function's control flow instructions.

Given this first normalization step, we determine if two functions are equivalent by proceeding according to Algorithm 1. Starting from the entry point of each function, we iterate over the CFGs in lockstep. For each basic block pair, we require that the successor edges are of the same number and kind. We then recursively examine the successors as determined by the pairing of the edge labels (taken-if-true, taken-if-false, fall-through).

It is easy to show that the algorithm requires the two CFGs to be isomorphic in order to yield a match. For each basic block pair, we additionally require the statements of the intermediate representation to be the same, except for the jump targets in control flow statements. This algorithm is simple to implement and we will see that

Algorithm 1 Minimal Equivalence Algorithm

```
function COMPARE-EDGES(edges_L, edges_R)
    (e_L, rest_L) ← TAKE-FIRST(edges_L)
    (e_R, rest_R) ← TAKE-FIRST(edges_R)
    if LABEL(e_L) = LABEL(e_R) then
        COMPARE-EDGES(rest_L, rest_r)
    else
        false
    end if
end function
function COMPARE-BBS(BB_L, BB_R)
    if COMPARE-STATEMENTS(BB_L, BB_R) then
        return false
    end if
    succ_L ← SUCCESSOR-EDGES(BB_L)
    succ_R ← SUCCESSOR-EDGES(BB_R)
    if LENGTH(succ_L) ≠ LENGTH(succ_R) then
        return false
    end if
    edges_L ← SORT-BY-LABEL(succ_L)
    edges_R ← SORT-BY-LABEL(succ_R)
    COMPARE-EDGES(edges_L, edges_R)
end function
function VISIT-BBS(BB_L, BB_R)
    if BB_L ∈ visited_L ∧ BB_R ∈ visited_R then
        true
    else if BB_L ∉ visited_L ∧ BB_R ∉ visited_R then
        COMPARE-BBS(BB_L, BB_R)
    else
        false
    end if
end function
function COMPARE-CFGS(L, R)
    visited_L ← ∅
    visited_R ← ∅
    entry_L ← ENTRY(L)
    entry_R ← ENTRY(R)
    VISIT-BBS(entry_L, entry_R)
end function
```

despite its simplicity, it is remarkably effective in specific domains (Section III-C).

C. Function Identification With Advanced Normalization

We now discuss a more advanced normalization procedure to handle cases where the compiler introduced more subtle differences between the two functions. First, we make the case for our equivalence verification algorithm, which is both sound and abstracts away syntactic differences in the computation. Then, we describe the sequence of normalizing transformations that we perform, so that the equivalence algorithm will only have to deal with a highly canonicalized form.

As an example, consider differences in register allocation. If a BB in a registry function uses register EAX, while the BB in the second (candidate) function uses EBX, the functions may still be equivalent, provided the register usage is consistent not just in this BB, but also in the others. For instance, if in one function BB1 assigns a value to EAX and increments that value in BB2, then the other function should write to EBX in BB1' and increment EBX in BB2'. Similarly, a program may have reordered some instructions while still being equivalent. And so on.

1) Abstraction of computation: While Algorithm 1 is simple and quick about identifying equivalent functions, it fails to account for a large number of transformations that compilers introduce during code generation and optimization.

As a more general approach, we propose to express the computations performed in each basic block symbolically. We associate each basic block with a set of *transfer functions* (TFs) that express the outputs of a basic block (BB) as a function of its inputs. The TFs serve to abstract the *computation* performed in a BB away from such expression details as register or stack slot selection, register spills and the order of evaluation for the intermediate subexpressions (i.e., BB-local instruction reordering).

To build the transfer functions of a basic block, we consider its register and memory outputs. Specifically, we consider as outputs those registers that have uses in the parts of the CFG that are reachable from the BB in question. Clearly, there needs to be one TF for every output location. Additionally, for a BB ending in a conditional jump, the condition code is considered an output of the BB (in effect, it is used by the control flow decision). Memory stores are all outputs of a BB, since they are observable outside the function.

We calculate transfer functions from the Static Single Assignment (SSA) form of the intermediate representation. We also consider memory an input variable and update its SSA index after each store. Each resultant TF forms an expression tree, where the leaves might be memory regions, register values or constants. Inner nodes might contain arithmetic operations as well as the ϕ-functions introduced by the conversion to SSA form. Needless to say, we apply a number of arithmetic and logical identities in a single direction, in order to normalize the symbolic expression of the captured computation.

While the procedure sounds complex, a simple example will clarify things. Figure 1 presents two synthetic translations of the given C code, chosen for their explanatory value. Notice how the syntactic differences of the X86 code are abstracted away in the symbolic representations we have derived in Table I.

We determine that two transfer functions perform the same computation by recursively walking down the respective expression trees. The comparison continues as long as the inner nodes match, until we reach the leaves of the TF expressions. Those leaves can either be constants (in which case they have to be the same for the comparison

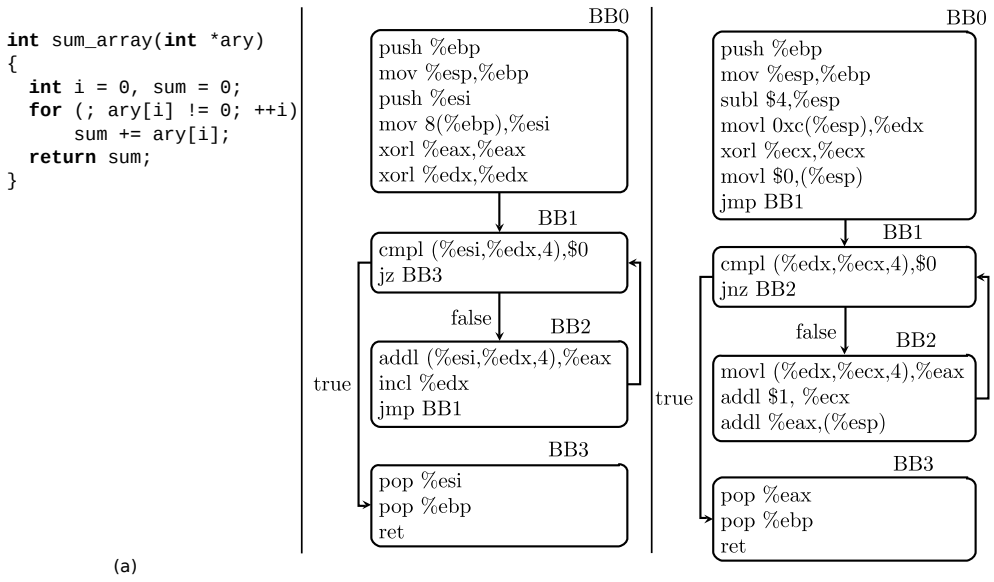


Fig. 1: **(a)** function sum_array(), **(b)** 1^{st} CFG, **(c)** 2^{nd} CFG

sum_array1		sum_array2	
BB0	$EDX_0 \leftarrow 0$ $ESI_0 \leftarrow R_0^{(1)}[0]$ $EAX_0 \leftarrow 0$	BB0	$ECX_0 \leftarrow 0$ $EDX_0 \leftarrow R_0^{(1)}[0]$ $R_0^{(2)}[0] \leftarrow 0$
BB1	$ZF_0 \leftarrow MEM_0[ESI_0 + \phi(EDX_0, EDX_1) * 4] == 0$ $temp_0 \leftarrow MEM_0[ESI_0 + \phi(EDX_0, EDX_1) * 4]$	BB1	$ZF_0 \leftarrow MEM_0[EDX_0 + \phi(ECX_0, ECX_1) * 4] == 0$ $temp_0 \leftarrow MEM_0[EDX_0 + \phi(ECX_0, ECX_1) * 4]$
BB2	$EDX_1 \leftarrow \phi(EDX_0, EDX_1) + 1$ $EAX_1 \leftarrow \phi(EAX_0, EAX_1) + temp_0$	BB2	$ECX_1 \leftarrow \phi(ECX_0, ECX_1) + 1$ $R_1^{(2)}[0] \leftarrow \phi(R_0^{(2)}, R_1^{(2)})[0] + temp_0$
BB3		BB3	$EAX \leftarrow R_1^{(2)}[0]$

TABLE I: Transfer functions for the example

BB pair	IN	OUT
$BB0 \longleftrightarrow BB0$	$\{R_0^{(1)}\} \longleftrightarrow \{R_0^{(1)}\}$ -	$\{EAX_0, EDX_0\} \longleftrightarrow \{ECX_0, R_0^{(2)}]\}$ $\{ESI_0\} \longleftrightarrow \{EDX_0\}$
$BB1 \longleftrightarrow BB1$	$\{EDX_0, EDX_1\} \longleftrightarrow \{ECX_0, ECX_1\}$ $\{ESI_0\} \longleftrightarrow \{EDX_0\}$ $\{MEM_0\} \longleftrightarrow \{MEM_0\}$	$\{ZF_0\} \longleftrightarrow \{ZF_0\}$ $\{temp_0\} \longleftrightarrow \{temp_0\}$ -
$BB2 \longleftrightarrow BB2$	$\{EAX_0, EAX_1, temp_0\} \longleftrightarrow \{temp_0, R_0^{(2)}, R_1^{(2)}\}$ $\{EDX_0, EDX_1\} \longleftrightarrow \{ECX_0, ECX_1\}$ $\{MEM_0\} \longleftrightarrow \{MEM_0\}$	$\{EAX_1\} \longleftrightarrow \{R_1^{(2)}\}$ $\{EDX_1\} \longleftrightarrow \{ECX_1\}$
$BB3 \longleftrightarrow BB3$	-	-

TABLE II: Location Mappings after the comparison of the transfer functions

to continue) or program locations that are inputs of the basic block.

When we encounter non-constant leaf nodes on both expressions, we declare that part of the comparison a success, under the assumption that the two leaf nodes (i.e. registers or memory locations) have a correspondence. This kind of *location mapping* (LM) can include more leaf nodes when input locations are not necessarily distinct. For example, when two TFs, $(a + b)$ and $(c + d)$, are declared equivalent, the input is $\{a, b\} \longleftrightarrow \{c, d\}$. This applies recursively to more complex transfer functions, as for the TFs derived for BB2 and the respective location mappings in Table II. This is because of the local scope of the TF comparison; when we are able to declare functions equivalent, we always produce accurate mappings between the input locations of the TFs.

Two basic block perform the same computation iff they compute the same values from the same inputs. Similarly, suppose that two functions have isomorphic CFGs, corresponding BBs perform the same computations and the location mappings are consistent on every pair of corresponding edges. Then, the two functions are equivalent as every feasible execution path writes to the same number of output locations and fills them in with the same computed values [4] (input locations are the same by assumption).

We ensure the consistency of two location mappings on CFG edges using Algorithm 2, where one LM (location mapping) is part of the OUT LMs of the source and

the other belongs to the IN LMs of the destination. We formulate and solve the consistency of all the location mappings incident to an edge as a constraint satisfaction problem [8] on top of Algorithm 2. Finally, we perform fixed-point iteration to ensure that the inferred location mappings are consistent across the whole set of edge pairs. When the algorithm completes, all the location mappings in Table II devolve to singular mappings.

Algorithm 2 Consistency Check of Location Mappings

```
function CONSISTENCY(IN, OUT)
    (IN_L, IN_R) ← IN
    (OUT_L, OUT_R) ← OUT
    LEFT ← IN_L ∩ OUT_L
    RIGHT ← IN_R ∩ OUT_R
    if |LEFT| ≠ |RIGHT| then
        Conflict
    else if LEFT = ∅ then
        (IN, OUT)
    else if |IN_L| = |LEFT| ∧ |OUT_L| = |LEFT| then
        (IN, OUT)
    else
        IN' ← (IN_L − LEFT) ⟷ (IN_R − RIGHT)
        OUT' ←
            (OUT_L − LEFT) ⟷ (OUT_R − RIGHT)
        (IN', OUT')
    end if
end function
```

2) Compiler-aware transformations: Symbolically expressing the computation performed in basic blocks only scratches the surface of normalizing two dissimilar pieces of machine code. As long as we apply the same normalizing transformation consistently and given that the intermediate representation after a transformation is equivalent to the one before, we are free to move, add or delete statements.

As an obvious way forward, we apply *global (SCC-based) value numbering*, which reuses intermediate expression results. This has the potential to make many computations redundant and expose them to dead code elimination (DCE). DCE is similarly helpful for deleting unused computations (e.g. most EFLAGS calculations). The idea with this pass is to duplicate transformations typically performed by the compiler. If the transformation has been performed, our own pass will be a no-op. If it has not, then we will apply it and in the process bring a number of constructs in normal form. Concretely, we extend BAP's value numbering algorithm to fold constants and *normalize the tree-like structure of arithmetic expressions* so that equivalent expressions are isomorphic.

Another significant normalizing transformation we perform is the *naming of stack slots* (R variables in Table II). By statically analyzing memory references, we are able to determine the locations of memory accesses which are local to the stack frame and upgrade them to variables (including arrays, as long as we can statically determine the bounds). This works to our advantage in two ways. First, the location becomes transparent to the value numbering pass, hence values temporarily stored to the stack are forwarded to their uses and the temporary store is elided. Second, the stack slot becomes a location which can participate in a location mapping with what might be a register on the other side. In effect, stack locals can be treated straightforwardly as registers.

Partial redundancy elimination (PRE) is a significant compiler transformation which may add basic blocks in order to make expressions fully redundant. Contrary to our approach for value numbering, we implemented a *"buoyancy"* pass which moves variables definitions up the dominator tree of basic blocks, to the earliest point at which their dependencies are available. This does not only canonicalize the location of computations, but also renders BBs added by PRE redundant, so that they are removed by subsequent DCE and basic block coalescing passes.

On X86, use of partial registers can introduce false dependencies to the originating location of the unused bits of a register. This results in the use of dead values (and unnecessary introduction of ϕ functions) that are hard for DCE to eliminate. In order to normalize away this compiler choice, we perform *BB-local DCE with tracking of sub-register values* before the conversion to SSA form.

At the CFG level, we also make sure to *normalize conditional jumps* so as to make sure that there are no negated computations (either arithmetic or boolean). We ensure that CFGs have a single exit BB and deduplicate the trailing instructions to *undo the effects of BB cloning*. Thus, the transformed code becomes less sensitive to the syntactic expression of control flow decisions.

3) Implementation Limitations: The aggressive normalization that we consider here is implemented on top of the Binary Analysis Platform 0.6 which only supports 32-bit X86 binaries. We assume accurate disassembly and function boundary identification. Since our target applications did not require it, we only consider functions that do not contain calls. Similarly, we do not resolve jump tables, hence we've had to disqualify some function implementations from our evaluation set (Table III). Those are assumptions which are orthogonal to our approach and could be lifted with additional implementation effort [9].

III. Evaluation

Our evaluation seeks to answer the question: is normalization useful and sufficient for identifying functions in a registry? We examine the question in two scenarios: finding a function compiled for a different CPU architecture (which would enable binary rejuvenation) and finding a function that has been compiled with closely related but unknown or unavailable compiler version (a concern both for reverse engineering and binary rejuvenation).

A. Data set and Compiler versions

For this evaluation, we built all GCC versions from 3.0.0 to 4.9.2, from source, in a single Debian Lenny 32-bit chroot and performed all precompilation inside it. A

Primitive	Implementations
memcpy	DragonFly, Morrow [10], musl
strlen	dword-at-a-time, dietlibc, libc4, musl
strcmp	libc4, musl
strncmp	libc4, musl
memset	dietlibc, libc4, musl, naive
crc32	Brown

TABLE III: Evaluation set: implementations of well-known functions

container-based setup would allow installing the compilers from distribution packages and can achieve comparably fast precompilation. The bulk of the work to assemble the array of compiler versions took approximately a week. This cost would be paid up front for any system relying on precompilation; keeping the array of compilers up to date as new versions are released is a marginal maintenance cost.

For the registry of well-known functions, our goal was to evaluate on a representative set of functions. We settled on the set of functions and implementations listed in Table III. All these functions have been the target of intense manual optimization over the years, something attested to by their eventual inclusion into modern compilers as intrinsic functions. That makes them ideally representative of the set of functions which might end up embedded in binaries in inefficient, buggy or insecure versions. In other words, they serve as an unbiased proxy for the set of functions that are candidates for binary rejuvenation. Simultaneously, they are typical instances of the easily-summarizable utility functions one runs into while reverse engineering a binary.

We performed all our experiments on a 16-core 96GB VM. Compiling the template functions (Table III) for all microarchitectures supported by each compiler version is an embarrassingly parallel problem and only took 213s. These run times suggest that precompilation with a large array of compilers and compiler versions is practical and that its cost is dominated by the person-hours required to import a new compiler version into the system.

B. Normalization of microarchitecture differences

We start our evaluation with a problem pertinent to binary rejuvenation. Namely, we investigate how effective normalization is in eliminating differences in the resulting binary code, which originate from the vendor targeting different deployment environments. It is fairly common for a vendor to be distributing binaries compiled for a "lowest common denominator" microarchitecture, to make sure the resulting binary only makes use of instructions which are available on most CPU chips in use. Similarly, an organization which has standardized on a source-based OS distribution may elect to target the specific microarchitecture which is in use on their hardware platform, hoping for increased performance. Clearly, the compiler version used and the targeted microarchitecture vary over time, making this a concern for binary rejuvenation efforts. Importantly,

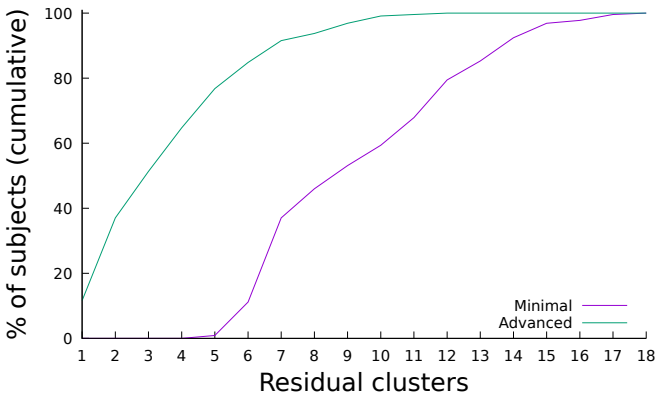


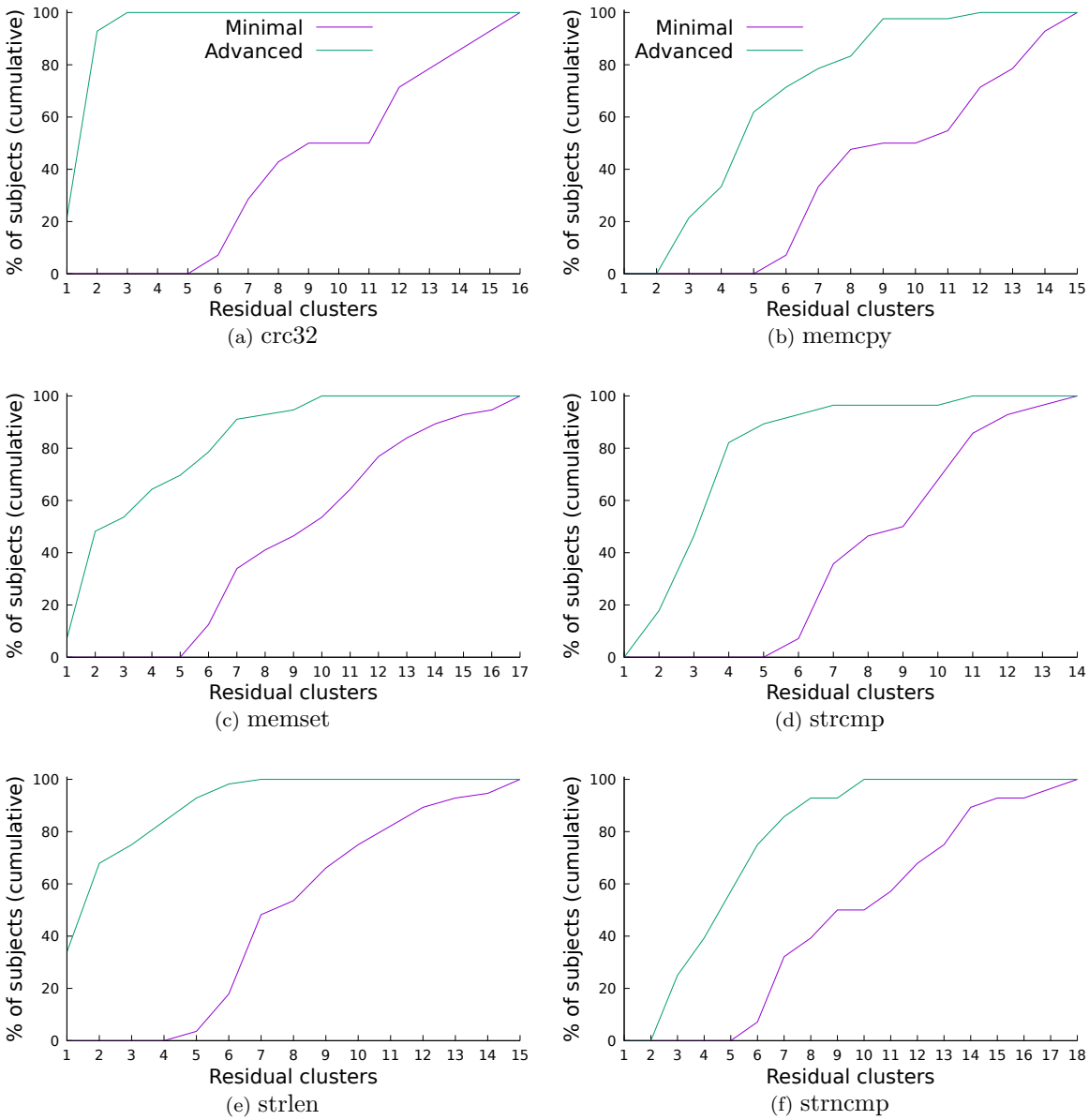
Fig. 2: Cumulative Distribution Function (CDF) of the percentage of subjects against residual clusters (aggregate)

code generation is affected by dozens of compiler flags, any combination of which would need to be accounted for.

Normalization is a way of keeping the problem manageable. In order to quantify its effectiveness, we consider the binary versions of all the functions in our evaluation data set, compiled at -O2, with all available major GCC versions (a major version corresponds to months or years of development effort, i.e. for GCC it is of the form x.y.0) and targeting all microarchitectures known to the compiler. Clustering takes place between compilation artifacts sharing the tuple (*primitive*, *impl*, *ccver*), i.e. only the `-march` varies. We name this tuple a *subject* and artifacts which we were able to declare equivalent form a *cluster*. After the experiment, we report the final number of clusters for a subject. This gives us a representative metric of the degree to which normalization is sensitive to the variance introduced by targeting different CPU microarchitectures. Note that functions are never misidentified, thus our metric is not precision/recall as for binary clone detectors.

Because of the significantly different run times of the algorithms, we first apply the minimal method and then further limit the number of clusters by doing advanced normalization. To exploit the available parallelism of the problem, we use a divide-and-conquer algorithm which recursively divides the set of clusters in two, calls out to the normalization tools, and merges clusters which were deemed equivalent.

The aggregated results can be seen in Figure 2. The Y axis represents the percentage of subjects that are left with X clusters, after each method has finished. Note that we elect the residual number of clusters as a metric, because the initial number of subjects varies with the compiler version; newer versions of GCC support more microarchitectures than older ones. We believe this puts the focus on the suitability of each method to applications. For completeness, we depict the number of residual for a function of which we only had one implementation to test with, `crc32`, in Figure 4. In this case, the minimum



(a) crc32 (b) memcpy (c) memset (d) strcmp (e) strlen (f) strncmp

Fig. 3: CDF of the percentage of subjects against residual clusters (per symbol)

number of clusters was two, for 6 of the 14 GCC versions we tested with.

The differences in the resulting binaries are primarily because of different instructions selected during code generation and instruction reordering. There is enough variation that the minimal normalization algorithm never manages to limit the number of clusters below six. The advanced normalization algorithm fares much better and is able to limit the number of clusters to at most two in the majority of the cases (in aggregate).

Figure 3 breaks down the residual clusters by symbol. The high-level pattern we observe is that, the simpler the function, the better the results. A less complicated function presents less opportunity for variation during code generation, hence benefits both normalization methods. The functions on the left column involve a single loop, whereas the functions on the right column are typically implemented with two nested loops.

During the parallel execution, the minimal normalization utility responded after a median of 1.21s (1.79s on average), whereas the advanced utility needed a median 167s (average 277s) over all invocations. The difference in complexity also manifests itself in the overall duration of the function identification. As the run time is dominated by the time taken for the advanced normalization, the consistency check of the location mappings plays a more significant role for certain implementations of `strcmp` (Figure 5).

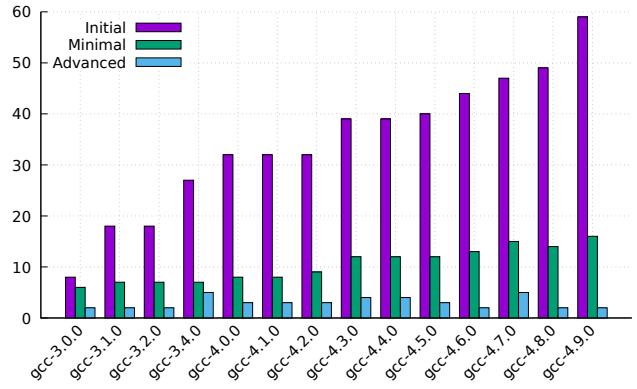


Fig. 4: Residual clusters for `crc32` (per compiler version)

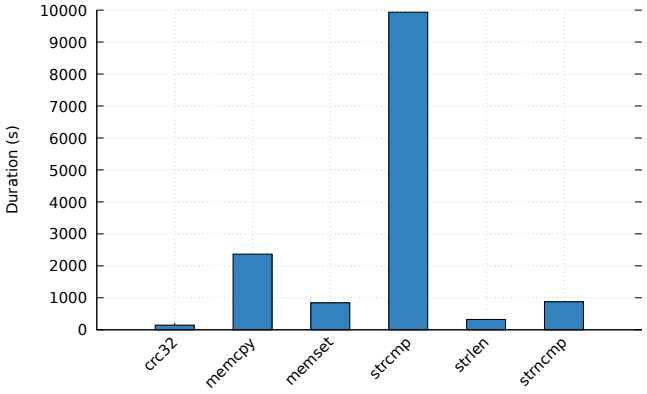


Fig. 5: Execution time (per symbol)

Here we can observe the limits of the advanced normalization approach, as sampling of the cases that end up grouped in two clusters reveals a common issue. On the K6 microarchitecture family, GCC prefers the `loop` instruction, which results in an induction variable which is being decremented, whereas in other architectures the induction variable is typically incremented. This suggests the idea of normalization for induction variables. Unfortunately, manual sampling of the mismatches in other functions reveals a range of disparate causes, none of which appear to dominate. This is in line with our experience in developing new normalizations (III-D).

C. Normalization of minor differences in the compiler version

How sensitive is normalization to changes introduced by a different compiler version? This is an important question, as it may be that the exact version of the compiler used to translate a function is not available, but a related one is. Operating system vendors (including distributions) tend to carry local modifications to the upstream version of the compiler they distribute. Similarly, individual code bases occasionally override the default compiler behavior in small things (e.g. omitting the frame pointer).

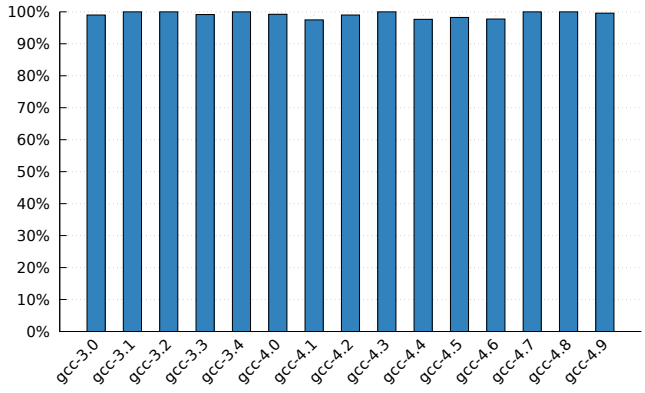


Fig. 6: Successfully identified minor subjects within a GCC major group

	Initial subjects	Identified
Minimal	5760	5665 (98.35%)
Advanced	95	45 (47.37%)
Combined	5760	5710 (99.13%)

TABLE IV: Successfully identified minor subjects within a GCC major group, by normalization method

Relying on the same evaluation set, we compile the set of functions in Table III with all minor compiler versions (i.e. GCC x.y.z), this time varying the optimization level (but not the microarchitecture). Therefore, our subjects comprise all combinations of ($function$, x, y, z, $olvl$). Intuitively, we are investigating the worst-case effects of not having access to the exact compiler version used to produce a certain version of a function, but having access to a closely related one. Simultaneously, we aim to get an early sense for how sensitive normalization is to differences between related compiler versions (the question is further explored in Section III-D).

To quantify the efficacy of normalization, we partition our subjects into groups of related minor versions (e.g. 4.0.0-4, 4.1.0-2). We then try to identify each compiled function in that group, assuming we only have access to the remaining versions. So for function F compiled with GCC 4.1.1, we would try to identify it assuming we don't have access to that compiler version but only to 4.1.0 and 4.1.2. We consider all available optimization levels, but only perform comparisons at the same optimization level. As different optimization levels may introduce tens of transformation passes, we contend that those differences are better accounted for by precompilation at the same optimization level by potentially distant compiler versions (a scenario we consider in Section III-D).

The results are depicted in Figure 6. It is clear that normalization is up to the task of accounting for a function having been compiled with an unavailable compiler version, if that version is closely related to one in our array of compilers. Our identification percentage reaches up to

100% (in two cases) and only drops as low as 97.5%. Yet how aggressive does our normalization need to be? Table IV breaks down the successful identifications by normalization strength. It is clear that our minimal normalization is so effective that the advanced normalization barely needs to be employed. When it does need to be brought to bear, the advanced normalization helps with about half the remaining (harder) cases.

A different way of looking at this is that minor differences in the compiler versions rarely introduce significant differences in the produced code for our evaluation set. When they do, the changes are such that they are only partially accounted for even by advanced normalization strategies. Given the spectacular degree to which minimal normalization suffices for function identification in this scenario, and the speed with which it finishes (41.72s for the 12480 pairs of subjects examined), we submit that the stratagem of using a compiler array and only doing minimal normalization is adequate for implementing a robust function identification system. What is more, the latency of such a system could be low enough to allow for online use, for example to assist a reverse engineer interactively working with a sample. Equally, the performance of such a system would allow for the batch processing of a large number of binaries for the purposes of rejuvenation.

Given the results of Table IV, it is hard to justify the use of advanced normalization in this problem domain. Instead of the diminishing returns we anticipated when we set out, spending more implementation effort and CPU power on aggressive normalization is clearly of marginal utility and limited potency in this scenario. While it is not sufficient to prove the approach infeasible, we suggest that the number and strength of the tranformations we employed (II-C2) at the very least suggest that the avenue of function identification using expert normalization rules is impractical and that this should inform future research in this direction. This conclusion is further supported by the results of the next subsection.

D. Normalization of significant differences in the compiler version

Both the previous evaluation setups are biased in favor of small differences in the compilation process. At what point does normalization break down? To this purpose, we consider versions of a function compiled with adjacent major versions of the compiler (e.g. 4.0.0 and 4.1.0, 4.1.0 and 4.2.0 and so forth). Ergo, our subjects are all combinations of ($function$, x, y, $olvl$). In order to collect enough data points, we test all adjacent pairs, at all optimization levels (again keeping comparisons within the same optimization level). The results appear in Figure 7. Note that, contrary to Section III-C, the percentages refer to successful comparisons, not successfully identified subjects. This is to facilitate the discussion of limitations in the advanced normalization. If the accounting were similar to Section III-C, the percentages would be slightly higher.

	Initial	Of which
Identified after minimal	1040	190 (18.27%)
Isomorphic before advanced	850	110 (12.94%)
Isomorphic after advanced	850	445 (52.35%)
Identified after advanced	445	184 (41.34%)
Combined	1040	425 (40.86%)

TABLE V: Analysis of identifications for subjects from adjacent GCC major versions

We immediately notice that minimal normalization is much less effective across subjects compiled with different GCC major versions. This is to be expected, as the generated code is significantly affected by the improvements (and regressions) introduced between major compiler versions. To better understand the bounds that the advanced normalization can not push through, we plot the most significant one, non-isomorphism. The middle bars in Figure 7 present the percentage of function pairs that were isomorphic to begin with (before our compiler-aware transformations) and the percentage of isomorphic function pairs afterwards. Clearly, we are effective in undoing some of the CFG-altering transformations, but cannot account for the full range of compiler effects. In one pathological case, 3.4.0 vs 4.0.0, the percentage of non-isomorphism actually goes up. Space constraints do not allow us to present the effects of individual compiler-aware transformations. We have found that there is a large degree of synergy; their combined effects exceed the benefit of each one considered separately and no single one dominates. Put another way, each of the aggressive normalizations of subsection II-C2 only results in an incremental improvement in accuracy.

Focusing on the comparisons of isomorphic functions, advanced normalization can only determine equivalence in 4/10 cases. This strongly implies that even a hypothetical [1] introduction of cutpoints, as in [11], which would lift the requirement for compared functions to be isomorphic, would bring modest improvement at best.

In aggregate, normalization suffices to identify functions as equivalent in 40.86% of the cases. This result is not promising for building a reliable system of binary rejuvenation. We therefore conclude that, for cross-compiler version function identification, the coverage of available compilers needs to be comprehensive. That is, the registry needs to contain at least one translation of a function by a closely related compiler version, so that Figure 6 will apply. If that condition is not satisfied, advanced normalization would be of limited utility although, at 31 minutes to process the 1040 comparisons of this experiment, it could still save valuable person hours for the function it identifies, as a human would not need to look into the binary function at all to know its functionality and semantics.

[1] The location of cutpoints is informed by dynamic analysis, which may run into coverage issues.

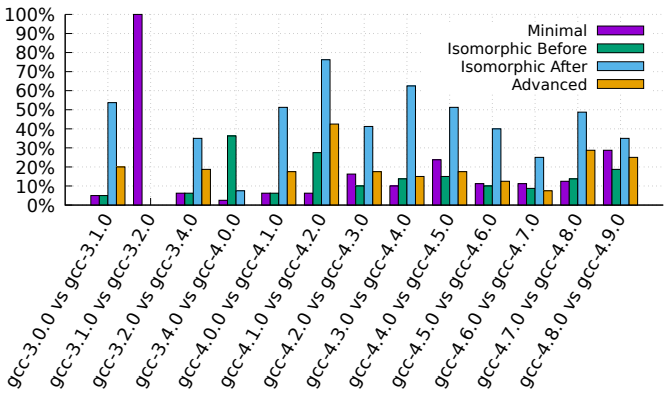


Fig. 7: Successful function equivalence verifications for subjects from adjacent GCC major versions

IV. Related Work

Most research on function identification focuses on *similarity*. While complementary to our work, similarity detection may, as a pre-filtering step, boost the scalability of *exact* function identification. In clone detection, Kononenko et al. [12] consider the effects of compilation as a normalizing transformation in clone detection. Recent work [13] employs similar ideas using a combination of compilation and decompilation to achieve normalization. BinClone [14] applies normalization to code fragments for clone detection. Their normalizer replaces a given register with its assigned class and cannot differentiate between registers in the same class, so that fragments declared as "exact clones" may not actually perform the same computation. Significantly, it detects clones in a linear scan with a sliding window, forgoing any CFG normalization.

Other approaches take into account the structure of the CFG. BinHunt [15] abstracts away syntactic differences in the basic blocks by expressing the computations in a BB as symbolic expressions, similar to our approach. However, it uses those expressions (only) to assign a matching strength score to pairs of BBs and makes no effort to ensure consistent use of input and output locations across all basic block pairings in the two CFGs.

In malware analysis, Caballero et al. [5] stress the necessity of focusing on self-contained functions with a well-defined interface. Inspired by compiler provenance [16], Chaki et al. [6] propose the concept of provenance-similarity, reporting that "a small number of related compilers are used to produce the vast majority of malware", which supports our assumption about the availability of compiler versions closely related to the one used for translating a specific malware sample.

While detection of *similar* binary code may be a mature field [17], [18], [19], [15], [20], [21], [22], [23], [24], [25], [26], this is not true for *equivalence* detection. Papadakis et al. [27] perform equivalent mutant detection in the context of source-based mutation testing by compiling to machine code and then checking whether the resultant function is identical. Given its domain, it does not consider use of more than one compiler version. Equivalence verification has also been successfully applied to in low-level embedded applications [28], [29]. The verification is based on finding formal relationships between the inputs and outputs of functions using symbolic execution [30] and relies on automated decision procedures to prove that the outputs are always equivalent. The scalability issues of the approach proposed by Currey et al. [31] are addressed by both [32] and [33]. Feng et al. [33] introduce *cutpoints* in order to constrain the blow up in memory usage when employing symbolic execution.

UC-KLEE [34] uses symbolic execution to verify automatically that two functions written in C are equivalent with the goal of *cross-checking*. It can deal with complex control- and data flows but makes non-trivial assumptions on object sizes. Coming from the field of translation validation [35], DDEC [11] uses cutpoints to divide a function into loop-free constituents. DDEC can inductively verify equivalence by proving that *invariants* hold at the transition of matched *cutpoints* by generating and verifying *verification conditions*. However, both for the *cutpoints* and the invariants [36], DDEC relies on dynamic analysis and, like all such approaches, strongly depends on coverage. Sharma et al. discuss how not being purely static can be a limitation, but believe that regression tests should suffice to provide the required input.

Shashidhar et al. [4] demonstrate equivalence checking for a specific class of programs at the binary level, allowing for three types of transformations. In contrast, our approach is simpler but covers a larger class of programs and considers common compiler transformations beyond those of [4] (e.g., partial redundancy elimination). Ciobâcă et al [37] show language-independent full program equivalence for simple languages which lack the expressivity of machine code (e.g., pointers). SYMDIFF [38] performs equivalence checking and generates a description of the semantic differences between two programs. Later work [39] builds on it to identify the root cause of semantic differences, but is limited to loop-free programs.

V. Conclusion

We have studied the potential and limitations of normalization with an eye on using it for reverse engineering and rejuvenation of legacy binaries. Minimal normalization turns out to be very useful for function identification, across expansive set of compiler versions. We implemented both novel and existing transformations to aggressively normalize away differences in the targeted microarchitecture and utilized compiler version. Our evaluation shows that advanced normalization is of practical utility for the first task, but of unexpectedly limited use in accounting for differences in the compiler version.

Acknowledgements

We would like to thank the anonymous reviewers for their comments. This work was supported by the United States Office of Naval Research (ONR) under contract N00014-17-1-2782 and by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing", NWO 639.021.753 VENI "PantaRhei", and NWO 629.002.204 "Parallax". This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

References

[1] A. Oikonomopoulos, C. Giuffrida, S. Rawat, and H. Bos, "Binary rejuvenation: Applications and challenges," *IEEE Security & Privacy*, vol. 14, no. 1, pp. 68–71, 2016.

[2] R. Koo, "Ibm invests in cobol modernization with advanced binary optimization," *IBM Systems Magazine*, October 2016.

[3] A. R. Bernat and B. P. Miller, "Anywhere , any-time binary instrumentation," in *Proc. of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, 2011.

[4] K. C. Shashidhar, M. Bruynooghe, and F. Catthoor, "Verification of source code transformations by program equivalence checking," *Compiler . . .*, 2005.

[5] J. Caballero, N. M. Johnson, S. McCamant, and D. Song, "Binary code extraction and interface identification for security applications," in *NDSS*, 2010.

[6] S. Chaki, C. Cohen, and A. Gurfinkel, "Supervised learning for provenance-similarity of binaries," in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '11. New York, NY, USA: ACM, 2011, pp. 15–23. [Online]. Available: http://doi.acm.org/10.1145/2020408.2020419

[7] D. Brumley, I. Jager, T. Avgerinos, and E. Schwartz, "Bap: A binary analysis platform," in *Proc. of the 23rd Int'l Conf. on Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds., 2011, pp. 463–469.

[8] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Pearson Education, 2010.

[9] X. Meng and B. P. Miller, "Binary code is not easy," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 24–35. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931047

[10] M. Morrow, "Optimizing memcpy improves speed," http://www.embedded.com/design/configurable-systems/4024961/Optimizing-Memcpy-improves-speed, 2004.

[11] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken, "Data-driven equivalence checking," in *Proc. of the 2013 ACM SIGPLAN Int'l Conf. on Object Oriented Programming Systems Languages & Applications*, 2013, pp. 391–406.

[12] O. Kononenko, C. Zhang, and M. W. Godfrey, "Compiling clones: What happens?" in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sept 2014, pp. 481–485.

[13] C. Ragkhitwetsagul and J. Krinke, "Using compilation/decompilation to enhance clone detection," in *2017 IEEE 11th International Workshop on Software Clones (IWSC)*, Feb 2017, pp. 1–7.

[14] M. Farhadi, B. Fung, P. Charland, and M. Debbabi, "Binclone: Detecting code clones in malware," in *Proc. of the Eighth Int'l Conf. on Software Security and Reliability*, 2014, pp. 78–87.

[15] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *ICICS '08: Proceedings of the 10th International Conference on Information and Communications Security*, 2008.

[16] N. E. Rosenblum, B. P. Miller, and X. Zhu, "Extracting compiler provenance from program binaries," in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '10. New York, NY, USA: ACM, 2010, pp. 21–28. [Online]. Available: http://doi.acm.org/10.1145/1806672.1806678

[17] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: dynamic similarity testing for program binaries and components," in *SEC'14: Proceedings of the 23rd USENIX conference on Security Symposium*, 2014.

[18] Y. David and E. Yahav, "Tracelet-based code search in executables," in *Proc. of the 35th ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2014, pp. 349–360.

[19] H. Flake, "Structural comparison of executable objects," in *Proc. of the Int'l Conf. on Detection of Intrusions and Malware & Vulnerability Assessment*, 2004, pp. 161–174.

[20] J. Jang, D. Brumley, and S. Venkataraman, "Bitshred: feature hashing malware for scalable triage and semantic analysis," in *Proc. of the 18th ACM Conf. on Computer and Communications Security*, 2011, pp. 309–320.

[21] J. Jang, M. Woo, and D. Brumley, "Towards automatic software lineage inference," in *Proc. of the 22nd USENIX Security Symp.*, 2013, pp. 81–96.

[22] W. M. Khoo, A. Mycroft, and R. Anderson, "Rendezvous: A search engine for binary code," in *Proc. of the Tenth Working Conf. on Mining Software Repositories*, 2013, pp. 329–338.

[23] A. Lakhotia, M. D. Preda, and R. Giacobazzi, "Fast location of similar code fragments using semantic 'juice'," in *Proc. of the Second ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, 2013, pp. 1–6.

[24] B. H. Ng and A. Prakash, "Exposé: Discovering potential binary code re-use," in *Proc. of the IEEE 37th Annual Computer Software and Applications Conf.*, 2013, pp. 492–501.

[25] A. Saebjornsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proc. of the 18th Int'l Symp. on Software Testing and Analysis*, 2009, pp. 117–128.

[26] Z. Wang, K. Pierce, and S. McFarling, "Bmat: a binary matching tool for stale profile propagation," *Journal of Instruction-Level Parallelism*, vol. 2, pp. 1–20, 2000.

[27] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 936–946. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818867

[28] X. Feng and A. J. Hu, "Automatic formal verification for scheduled vliw code," in *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems*, 2002.

[29] D. W. Currie, A. J. Hu, S. Rajan, and M. Fujita, "Automatic formal verification of dsp software," in *Design Automation Conference, 2000. Proceedings 2000*, 2000.

[30] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, 1976.

[31] X. Xiao, S. Li, T. Xie, and N. Tillmann, "Characteristic studies of loop problems for structural test generation via symbolic execution," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, Nov 2013, pp. 246–256.

[32] T. Matsumoto, H. Saito, and M. Fujita, "Equivalence checking of c programs by locally performing symbolic simulation on dependence graphs," *Quality Electronic Design, 2006. ISQED '06. 7th International Symposium on*, 2006.

[33] X. Feng and A. J. Hu, "Cutpoints for formal equivalence verification of embedded software," in *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, 2005.

[34] D. A. Ramos and D. R. Engler, "Practical, low-effort equivalence verification of real code," in *CAV'11: Proceedings of the 23rd international conference on Computer aided verification*, 2011.

[35] G. C. Necula, "Translation validation for an optimizing compiler," in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2000, pp. 83–94.
[36] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "Using dynamic analysis to discover polynomial and array invariants," in *ICSE '12: Proceedings of the 34th International Conference on Software Engineering*, 2012.
[37] Ş. Ciobâcă, D. Lucanu, V. Rusu, and G. Roşu, "A language-independent proof system for full program equivalence," *Formal Aspects of Computing*, vol. 28, no. 3, pp. 469–497, 2016.
[38] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, "Symdiff: a language-agnostic semantic diff tool for imperative programs," in *CAV'12: Proceedings of the 24th international conference on Computer Aided Verification*, 2012.
[39] S. K. Lahiri, R. Sinha, and C. Hawblitzel, *Automatic Root-causing for Program Equivalence Failures in Binaries*. Cham: Springer International Publishing, 2015, pp. 362–379.