Fast Location of Similar Code Fragments Using Semantic 'Juice'

Arun Lakhotia
Center for Advanced Computer Studies
University of Louisiana at Lafayette
Lafayette, LA, U.S.A.
arun@louisiana.edu

Mila Dalla Preda
University of Bologna
Bologna, Italy
dallapre@cs.unibo.it

Roberto Giacobazzi
University of Verona
Verona, Italy
roberto.giacobazzi@univr.it

Abstract

Abstraction of semantics of blocks of a binary is termed as 'juice.' Whereas the denotational semantics summarizes the computation performed by a block, its juice presents a template of the relationships established by the block. BinJuice is a tool for extracting the 'juice' of a binary. It symbolically interprets individual blocks of a binary to extract their semantics: the effect of the block on the program state. The semantics is generalized to juice by replacing register names and literal constants by typed, logical variables. The juice also maintains algebraic constraints between the numeric variables. Thus, this juice forms a semantic template that is expected to be identical regardless of code variations due to register renaming, memory address allocation, and constant replacement. The terms in juice can be canonically ordered using a linear order presented. Thus semantically equivalent (rather, similar) code fragments can be identified by simple structural comparison of their juice, or by comparing their hashes. While BinJuice cannot find all equivalent constructs, for that would solve the Halting Problem, it does significantly improve the state-of-the-art in both the computational complexity as well as the set of equivalences it can establish. Preliminary results show that juice is effective in pairing code variants created by post-compile obfuscating transformations.

1. Introduction

There is a growing need for the comparison of binary executables in applications such as binary patching [5, 17], malware analysis [6, 9, 13] and copyright infringement investigation [1]. In these applications, it is important that the comparison algorithm account for changes due to code evolution, changes in compiler optimizations, and post-compile obfuscations. Furthermore, while binary patching typically requires comparing a pair of binaries known to be successive versions of the same software, the other applications require searching for matches to a given binary within a large collection of binaries. For malware analysis in particular, this collection may be extremely large, consisting of hundreds of thousands, if not millions, of malware.

This paper introduces a notion of 'juice', a generalization of the denotational semantics of a program. The juice captures the essential relations established by a piece of code, independent of choices of registers and literal constants. The juice then serves as a template of the code that is invariant against certain choices made by compilers or by code obfuscation tools.

Table 1 presents an example of binary code, its semantics, and its juice. The first column presents the hex dump of an executable code fragment along with its disassembled code, the second column contains the (denotational) semantics of the code fragment, and the third column shows its juice. The semantics give the result of executing the code fragment as a function of the state before execution. This state is given by the function 'def' (for default). Thus, $\mathrm{def}(\mathrm{ebx})$ represents the content of the 32-bit register ebx at the entry of the code fragment. The semantics indicates that upon execution of the code fragment, the register eax will contain the value 5 and the register ebx will contain the result of multiplying $\mathrm{def}(\mathrm{ebx})$ by 5 and adding 20. The presented semantics also contains the steps in computing the value of ebx. It is assumed that the state of all other register and memory locations remain unchanged. For simplicity the affect on the flag registers is not presented. The juice is computed by replacing in the semantics the register names and literal constants with typed variables and introducing algebraic constraints. In the presented juice, the symbols $\mathrm{N1}$, $\mathrm{N2}$, and $\mathrm{N3}$ are assumed to be of numeric type and the other symbols are 32-bit registers. The juice shows that the register variable A will contain number $\mathrm{N1}$ and register variable B will contain the number computed by multiplying the previous value of B by number $\mathrm{N1}$ and adding number $\mathrm{N2}$. Further, numbers $\mathrm{N1}$ and $\mathrm{N2}$ are related using a third number $\mathrm{N3}$ such that number $\mathrm{N2}$ is equal to number $\mathrm{N1}$ multiplied by number $\mathrm{N3}$. In other words, number $\mathrm{N3}$ is a multiple of number $\mathrm{N1}$.

As illustrated by the above example, the juice of a code fragment is an abstraction of its semantics, which in turn is an abstraction of code. In the above example, the semantics in the second column may be used to represent all code fragments that result in eax containing the value 5 and ebx containing the value $\mathrm{def}(\mathrm{ebx}) \times 5 + 20$, leaving all other registers and memory unchanged. The juice, on the other hand, represents all code fragments whose semantics can be abstracted as the given algebraic and type constraints. In the above example, the juice represents all code fragments that result in one 32-bit register (A) containing some specific number ($\mathrm{N1}$) and the value in a second 32-bit register (B) being multiplied by the previous value ($\mathrm{N1}$) and summed with a second number ($\mathrm{N2}$), where $\mathrm{N2}$ is a multiple of $\mathrm{N1}$.

Juice may be computed at varying levels of abstractions. For instance, at the lowest level, one may abstract the register names, but not the literal constants. Such an abstraction may be used to relate code fragments that have the same semantics, modulo register names. On the other end, the literal constants may be generalized

PPREW '13 Jan 26, 2013, Rome, Italy.

(a) Code	(b) Semantics	(c) Juice
401290: b8 05 00 00 00 mov eax,0x5 401295: 81 c3 04 00 00 00 add ebx,0x4 40129b: 6b c3 imul eax,ebx	eax = 5 ebx = $(\mathrm{def(ebx)} + 4) \times 5$ = $\mathrm{def(ebx)} \times 5 + 20$	A = N1 B = $\mathrm{def}(B) \times \mathrm{N1} + \mathrm{N2}$ where $\mathrm{N2} = \mathrm{N1} \times \mathrm{N3}$ and $\mathrm{type}(A) = \mathrm{type}(B) = \mathrm{reg32}$

Table 1. Example code, semantics, and its juice

but the algebraic and register size constraints may be ignored, thus significantly expanding the code fragments that may be placed in an equivalence class. Consider the juice resulting from ignoring the register size and algebraic constraints in the example of Table 1. The resulting juice will relate all code fragments with the semantics that one register, A, of any size contains some number, say N1, and a second register, B, contains the $\mathrm{def}(B) \times \mathrm{N1} + \mathrm{N2}$, where N2 is some number (with no explicit relation to N1).

We have implemented a system BinJuice that computes the semantics and the juice of individual blocks of instructions, where a block is defined in the classical sense. The juice of a block consists of three components: the generalized semantics, the generalized algebraic constraints, and the type constraints. Depending on the need of the application, one may use the code (from the original program), the semantics, or the juice at any of the varying levels of abstractions.

BinJuice provides an important building block for various applications requiring comparison of binaries. At the core level, BinJuice abstracts a given binary to its semantics and its juice. A given pair of binaries can then be compared by matching the semantics, the juice, or both. BinJuice provides the primitives for finding differences in two related binaries as that performed by BinDiff [17] or BinHunt [5]. On the other hand, the juice extracted by BinJuice may be used to create features for data mining tools, such as BitShred [6], BigGrep [8], or Vilo [15], to aid applications that require searching a large collection of binaries for matches.

This paper makes the following contributions:

- It introduces the concept of juice, an abstraction of semantics along with algebraic constraints relating literal constants.
- It presents a novel method to compute the algebraic constraints in the juice.
- It presents an ordering relation that enables fast equality test of semantic terms and of juice terms.

The rest of this paper is organized as follows. Section 2 presents an overview of related works. Section 3 presents our algorithm for computing juice. Section 4 describes the result of our preliminary experiments in the use of BinJuice. Section 5 discusses the limitations of the method presented, and is followed by our conclusions.

2. Related Works

Previous efforts in finding similar code fragments in binary executables differ in the abstraction applied on the binary and the method used for performing the match.

Venable et al. [15] present a system Vilo to search a large database of malware executables for closest matches for a given malware. They represent an executable using a vector of the n-perms [9] of its abstracted disassembly—the sequence of mnemonics. Jang et al.'s BitShred [6], like Vilo, uses abstracted disassembly. However, instead of n-perms, they use n-grams.

Cohen et al. [2] split a binary into its respective functions. But they use a cryptographic hash of the sequence of bytes (almost) as-is from the code to represent each function. Their focus is on finding matches for a given function, rather than a given binary.

Jin et al.'s work [7] is closest to ours in that they compute denotational semantics of individual blocks of code. Like Cohen et al. [2], their aim is to find similar functions. They represent a function using the hashes of the semantics of its blocks. Unlike our approach, they use the exact semantics of blocks and, hence, cannot match code segments that differ only on the choice of registers.

Zynamics BinDiff [17] addresses the inverse problem, that of finding differences in two binary executables. In their application it is assumed that the binaries being compared are successive releases of the same program. The comparison is motivated by the desire to find the location of bug fixes in the latter version. BinDiff constructs the CFG of each function with abstracted disassembly in each block, and then uses graph isomorphism to pair matching functions. The system pairs corresponding functions in the two binaries, and also pairs their corresponding basic blocks.

Gao et al.'s BinHunt [5] also has the same motivation as BinDiff—to find the differences between successive versions of a binary. Unlike Bindiff, it aims to use a program's semantics to account for differences arising due to code reordering, register renaming, and differences in compiler optimizations. Like BinDiff, BinHunt too constructs the CFG of individual functions and then uses graph isomorphism to pair matching functions. It finds matching blocks by symbolically executing them and then using a theorem prover to determine whether the blocks are semantically equivalent. In order to account for register renaming, BinHunt exhaustively tries all pair-wise comparisons to check if there exists "a permutation of the registers and variables between the two basic blocks such that all matched registers and variables contain the same value." This exhaustive exploration makes BinHunt computationally expensive.

Linger et al.'s recent work in "function extraction" (FX) [10] takes the semantic analysis of binaries to a new level. Their goal is to compute the denotational semantics of individual functions of a binary. They do so by first transforming a potentially irreducible CFG into a reducible CFG—a CFG whose loops are natural loops—and then computing the semantics of single-entry, single-exit subgraphs by starting from a CFG block and expanding outward. They encode "semantic reduction theorems" to describe the relation between high level concepts, such as matrices and vectors, in terms of flow level structures, such as memory locations. The issues faced by Gao et al. 's BinHunt resurface when using Linger et al.'s FX to determine whether two code fragments are equivalent modulo register renaming.

Our work is motivated by applications such as that described by Pfeffer et al. [12] for establishing lineage between malware variant using the evolutionary history encoded in their code. Any method for discovering malware lineage would benefit from improved ability to compare malware binaries.

`eax = 5`
`ebx = 10`
`mem(def(eax)) = def(ebx)`
`mem(def(ebx)) = def(eax)`
	`ebx = 10`
	`ecx = 5`
	`mem(def(ebx)) = def(ecx)`
	`mem(def(ecx)) = def(ebx)`
		`R1 = N1`
		`R2 = N2`
		`mem(def(R1)) = def(R2)`
		`mem(def(R2)) = def(R1)`
			`R = N`
			`mem(def(R)) = def(R)`

(a) Semantics 1	(b) Semantics 2	(c) Juice	(d) Abstracted Juice

Table 2. Challenges in ordering semantic juice

3. Method

The proposed method of extracting a binary's juice consists of the following steps:

1. Disassemble the binary.
2. Decompose the disassembled program into procedures and blocks.
3. Compute the semantics of a block.
4. Compute the juice of a block.

The first two steps are routine, and may be performed by many tools, such as IDA Pro[4] and objdump [14]. The quality of the computed semantics and the extracted juice depends on the quality of the results of these two steps. Since both disassembly and procedure boundary detection are undecidable problems, any solution is necessarily an approximation. The disassembly produced by the aforementioned tools (and others) neither guarantee sound (over approximate) nor complete (under approximate) solutions. Hence the soundness or completeness properties of semantics and juice computations discussed in this paper are predicated upon precise solutions to the first two steps.

A procedure resulting from step 2 is assumed to be represented as a control flow graph (CFG). A node of this graph is a block: a sequence of instructions such that if execution starts at the first instruction, the control will fall through to the last instruction, subject to termination. The semantics and juice is computed for individual blocks. The semantics (or juice) of a procedure is then represented as a semantic (or juice) graph that is isomorphic in structure to the CFG and whose nodes represent the semantics (or juice) of the corresponding node in the CFG.

The crux of the algorithm lies in the computation of semantics and extraction of juice of an individual block. These are described below.

3.1 Computing Semantics

A code fragment is mapped to its semantics through symbolic interpretation. The operation encoded by an assembly instruction, such as ADD, is performed on symbolic values. Whenever the operand values are known to be of type Int, the computation is performed immediately by the interpreter, thus resulting in a specific value. However, when one or both operands of a binary operator are not Int, then the operation is frozen as a structure $r_1 \,\mathrm{op}\, r_2$. Unary operators are also treated similarly. The symbolic interpreter has the following function signature.

$$\begin{aligned}\mathrm{Interpret} : \mathrm{seq}(\mathrm{Instruction}) \times \mathrm{State} &\rightarrow \mathrm{State}\\ \text{where } \mathrm{State} &= \mathrm{LValue} \rightarrow \mathrm{RValue}\end{aligned}$$

The semantics of a program denotes its affect on the store, which is represented as semantic domain State defined inductively as follows:

$$\begin{aligned}\mathrm{State} &= \mathrm{LValue} \rightarrow \mathrm{RValue}\\ \mathrm{LValue} &= \mathrm{Register} + \mathrm{Mem}\\ \mathrm{Mem} &= \mathrm{RValue} \rightarrow \mathrm{RValue}\\ \mathrm{RValue} &= \mathrm{Int} + \mathrm{def}(\mathrm{LValue})\\ &\quad + (\mathrm{RValue}\,\mathrm{op}\,\mathrm{RValue}) + (\mathrm{op}\,\mathrm{RValue})\end{aligned}$$

The set $\mathrm{Register}$ represents the set of general purpose registers, such as, eax, ebx, etc., as well as flags, such as, zf, cf, of, etc. The set Mem represents memory. An element of this set maps a RValue to RValue. The set Int represents the set of numbers. The operator 'def', an element of State, represents the previous state (being updated). The term '$r_1 \,\mathrm{op}\, r_2$', where $r_1, r_2 \in \mathrm{RValue}$, are symbolic expressions, and 'op' is a binary operator, such as, '$+$', '$-$', etc. Similarly, '$\mathrm{op}\, r_1$', where $r_1 \in \mathrm{RValue}$ represents a unary operator.

Along with symbolic interpretation, we also assume a function that performs algebraic simplification of an RValue:

$$\mathrm{Simplify} : \mathrm{RValue} \rightarrow \mathrm{RValue}$$

The function $\mathrm{Simplify}$ uses the associative, commutative, and distributive properties of operations to transform a symbolic expression to sum-of-product form [11, Chapter 12]. The commutative and associative properties are used to reorder operands of an expression into a canonical form. For instance, the expressions:

$$\begin{aligned}&(\mathrm{def}(\mathrm{eax}) + 2) + \mathrm{def}(\mathrm{ebx}),\\ &(\mathrm{def}(\mathrm{eax}) + \mathrm{def}(\mathrm{ebx})) + 2, \text{and}\\ &(2 + \mathrm{def}(\mathrm{ebx})) + \mathrm{def}(\mathrm{eax})\end{aligned}$$

are all transformed to $2 + (\mathrm{def}(\mathrm{eax}) + \mathrm{def}(\mathrm{ebx}))$.

The distributive property is used to refactor an expression so as to propagate operations of higher precedence deeper within the expression. Thus, the expression $(\mathrm{def}(\mathrm{eax}) + 2) \times \mathrm{def}(\mathrm{eax})$ is transformed to $(\mathrm{def}(\mathrm{eax}) \times \mathrm{def}(\mathrm{eax})) + (2 \times \mathrm{def}(\mathrm{eax}))$. The algebraic simplifier also includes rules of identities and zeroes of various arithmetic and logical operators. These identities and zeroes are also used to simplify expressions, such as reducing an expression of the form $(\mathrm{def}(\mathrm{eax}) - \mathrm{def}(\mathrm{eax})) \times \mathrm{def}(\mathrm{ebx})$ to the integer 0.

In addition, a linear order over RValue is used to map commutative operations to canonical form. Akin to ordering of ground terms in Prolog, the ordering is defined by using the names and arity of functions to order terms. A function with smaller arity is smaller than one with larger arity. Two functions of the same arity are ordered using lexicographic ordering of their function names. Numeric values are ordered using numeric order and are considered smaller than functions and symbols.

The semantics of a code segment 'c' is the state 's' resulting from the mapping Interpret(c, def) = s. In Table 1, the semantics is presented as 'updates' to def. The expression '$\mathrm{eax} = 5$' means that upon execution of that block of code, register eax will contain the

value 5. Using the linear order over RValue, which is also extended to LValue, a State can be represented as an ordered sequence of pairs of LValue and RValue. With the sorted representation, the semantics of two code segments can be compared in linear time with respect to the sizes of their states, or can be tested for equality in constant time using their hashes.

An algebraic simplifier's ability to map equivalent code to the same semantic structure depends on, among other things, whether the rewrite rules used by $\mathrm{Simplify}$ are confluent. Completing a set of rewrite rules to make the set confluent is an undecidable problem. Thus, the simplifier does not in any way bypass undecidability. However, as articulated by Linger et al. [10], in spite of its limitation, a simplifier based on (known) algebraic equalities can normalize a large set of expressions, so as to be useful on real-world code.

3.2 Extracting Juice

The purpose of extracting juice is motivated by the desire to efficiently determine whether two code segments are semantically equivalent modulo renaming of registers. Previous works have addressed this challenge by generating all possible permutations of renamed code segments and comparing their semantics [5]. While such a method is safe, it is computationally expensive and does not scale for the problem of searching for semantically similar code in a large collection of malware. On the other hand, the semantics of blocks can directly be used to create a scalable search algorithm [7], but such an algorithm will not account for register renaming, and hence be too strict.

As stated earlier, juice is a generalization of semantics along with type and algebraic constraints. Whereas semantics consists of ground terms, juice terms may contain logical variables. The generalization of semantics to juice may be performed by consistently replacing register names with logical variables. The replacement is consistent in that two occurrences of the same register name are always replaced by the same variable. In addition to abstracting the registers used, one may also abstract the literal constants. In the example of Table 1, the semantics '$\mathrm{ebx} = \mathrm{def}(\mathrm{ebx}) \times 5 + 20$' generalizes to '$B = \mathrm{def}(B) \times \mathrm{N1} + \mathrm{N2}$' by consistently renaming its registers and literal constants. The type of the variables introduced follows directly from the type of the term they replace. Since the logical variable B replaces the 32-bit register eax, it follows that B is of type $\mathrm{reg32}$. Similarly, it follows that $\mathrm{N1}$ and $\mathrm{N2}$ are of type Int.

The problem then remains how does one generate the algebraic constraints between the logical variables, for instance, the constraint '$\mathrm{N2} = \mathrm{N1} \times \mathrm{N3}$' in Table 1. We present an innovative approach for generating such algebraic constraints. The key idea is to augment the symbolic interpreter to track the simplifications it performs. In our example, the term 20 in the expression '$\mathrm{def}(\mathrm{ebx}) \times 5 + 20$' results from the immediate simplification of the expression 5×4, which in turn follows from the distributive property of multiplication. In this example, the interpreter will annotate the semantics with the tautology '$20 = 5 \times 4$'. Then, when extracting juice, the annotations are also generalized along with the semantics. Thus, the term 20 is replaced by $\mathrm{N2}$ and 5 by $\mathrm{N1}$ in both the annotation and the semantics, yielding the constraint '$\mathrm{N2} = \mathrm{N1} \times \mathrm{N3}$'.

Having computed the juice for code fragments, the question still remains: how does one efficiently determine whether two fragments have the same juice? As done for semantics, it would be ideal if equivalent terms can be mapped to a canonical form. However, the existence of logical variables in juice terms poses a challenge. One possibility is to name these variables in the order in which substitutions are performed, and use the resulting order for comparison. Then the juice terms can be ordered using this linear order. When two such ordered juice terms match, the corresponding code fragments will be equivalent, modulo renaming of variables and literal constants. However, it is also possible that a different ordering of variables may lead some other pairs of terms to match, and thus correctly identify other sets of equivalent code fragments. Thus, even though an arbitrary order imposed on logical variables will correctly identify equivalent code fragments, it may also miss some equivalences.

Since an arbitrary order on logical variable is not satisfactory, we prefer to assume treat variables as unordered. This leads to a partial order on juice in which two equations in the juice of a code segment cannot be ordered linearly if and only if they are variants of each other, i.e., they are identical except for their variables. Table 2 presents an example to illustrate this. The first two columns of the table contain the semantics of two code fragments. It is evident that the semantics are equivalent, except for the choice of registers. The semantics in column (a) can be transformed to that in column (b) by replacing the register eax by ecx. The two semantics naturally result in juice that differs only in the logical variables. One such juice is given in column (c). Whereas the terms in the semantics could be linearly ordered, the same is not true of the juice. The two terms '$R1 = \mathrm{N1}$' and '$R2 = \mathrm{N2}$' cannot be ordered. The same is true of the other two terms.

Table 2 column (d) shows another abstraction of juice that trades safety for an increase in the set of code fragments that may be deemed equivalent. This abstraction takes advantage of the observation mentioned above that only mutually variant terms in juice cannot be ordered. In this abstraction, the variant terms in the juice are unified, yielding a structure that is linearly ordered. As is evident from the example, such a generalization may result in a significant loss and may not always be prudent.

4. Preliminary Results

We have developed a system BinJuice that implements the method described in the previous section. The symbolic interpreter and simplification algorithm of BinJuice are written in SWI-Prolog and its frontend in Python. The system takes as input disassembled code, such as that produced by objdump [14]. The Python frontend parses the disassembly, partitions it into procedures, constructs their CFGs, and then feeds each block of code to the Prolog backend over a http connection. The Prolog backend computes each block's semantics and extracts its juice. The juice consists of four components: the generalized semantics, the type constraints, the algebraic constraints, and the generalized code fragment. The generalized code fragment is produced by generalizing the original code using the same variable substitutions as those used for the generalized semantics.

We have studied the efficacy of BinJuice's semantics computation and juice extraction capability using two sets of data: 1) variants of Win32.Evol, a metamorphic virus and 2) ten versions each of two benign programs downloaded from Github. The first set of programs represent variants that may be generated by post-compile obfuscation transformation and the second set represents versions resulting from normal code evolution.

The purpose of the first study, that with Win32.Evol, was to determine whether BinJuice can aid in locating matching code blocks in automatically generated variants. The purpose of the second study was to determine whether use of juice improved upon the similarity computation as compared with using semantics or n-perms.

The Win32.Evol variants were acquired from a previous study [16]. In that study, the malware was used to infect eight Microsoft Windows 'goat' executables. The infected executables were then run again to reinfect the goat files. Since the malware is metamorphic, at each infection it introduces a modified copy of its code and

```
push(ebp)
mov(ebp,esp)
sub(esp,16)
mov(eax,dptr(ebp))
push(edi)
push(ebx)
mov(ebx,ebp)
mov(edi,ebx)
pop(ebx)
push(ebx)
mov(ebx,46)
sub(edi,ebx)
pop(ebx)
mov(dptr(edi+42),eax)
pop(edi)
lea(eax,wptr(ebp-16))
push(esi)
mov(esi,1634038339)
mov(dptr(eax),esi)
pop(esi)
push(esi)
mov(esi,32509012)
add(esi,1733712160)
mov(dptr(eax+4),esi)
pop(esi)
push(edx)
mov(edx,4285804)
mov(dptr(eax+8),edx)
pop(edx)
push(eax)
```

(a) Variant 1

```
push(ebp)
mov(ebp,esp)
sub(esp,16)
mov(eax,dptr(ebp))
mov(dptr(ebp-4),eax)
lea(eax,wptr(ebp-16))
mov(dptr(eax),1634038339)
mov(dptr(eax + 4),1766221172)
mov(dptr(eax + 8),4285804)
push(eax)
```

(b) Variant 2

```
 eax = -20 + def(esp)
 ebp = -4 + def(esp)
 esp = -24 + def(esp)
 memdw(-24 + def(esp)) = -20 + def(esp)
 memdw(-20 + def(esp)) = 1634038339
 memdw(-16 + def(esp)) = 1766221172
 memdw(-12 + def(esp)) = 4285804
 memdw(-8 + def(esp)) = def(ebp)
 memdw(-4 + def(esp)) = def(ebp)

Simplifications performed
 1766221172 = 32509012 + 1733712160
 -50 = -4 - 46
 -20 = -4 - 16
 -16 = -20 + 4
 -12 = -20 + 8
 -8 = -50 + 42
```

(c) Semantics

```
 A = -N1 + def(B)
 C = -N2 + def(B)
 B = -N3 + def(B)
 memdw(-N3 + def(B)) = -N1 + def(B)
 memdw(-N1 + def(B)) = N4
 memdw(-N5 + def(B)) = N6
 memdw(-N7 + def(B)) = N8
 memdw(-N9 + def(B)) = def(C)
 memdw(-N2 + def(B)) = def(C)
where
 N6 = N10 + N11
 -N12 = -N2 - N13
 -N1 = -N2 - N5
 -N5 = -N1 + N2
 -N7 = -N1 + N9
 -N9 = -N12 + N14
```

(d) Juice

Table 3. Two variant code segments from Win32.Evol and their semantic juice

creates a new variant. Some goat files were reinfected up to six times, creating up to six 'generations' of infection.

Table 3 gives an example of semantic juice extraction for code transformed using the metamorphic engine of Win32.Evol. The code segments in columns (a) and (b) are for corresponding blocks of the two variants of the malware and are semantically equivalent. Column (c) shows their (identical) extracted semantics, and it also shows the tautologies used in algebraic simplification for Variant 1. The example is indicative of what we find for all the variants of Win32.Evol. The semantics can be used to identify corresponding blocks of code for variants even though the code for the blocks may be transformed. Column (d) shows the corresponding juice along with the algebraic constraints.

Our experience with Win32.Evol showed that the semantics and juice computed by BinJuice can be used to accurately pair corresponding blocks so long as they were transformed by equivalence preserving transformations. Win32.Evol contains transformations that, while preserving the semantics of the overall program, do not preserve the semantics of a block. For instance, it has transformations that introduces computation on registers that are known to be dead at the end of the block. In such a situation, the intersection of the semantics (juice) of the corresponding blocks always yields the correct semantics (juice), though the two semantics (juice) are not structurally equal.

In the second experiment we downloaded ten versions of tinyrb and mcmap programs from GitHub. Tinyrb is a "tiny and fast subset of Ruby Virtual Machine" and mcmap is "a tiny map visualizer for Minecraft." We compiled the sources using gcc and generated x86, 32-bit binaries. These were then stripped and used to extract the following features: n-perms, semantic hashes, and juice hashes. These features were then used in the MAAGI system to construct lineage – the evolutionary relationships between the versions [12].

Our preliminary results show that the use of semantics or juice did not significantly improve upon the lineage computed using n-perms. Though these results are not exciting, they are also not conclusive. Since we used the same version of the compiler with the same optimization levels, the unchanged functions between two successive versions had exactly the same code. Thus, the similarity computed using n-perms was just as good as that using semantics or juice.

Taken together the results of the two preliminary studies indicate that the semantics and juice computed using BinJuice would be effective in finding similar code fragments in a repository of malware where post-compile obfuscators are used to generate variants. However, for a repository of benign programs, the lower cost features generated using n-perms or n-grams may be sufficient.

5. Limitations

Any method for finding semantically similar code fragments is fundamentally limited by Rice's Theorem. So it is a given that no computer program can precisely identify equivalent (or similar) code fragments between all pairs of programs. Yet, it is instructive to identify obvious limitations that may influence the results in practical real-world programs. These limitations also identify weaknesses a malware author may exploit to defeat a system employing the method presented.

The most significant limitation of the proposed method is that it is centered around semantics of basic blocks. As a result semantically equivalent programs that differ in how computation is spread across the basic blocks would not have similar block semantics or juice. Such differences in content and structure of blocks may arise due to differences in compilers and optimization levels. The differences may also be explicitly forced by post-compile obfuscators.

The block-centric limitation may be overcome by computing the semantics for single-entry, single-exit subgraphs, as done by Linger et al. [10]. The benefit comes at a cost, since the number of such subgraphs is quadratic on the number of blocks. Another alternative may be to use semantics of n nodes along a path in the CFG, akin to n-grams. Though this may counter some optimizations and obfuscations, it is sensitive to even very small changes in semantics due to small evolutionary changes or due to dead code insertion by obfuscators. As a result, one may need to use inexact matches on semantics or juice, thereby making the operation expensive.

As with data flow analysis [11], aliases bring a new set of challenges. Two code segments may be equivalent under aliasing, for instance, when two registers hold the address for the same memory location. Determining such equivalences may require the need of theorem provers, and may not be amenable to simple structural comparisons. However, the use of such equivalences may also produce false matches, causing segments of code to be considered equivalent under aliasing even though such aliasing may not arise in the program.

Specific methods of defeating semantics-based (and other) methods of finding similar code fragments may be found in Collberg and Nagra's [3] treatise on software obfuscation and deobfuscation.

6. Conclusions

This paper introduces two improvements to prior works on the use of symbolic interpretation to find similar code fragments. First, it canonicalizes the terms representing the semantics using algebraic simplification such that equivalence can be determined using structural equality. This offers a very fast method for comparing the semantics of code blocks, albeit subject to the limitations of the algebraic simplifier. Second, it introduces a novel concept: juice—a generalization of the semantics. Juice is computed by replacing register names and literal constants in the semantics by typed variables. More importantly, juice also maintains algebraic constraints relating those variables. Thus, juice serves as a template of the semantics and may be used to match code fragments that differ due to register renaming and also choice of constants.

This work is motivated by the need to find similar code fragments in a large repository of binaries, an application in which a fast method to match semantics (or any abstraction) is a prerequisite. Unlike semantic terms, juice terms cannot be linearly ordered and thus cannot be compared using their hashes. We present further abstraction of the juice so as to impose a linear order, thereby enabling fast comparison. The abstraction comes at a loss of accuracy whose effect on real-world applications needs to be determined empirically.

7. Acknowledgments

The authors thank Jacob Ouellette and Avi Pfeffer of Charles Rivers Analytics, Cambridge, MA for their participation in the experiments using BinJuice. This material is based upon work supported by the Air Force Research Laboratory, Rome, NY and the Defense Advanced Research Program Agency under Award No. FA8750-10-C-0171, the Air Force Office of Scientific Research under Award No. FA9550-09-1-0715, and the Air Force Research Laboratory, Rome NY under Contract No. FA8750-12-C-0144.

References

[1] Black Duck: Open source management software & consulting, `https://www.blackducksoftware.com/`

[2] Cohen, C., Havrilla, J.S.: Function hashing for malicious code analysis. In: CERT Research Annual Report 2009, pp. 26–29. Software Engineering Institute, Carnegie Mellon University (2010), `http://www.cert.org/research/2009research-report.pdf`

[3] Collberg, C., Nagra, J.: Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Pearson Education (2010)

[4] Eagle, C.: The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler. No Starch Press (2008)

[5] Gao, D., Reiter, M.K., Song, D.: BinHunt: automatically finding semantic differences in binary programs. In: Chen, L., Ryan, M.D., Wang, G. (eds.) Information and Communications Security, pp. 238–255. No. 5308 in Lecture Notes in Computer Science, Springer Berlin Heidelberg (Jan 2008), `http://link.springer.com/chapter/10.1007/978-3-540-88625-9_16`

[6] Jang, J., Brumley, D., Venkataraman, S.: BitShred: feature hashing malware for scalable triage and semantic analysis. In: Proceedings of the 18th ACM conference on Computer and communications security. p. 309320. CCS '11, ACM, New York, NY, USA (2011), `http://doi.acm.org/10.1145/2046707.2046742`

[7] Jin, W., Chaki, S., Cohen, C., Gurfinkel, A., Havrilla, J., Hines, C., Narasimhan, P.: Binary function clustering using semantic hashes. In: Proceedings of the 11th International Conference on Machine Learning and Applications (ICMLA), (Dec 2012), `http://www.contrib.andrew.cmu.edu/~schaki/publications/ICMLA-2012.html`

[8] Jin, W., Hines, C., Cohen, C., Narasimhan, P.: A scalable search index for binary files. In: Proceedings of 7th International Conference on Malicious and Unwanted Software (Malware 2012). Fajardo, Puerto Rico (Oct 2012)

[9] Karim, M., Walenstein, A., Lakhotia, A., Parida, L.: Malware phylogeny generation using permutations of code. Journal in Computer Virology 1(1), 13–23 (2005)

[10] Linger, R., Daly, T., Pleszkoch, M.: Function extraction (FX) research for computation of software behavior: 2010 development and application of semantic reduction theorems for behavior analysis. Technical Report CMU/SEI-2011-TR-009, Carnegie Mellon University, Software Engineering Institute (Feb 2011), `http://www.cert.org/archive/pdf/11tr009.pdf`

[11] Muchnick, S.S.: Advanced Compiler Design Implementation. Morgan Kaufmann (1997)

[12] Pfeffer, A., Call, C., Chamberlain, J., Kellogg, L., Ouellette, J., Patten, T., Zacharias, G., Lakhotia, A., Golconda, S., Bay, J., Hall, R., Scofield, D.: Malware analysis and attribution using genetic information. In: Proceedings of the 7th IEEE International Conference on Malicious and Unwanted Software (MALWARE 2012). IEEE Computer Society Press, Fajardo, Puerto Rico (Oct 2012)

[13] Porst, S.: Comparing different versions of SDBot using SABRE BinDiff v1.7 (Sep 2005), `http://www.the-interweb.com/bdump/malware/bindiff.pdf`

[14] The GNU Project: GNU binutils, `http://www.gnu.org/software/binutils/binutils.html`

[15] Venable, M., Walenstein, A., Hayes, M., Thompson, C., Lakhotia, A.: Vilo: a shield in the malware variation battle. Virus Bulletin pp. 5–10 (2007)

[16] Walenstein, A., Mathur, R., Chouchane, M.R., Lakhotia, A.: Constructing malware normalizers using term rewriting. Journal in Computer Virology 4(4), 307–322 (Nov 2008), `http://link.springer.com/article/10.1007/s11416-008-0081-5`

[17] Zynamics: BinDiff 3.2 manual. http://www.zynamics.com/bindiff/manual/, `http://www.zynamics.com/bindiff/manual/`