Polymorphic Attacks against Sequence-based Software Birthmarks

Hyoungshick Kim
University of British Columbia
hyoung@ece.ubc.ca

Wei Ming Khoo
University of Cambridge
wmk26@cam.ac.uk

Pietro Liò
University of Cambridge
pl219@cam.ac.uk

***Abstract*—Sequence alignment algorithms have recently found a use in detecting code clones, software plagiarism, code theft, and polymorphic malware. This approach involves extracting birthmarks, in this case sequences, from programs and comparing them using sequence alignment, a procedure which has been intensively studied in the field of bioinformatics. This idea seems promising. However, we have shown that an attacker can evade detection by considering the positions of inserted dummy code and/or the frequency of function calls. Moreover, we found that randomly inserting and deleting symbols in the sequence was ineffective. By using birthmark sequences extracted from actual malicious and benign programs, we found that the most effective strategy was to use a hybrid approach incorporating "non-consecutive insertion" and "highest frequency deletion". We also discuss the implementation costs of such attacks and propose using non-determinism through concurrent programming as an alternative evasion strategy.**

I. Introduction

A software birthmark [15] is an inherent characteristic of a program that can be used to identify the program. By comparing birthmarks of programs, we can detect whether a program is a copy of another program or not. In order to provide practically usable software birthmarks, two major problems are considered: (i) what characteristics of a program can be used for its software birthmark, and (ii) how to efficiently compare the software birthmarks.

For problem (i), it is not easy to find useful characteristics of a program for the purpose of software birthmark – attackers will naturally try to perform semantics preserving transformations (such as optimization and obfuscation) in order to remove or distort the software birthmark of the original program. For example, it is not proper to use the hash value of the program code, which is usually used to verify the integrity of the program since the hash value of the program would be useless against semantics preserving transformations. A reasonable solution is to use a software birthmark to capture the invariable properties of a program (e.g. its run time behaviour) against semantics preserving transformations. Also, it seems desirable to use a software birthmark which has to be taken over a whole program rather than some specific parts of the program in order to maximize the attacker's cost of modifying the original program code. We use the term "polymorphic attacks" to represent semantics preserving transformations which are maliciously performed by an attacker to distort the software birthmark of the original program.

For problem (ii), the most popular approach is to reduce the software birthmarks of programs to sequences, and to compare them by computing the similarity between the sequences [20], [17], [22], [18], [9]. This is because the *sequence alignment* problem is already a well known and studied problem in bioinformatics; there are a number of algorithms and techniques for measuring similarity between sequences to identify biologically significant relationships. That is, it enables us to apply conventional sequence alignment techniques to the comparison of software birthmarks between programs. At first glance, this seems rather promising. However, we argue that it is not easy to achieve a reasonable level of robustness against attacks that target sequence alignment algorithms. While sequence alignment may cope well with accidental DNA and protein mutations, we show that it can be vulnerable to specific insertion and deletion schemes (Section III).

Until now, software birthmark research has largely focused on the first problem [15], [20], [16], [19], [21], [22], [18]. In many cases, researchers claim that their software birthmarks perform well by showing that the birthmarks of example programs are mostly resilient against a few existing obfuscation tools. However, we note that these tools are not particularly designed to address intentional and targeted transformations.

Our key contributions can be summarised as follows:

- As far as we are aware, we are the first to provide an evaluation of attack strategies targeting sequence-based birthmarking schemes.
- Second, we show that the intuitive strategies of adding noise by inserting dummy codes or replacing existing codes in a random manner are ineffective even at high levels of insertions (200%) or deletions (80%). Instead we show that a more effective attack strategy is to consider the locations of the insertion points in the birthmark and the frequencies of API calls or instructions. (see Section 4).
- Third, we discuss the potential implementation issues of such attacks and propose using non-determinism through concurrent programming as an alternative evasion strategy. In practice, inserting or deleting API calls is not a trivial task. An alternative is to exploit the use of non-determinism through concurrent programming.

We demonstrate the effectiveness of this technique (see Section 5).

In the rest of this article, we will discuss the limitation of sequence alignment techniques as tools for comparing software birthmarks. We set out to quantify the performance of sequence alignment algorithms against reasonable polymorphic attacks; we demonstrate this empirically on real programs.

II. Computing similarity between sequence-based software birthmarks

The method of computing the similarity between software birthmarks depends on the type of software birthmark being used. For example, when birthmarks of programs are in the form of graphs, we need to compute the similarity between graph structures to determine their similarity.

In this paper, we focus only on sequence-based software birthmarks that have recently been proposed as a promising birthmarking scheme. In this scheme, the birthmark of a program is represented as a sequence consisting of API calls [20], [19], [17], [22], [9] and/or instructions [18] to capture the runtime behaviour of a program. Without loss of generality, we use an execution trace of the instructions and API calls used in a program as its software birthmark. Instruction and API call traces can be efficiently generated on most modern operating systems either through hardware or software. We formally define the software birthmark `mark(`P`)` of program P to be a sequence of symbols from a finite alphabet set $\Sigma = \{a_1, \cdots, a_k\}$ representing an API or instruction. In other words, the instructions and API calls are mapped to symbols in Σ and the trace of the instructions and API calls used in a program is represented as a sequence consisting of symbols in Σ.

An example is illustrated in Table I. When we ignore the arguments, the software birthmark of Table I can be represented as a sequence of letters as follows: (ABCDEDE).

Table I
AN EXAMPLE OF AN EXECUTION TRACE OF API CALLS TAKEN FROM SKYHOO. WHEN WE IGNORE THE ARGUMENTS, THE TRACE CAN BE REPRESENTED AS THE SEQUENCE OF (ABCDEDE).

kernel32.dll:OutputDebugStringA	A
kernel32.dll:GetModuleHandleA	B
ADVAPI32.DLL:OpenSCManagerA	C
kernel32.dll:CreateFileA	D
ADVAPI32.DLL:OpenServiceA	E
kernel32.dll:CreateFileA	D
ADVAPI32.DLL:OpenServiceA	E

To check whether program P is a copy (or variant) of program Q, we will compute the similarity score between the two sequences `mark(`P`)` and `mark(`Q`)` which are generated from P and Q, respectively, and then test whether the computed score exceeds a given significance level called *similarity threshold*.

The best method to compute the similarity between sequences is to use sequence alignment algorithms which are frequently used in bioinformatics to analyse biological sequence data – sequence alignment is a process of arranging two or more sequences placed one below each other with a scoring system which rewards with a positive score those positions at which the sequences agree and with a negative score (a penalty) those positions where there is a disagreement ('mutation') and insertion of a blank ('gap'). With scoring schemes sequence alignment algorithms are categorized into three classes: global alignment, local alignment, and semi-global alignment algorithms. A global alignment algorithm calculates the similarity using the whole sequences; A local alignment algorithm is used to find maximum subsequence matching; A semi-global alignment algorithm is a modification of the global alignment where the leading and terminal gaps are not penalized [2]. Several software birthmarks [17], [18], [9] were proposed based on sequence alignment algorithms.

We here use the global alignment with affine gap costs where the first gap character in a gap incurs a substantial "gap opening" cost, and each subsequent gap character incurs a somewhat lesser "gap extension" cost [2]. It has much finer granularity than any fixed gap scoring model which assumes that the "gap opening" cost is the same as the "gap extension" cost. Affine gap scores generally work fairly well to identify biologically significant relationships for DNA or protein sequences. We will optimize the parameters for affine gap scores over the software birthmarks of real programs and use them to evaluate the effectiveness of the attack strategies in Section IV.

III. Polymorphic attacks

In sequence-based software birthmarks, a symbol of a birthmark corresponds to an API call or instruction. In order to remove or distort the software birthmark of the original program, an attacker can modify the program by performing semantics preserving transformations that add new API calls or instructions, delete and/or replace existing ones that do not affect the functionality of the program without severe penalties in its efficiency. We consider three representative *polymorphic attacks*: "insertion of bogus symbols", "deletion of existing symbols" and "hybrid of insertion and deletion".

A. Insertion of bogus symbols

Adding bogus symbols is accomplished by using several obfuscation techniques. For example, when a software birthmark of a program is generated from the control flow graph of the program, opaque predicates [5] can be used to insert dead pieces of code that will be converted to bogus symbols – a predicate is called *opaque* if its outcome is known at obfuscation time but is difficult for the deobfuscator to deduce. The precise static analysis of opaque predicates is generally either undecidable or extremely time consuming [5].

Given a sequence-based software birthmark mark(P) of length n, a finite alphabet set $\Sigma = \{a_1, \cdots, a_k\}$ to denote all API calls or instructions and the ratio x_i of newly added bogus symbols where $0 < x_i$, the *insertion attack* is to add $\lfloor n \cdot x_i \rfloor$ symbols which are randomly selected from Σ into the birthmark mark(P).

The simplest attack strategy is to place $\lfloor n \cdot x_i \rfloor$ bogus symbols into random positions of mark(P). We call this strategy "random insertion". Some bogus symbols can be adjacent to each other when we add bogus symbols randomly. In this case, the random insertion attack strategy may be ineffective since affine gap penalties in sequence alignment algorithms are commonly used so that a sequence of gaps is assigned less penalty than treating them as individual gaps.

We propose instead "non-consecutive insertion"; if we add bogus symbols in a non-consecutive manner, this may cut the similarity score significantly since the number of individual gaps increases. We describe this process in detail. For a birthmark mark(P), $\lfloor n \cdot x_i \rfloor$ bogus symbols are inserted as follows:

1) Pick $\lfloor n \cdot x_i \rfloor$ symbols randomly from Σ and divide them into k non-empty subsequences where $k = \min\{\lfloor n \cdot x_i \rfloor, n\}$. Let $B = \{b_1, \cdots b_k\}$ be the set of these subsequences.
2) Divide mark(P) into k non-empty subsequences described as $\sigma_1, \cdots \sigma_k$ where σ_i represents the ith subsequences in mark(P).
3) For ith iteration from $i = 1$ to k, choose b_i from B and add it into the first position of subsequence $\sigma_{((k-i+1) \text{ MOD } k)}$ where $\sigma_k = \sigma_0$.

We use INS(R) and INS(N) to denote "random insertion" and "non-consecutive insertion" attack strategies, respectively. We will test the performance of a well-trained sequence alignment algorithm with varying x_i in Section IV.

B. Deletion of existing symbols

Removing existing symbols in the software birthmark of a program is accomplished by replacing existing API calls or instructions in the program with the alternative API calls or instructions that cannot be mapped onto Σ. In most programming languages, a function can be implemented in several ways. For example, the standard library functions such as fopen or fgets in the C programming language can be replaced by an equivalent Win32 API call to open a file with an access mode (read or write) on the MS Windows platform. If the Win32 API is not defined as an symbols in the sequence alignment scheme, it gives the effects of the deletion of the symbols representing fopen or fgets in the software birthmark.

Given a sequence-based software birthmark mark(P) consisting of m different symbol types and the ratio x_d of symbol types to be deleted in the birthmark mark(P) where $0 < x_d < 1$, the *deletion attack* is to remove $\lfloor m \cdot x_d \rfloor$ different types of symbol in the birthmark mark(P). We consider the two attack strategies: "random deletion" and "highest frequency deletion". In random deletion, we select $\lfloor m \cdot x_d \rfloor$ different symbol types randomly from the m symbol types in mark(P) and simply remove them whereas the "high frequency deletion" attack is more strategical: remove the corresponding symbol types which have top-$\lfloor m \cdot x_d \rfloor$ highest frequency values in mark(P).

We use DEL(R) and DEL(H) to denote "random deletion" and "highest frequency deletion" attack strategies, respectively. We will test the performance of a well-trained sequence alignment algorithm with varying x_d in Section IV.

C. Hybrid of insertion and deletion

In practice, a reasonable attack strategy is to use a combination of deletions and insertions. Here we use the deletion-then-insertion scheme considering the four possible combinations of the two deletion attacks in Section III-B and the two insertion attacks in Section III-A by applying first an deletion attack strategy between DEL(R) and DEL(H) and then an insertion attack strategy between INS(R) and INS(N). We note hybrid attack is a more general attack type – it is more flexible than merely using insertions or deletions alone and it can be controlled via x_d and x_i.

We use HYB(RR), HYB(RN), HYB(HR) and HYB(HN) to denote these attack strategies, respectively where HYB(XY) is a combination of DEL(X) and INS(Y). We will test the performance of a well-trained sequence alignment algorithm with varying x_d and x_i in Section IV.

IV. Experiments

We evaluate the performance of the polymorphic attacks described in Section III through the intensive simulation with real programs to compare the birthmark of each program and the birthmarks modified by the simulated attacks. We use the sequence of API calls on an important execution path in a program as its software birthmark. The use of API calls a very promising trend in defining software birthmarks [20], [10], [19], [17], [22], [9] since it is generally difficult to change such dynamic behaviour of programs. We used Pin [13] to capture the API calls executed in programs.

A. Parameter selection for sequence alignment algorithm

Before discussing the performance of polymorphic attacks, we first need to optimize parameters (the gap opening cost, the gap extension cost, the mismatch cost) for the sequence alignment algorithm. To select the parameters, we experimentally measure the performance of the sequence alignment algorithm with varying the parameters. We define a True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN) as:

- TP – when the birthmarks are generated from the same malware family (or program), they are correctly identified as the same;

- FP – when the birthmarks are generated from the same program, they are incorrectly identified as different;
- TN – when the birthmarks are generated from different programs, they are correctly identified as different;
- FN – when the birthmarks are generated from different programs, they are incorrectly identified as the same.

The performance of a sequence alignment algorithm can be evaluated by using the following three measurements:

- *Accuracy* – the rate of the testing birthmarks correctly identified ($\frac{TP+TN}{TP+TN+FP+FN}$);
- *Sensitivity* – the rate of true positive ($\frac{TP}{TP+FN}$);
- *Specificity* – the rate of true negative ($\frac{TN}{TN+FP}$).

We aim to choose the parameters to maximize the *accuracy*, *sensitivity* and *specificity* of the sequence alignment algorithm. We used the 49 software birthmarks generated from the two malware families (18 FakeAV-DO[1] and 20 Skyhoo[2] binaries), plus the four normal programs (3 triangle[3], 2 7zip, 3 WinSCP and 3 Notepad execution traces in each program). For each pair of software birthmarks, we vary the gap opening cost C^{open} from -1.0 to -0.5, the gap extension cost $C^{\text{extension}}$ from -0.2 to 0, and the mismatch cost C^{mismatch} from -1.0 to -0.8 while fixing the *similarity threshold* as 0 and the match cost as 1.0 – the testing birthmarks are from the same malware family (or program) when the pairwise alignment score is greater than or equal to the *similarity threshold*.

For the evaluation of the polymorphic attacks, we choose the gap opening cost as -0.9, the gap extension cost as -0.05, the mismatch cost as -1.0, and the match cost as 1.0 since these parameters shows the best performance of the sequence alignment algorithm with the programs used in our experiments. In the next section, we use these sequence alignment parameters to evaluate the performance of the polymorphic attacks discussed in Section III.

B. Performance results of polymorphic attacks

We evaluate the polymorphic attacks discussed in Section III on the two representative programs (`triangle` and `notepad`) and the two malware (`fak-do` and `skyhoo`). Some information about their birthmarks is described in Table II.

1) Results of insertion attacks: For the simulation of insertion attacks, we use the 370 API calls used in the 49 birthmarks in Section IV-A as Σ.

We repeat an insertion attack strategy ("random insertion" or "non-consecutive insertion") 100 times with varying the insertion ratio x_i of newly added bogus symbols from 0.0 to 2.0, respectively, for each of the four programs in Table II and compute the detection rate and the average alignment score of the sequence alignment algorithm over the modified 100 variants. Figure 1 shows how the detection rates and the average sequence alignment scores change with x_i.

From this figure, we can see that only `INS(N)` performed well: the detection rates for all programs fall to zero when $x_i \geq 1.2$ while `INS(R)` is not effective—even when $x_i = 2.0$ the detection rates remained unchanged. Inherently, there exists a relationship between the detection rates of the sequence alignment algorithm and its average alignment score: The detection rates for all programs start to drop dramatically when the average sequence alignment score falls below 0. Figure 1 also shows how the average alignment score change with x_i. This figure explains why `INS(R)` is not effective from the attacker's point of view; the average alignment score is still greater than 0 even when $x_i = 2.0$. The curve of `INS(R)` has a gentle slope from around $x_i = 0.5$ while `INS(N)` plunges towards 0 until $x_i = 1.0$.

2) Results of deletion attacks: We similarly repeated the deletion attack strategy ("random deletion" and "highest frequency deletion") 100 times by varying the deletion ratio x_d of symbol types to be deleted from 0.0 to 0.8 for the four programs in Table II and computed the detection rate over the 100 modified variants (Figure 2). There appeared to be a stepwise decreasing pattern in malware `fak-do` and `skyhoo` due to the small number of symbol types.

Unlike insertion attacks, the performance of deletion attacks was quite different between programs. For example, `DEL(H)` performed well for the `notepad` while this attack strategy was not effective for `triangle`. We surmise that the ratio of n (the length of a software birthmark) to m (the number of symbols in the birthmark) may explain this difference. The software birthmark of `notepad` had the highest n/m ratio with about 11.2099. The software birthmark of `triangle` had relatively diverse symbol types with the lowest n/m ratio of 2. This is reasonable; the replacement of a few API calls is not a good strategy when they are highly diverse. Therefore, the n/m ratio of the target program should be taken into account when considering a deletion scheme since the replacement of API calls would incur a high cost to attackers but yet might not be as effective.

Program	n	m	n/m	Entropy
`triangle`	42	21	2.0000	4.0736
`notepad`	908	81	11.2099	4.1500
`fak-do`	33	6	5.5000	1.8937
`skyhoo`	17	4	4.2500	1.5197

Table II
SUMMARY OF PROGRAMS USED IN THE SIMULATION. HERE n IS THE BIRTHMARK LENGTH, m IS THE NUMBER OF UNIQUE SYMBOLS AND THE ENTROPY IS $H(X) = -\sum_{i=1}^{m} p_i \log_2 p_i$ WHERE X IS A RANDOM VARIABLE DRAWN FROM A FINITE DISTRIBUTION $p_i = P(X = a_i)$, $i \in [1, m]$.

[1]The FakeAV-DO family is a trojan family that displays fake alerts that coax users into buying rogue antivirus products.

[2]The Skyhoo family is a computer worm family that is mainly being propagated by means of Yahoo instant messaging and Skype programs.

[3]This program takes the three lengths of the sides of a triangle, and returns what kind of triangle it is (see Figure 1 in [9]).

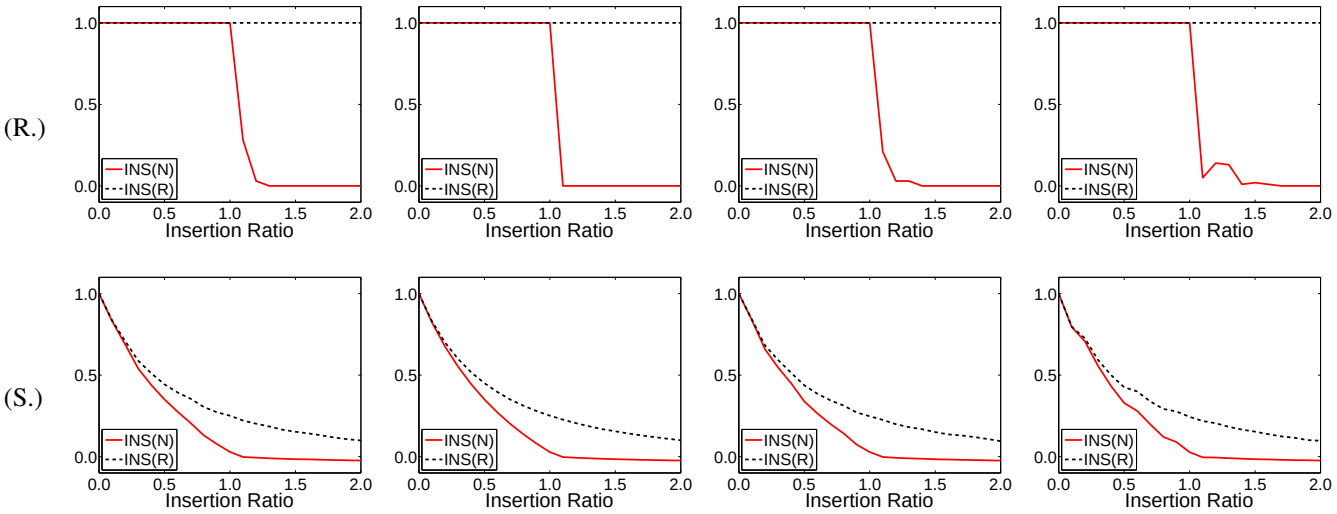


Figure 1. The detection rates (R.) and the average scores (S.) of sequence alignment algorithm against random insertions, INS(R), and non-consecutive insertions, INS(N), for insertion ratios x_i of between 0.0 to 2.0. The $\lfloor n \cdot x_i \rfloor$ symbols are added into the birthmark of each program (a–d) where n is the length of the birthmark.

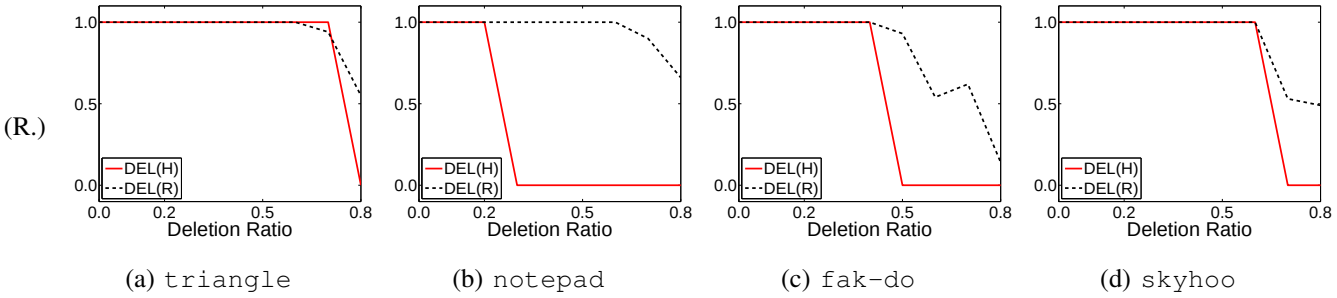


Figure 2. The detection rates (R.) of sequence alignment against random deletion, DEL(R), and highest frequency deletion, DEL(H), for deletion ratios x_d of between 0.0 to 0.8. The $\lfloor m \cdot x_d \rfloor$ symbol types are removed from the birthmark of each program (a–d) where m is the number of symbol types in the birthmark.

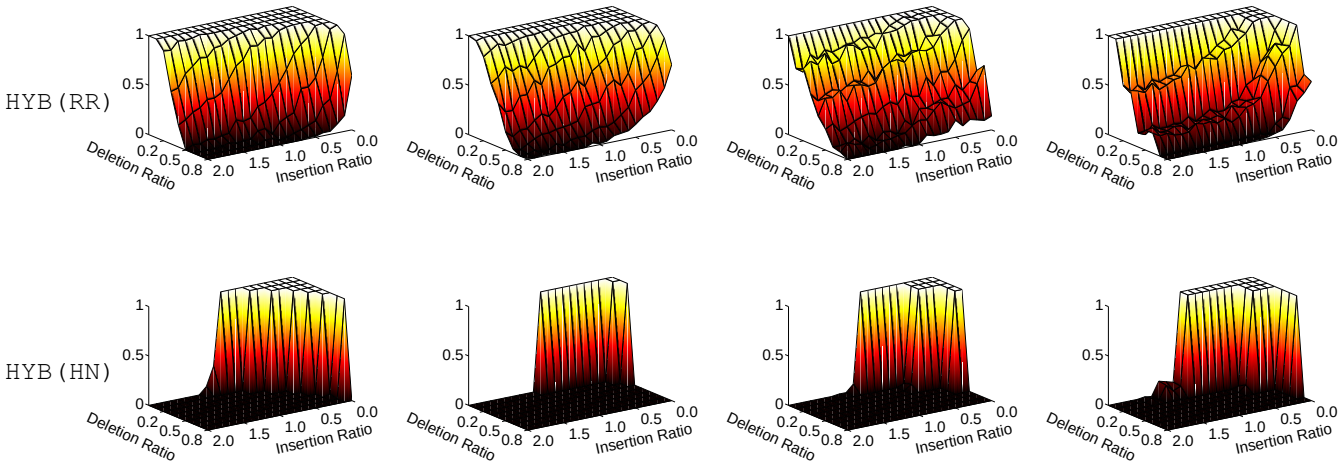
3) Results of hybrid attacks: Figure 3 shows how the detection rates of the sequence alignment algorithm varied with x_d and x_i. Our experimental results show that the `HYB(HN)` strategy produces the best overall performance with a small number of insertion/deletion operations. For example, a variant of `notepad` can be inexpensively generated with $x_d = 0.2$ and $x_i = 0.5$. Also, even simple random strategies (e.g. adding bogus symbols or deleting existing symbols randomly) can be effective enough if we can combine a random strategy `INS(R)` (or `DEL(R)`) with a sophisticated attack `DEL(H)` (or `INS(N)`). In Section IV-B1, we already observed that `INS(R)` is not sufficiently effective against the sequence alignment algorithm in all four programs; the detection rates remained unchanged and the average sequence alignment score is still greater than 0 even with a large insertion ratio $x_i = 2.0$. However, when we first use the `DEL(H)` strategy, the random insertion of bogus symbols can also be an effective strategy. For example, for `notepad`, an attacker can generate a variant with $x_d = 0.2$ and $x_i = 1.0$ to evade detection. Although `HYB(RR)` completely confused the sequence alignment algorithm (i.e. achieving a zero detection rate) with $x_i = 2.0$ and $x_d = 0.8$, when we consider how expensive this attack is, we would not recommend using `HYB(RR)`.

V. Discussion

A. Cost of obfuscation

The cost of obfuscation often refer to code, data and cycle bloat [5]. However, in the context of code insertions and deletions, we have to also consider the cost associated with the actual insertion or deletion points as well as the particular instruction or API call being inserted or deleted.

Different insertion or deletion points incur different costs. For example, an instruction that is inserted after a `malloc` has to ensure that the buffer pointer is still accessible after it returns, or that a new pointer is created. There are, however,



(a) `triangle` (b) `notepad` (c) `fak-do` (d) `skyhoo`

Figure 3. The detection rates (R.) of sequence alignment algorithm against `HYB(RR)` and `HYB(HN)` attacks, respectively, by varying the insertion ratio x_i from 0.0 to 2.0 and the deletion ratio x_d from 0.0 to 0.8.

insertion points that are free, meaning they do not incur a cost, for example at the beginning of the program since presumably no data has been initialised. In fact, packers can be seen as a special case of a polymorphic obfuscation scheme that only inserts code at position 0. As observed in the evaluation this sort of attack is only effective against sequence alignment techniques if the inserted segment is above 200%, and can be effective for small payloads.

In general, when inserting an symbol between statements i and $i+1$ one has to consider control-flow and data-flow dependencies between i, $i+1$ and other statements. Ignoring such dependencies will result in changes to the semantics of the program. As an example of a control-flow dependency, if i occurs in the n^{th} iteration of a loop and j occurs in the $(n+1)^{th}$ iteration, then any changes made to i should also be reflected in changes made to j. One method to deal with this issue is to use dynamic dependence profiling [14] so as to attach a cost to individual statements for making polymorphic transformations. Mak et al. [14] defined the cost of a statement v, $cost(v)$, is the minimum time required to execute v. The cost increases if v lies on the *critical path* of the program, since one or more tasks have to be completed prior to the execution of v. Similarly, for obfuscation the cost of modifying a statement v is higher if v lies on the critical path of the program. Conversely, statements not on the critical path are cheaper to obfuscate.

The second factor to consider is the actual instruction or API call being inserted or deleted as resilience, stealth and semantic-preservation have to be taken into consideration. Resilience refers to the possibility of the obfuscation scheme being broken by an automatic deobfuscator, and stealth refers to how well the obfuscated code blends in with the rest of the program [5]. Semantic-preservation refers to the fact the obfuscated program exhibits the same functionality as the original one. The insertion of an API call has to preserve semantics and be resilient and stealthy in order to be effective. This is harder to do for some API calls than for others. For instance, the `IsDebuggerPresent` API call queries the process environment block and writes it to register `eax`. One method to insert this API is to do the following.

```
push eax;
call IsDebuggerPresent;
pop eax;
```

This is semantic-preserving as the `push` and `pop` make the three instructions equivalent to a `nop` and is cheap to implement. However, `IsDebuggerPresent` is not a commonly used API in programs and may be easily flagged and removed by an automatic deobfuscator. Moreover, the fact that `eax` is written to immediately after the call seems suspiciously like a `nop` but this can be further obfuscated using techniques such as data obfuscation [4] to obscure this fact.

On the other hand, the `RegSetKeyValue` and `ExitProcess` API calls are more difficult, and thus more costly, to insert or delete. Modifying either of these API calls or their arguments may result in an error in the program and may involve implementing exception handling or additional complexity.

B. Exploiting concurrency

One underlying assumption of sequence-based software birthmarks is that the order of instructions is deterministic. However, for multi-threaded programs this may not be the case as instructions that are independent may be randomly reordered when tasks are run on more than one thread. We can simulate this random reordering using our framework to model a multi-threaded program with two, three and four threads to see how this affects the sequence alignment. We define an n-threaded program with 100% parallelism as having n interleaving threads or sequences of equal length, and a program with 0% parallelism as having only one thread. For a 3- and 4-threaded program, we enforced the first thread to be the longest, and the second to last threads to be sequences of equal length. We assumed that at each time-slice every running thread had an equal probability of executing. The sequences were intialised with symbols that were randomly chosen out of a possible set of 20 each time. The results of the simulation is given in Figure 4. The non-determinism exhibited by multi-threaded programs is able to defeat sequence alignment at about 48%, 33% and 23% parallelism for 2-, 3- and 4-threaded programs respectively.

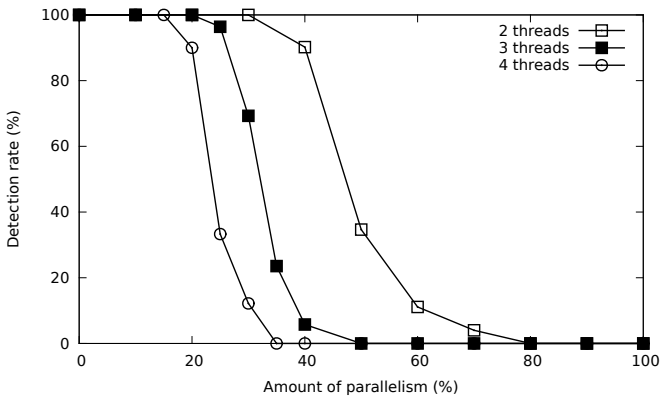


Figure 4. Detection rates for different number of threads, different levels of parallelism

One issue is that in order to maintain semantic equivalence when adding threads one needs to use locks for synchronisation. However, dynamic dependency profiling can be used to identify independent tasks and thus increase parallelism but do not require locks until after these tasks are completed. Moreover, a study has shown that automatic source code-level parallelism of between 2 to 16000 cores was achievable on the Cilk benchmark suite [14]. Together this shows that multi-threaded programming can be effective against a sequence alignment classifier. While converting a sequential program into a parallel one may be non-trivial, a cheaper more viable option is to introduce "dummy" threads to the existing ones. However, the use of dummy threads has to factor in the cost of insertion and its resiliency.

VI. Related work

The concept of "birthmark" for software was first coined by Derrick Grover [7]. Tamada et al. developed a birthmarking scheme [20] as a means to detect theft of java programs, and used method call sequences as one of four different birthmarks. Software birthmarks were formally defined by Myles and Collberg [15] who also introduced dynamic birthmarks. Software birthmarks differ from *software watermarks* in that no additional compile-time effort is needed to generate a birthmark, and a birthmark can only be used to show the similarity between two pieces of software, whereas software watermarks can prove authorship. Of the birthmarks proposed since then, sequence-based birthmarks have been most popular ([16], [17], [22], [18], [9]) and multiple sequence alignment is a new and promising birthmark classifier ([17], [9]). Wang et al. [22] suggested injection and reordering of system calls as possible attacks against birthmark detectors but did not do any evaluation. As far as we know, there has not been any evaluation of the effectiveness of an active attack strategy designed to defeat sequence-based software birthmarks.

Intrusion detection based on API calls have been intensively studied, including mimicry attacks on such systems ([12]). A good summary of the general principles and methods can be found written by Forrest et al. [6]. Mimicry attacks are different from our proposed attack in that for a program P and transformed program P' the objective of a mimicry attack is to find $mark(P')$ that is closest to $mark(Q)$ where Q is in a different detection class. In contrast, the our goal is to transform a program P to P' such that $mark(P')$ is furthest away from $mark(P)$.

API call monitoring for malware detection and behaviour-based profiling, has been studied extensively (for example [1]). Christodorescu et al. [3] proposed extracting malware specifications that are directed behaviour graphs based on system calls. A similar approach was adopted by Kolbitsch et al. [11] where the directed graph was augmented with information about data flow dependencies between system call parameters.

VII. Conclusion

Many researchers [20], [17], [22], [18], [9] have recently proposed software birthmarking schemes based on sequence alignment algorithms. This is because the sequence alignment problem is already a well-known and well-studied problem in bioinformatics; there are a number of algorithms and techniques which can be used to measure similarity between sequences. However, we have shown that intentional sequence modification is different from accidental biological mutations and we have found effective strategies to defeat sequence alignment using real programs and malware.

We found that the intuitive strategies of adding noise by inserting dummy code or replacing existing codes in a random manner were not as effective. Instead we found that

a more effective attack strategy was to consider the locations of the insertion points in the birthmark and the frequency of API calls or instructions.

We also discussed the implementation issues of such attacks and the possibility of non-determinism. Inserting or deleting code is a non-trivial task and can depend highly on the context—the location of the modification, taking into consideration task-level dependencies, and the semantics of the code in question as some are harder to manipulate than others. An alternative is to exploit the use of non-determinism: if an attacker uses a certain amount of multi-threaded programming, that may be sufficient to evade a sequence alignment classifier.

In analysing the performance of attacks, we directly modified software birthmarks rather than the actual programs. In future, we plan to develop obfuscation tools to automatically perform insertions and deletions so as to make such attacks more practical. It would also be interesting to evaluate the effectiveness of actual concurrent programs on a larger scale so as to further extend this attack strategy.

ACKNOWLEDGEMENTS

The authors thank Ross Anderson for bringing to our attention early work in software birthmarking and Christian Collberg for highlighting the issue of thread synchronisation. We also thank Markus Kuhn, Joseph Bonneau, Jonathan Anderson, Robert Watson, Sören Preibusch, Daniel Thomas and the anonymous reviewers for their feedback and for pointing us to other related work.

REFERENCES

[1] U. Bayer, P. Milani, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Proceedings 16th Annual Network and Distributed System Security Symposium (NDSS 2009)*, 2009.

[2] M. Brudno, S. Malde, A. Poliakov, C. B. Do, O. Couronne, I. Dubchak, and S. Batzoglou. Glocal alignment: finding rearrangements during alignment. *Bioinformatics*, 19(suppl 1):i54–i62, 2003.

[3] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In I. Crnkovic and A. Bertolino, editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 5–14, 2007.

[4] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, University of Auckland, July 1997.

[5] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 184–196, New York, NY, USA, 1998. ACM.

[6] S. Forrest, S. Hofmeyr, and A. Somayaji. The evolution of system-call monitoring. In *Proc. of the 2008 Annual Computer Security Applications Conference (ACSAC '08)*, pages 418–430, 2008.

[7] D. Grover. Program identification. *The protection of computer software: its technology and applications, The British Computer Society monographs in informatics*, 2, 1992.

[8] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3), 1998.

[9] W. M. Khoo and P. Liò. Unity in diversity: Phylogenetic-inspired techniques for reverse engineering and detection of malware families. In *Proceedings of the First SysSec Workshop SysSec 2011*. SysSec, 2011.

[10] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer. Behavior-based spyware detection. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, Berkeley, CA, USA, 2006. USENIX Association.

[11] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. yong Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th USENIX Security Symposium*, pages 351–366. USENIX Association, 2009.

[12] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *14th Annual Usenix Security Symposium*, 2006.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices - Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 40:190–200, June 2005.

[14] J. Mak, K.-F. Faxén, S. Janson, and A. Mycroft. Estimating and exploiting potential parallelism by source-level dependence profiling. In *Euro-Par (1)*, pages 26–37, 2010.

[15] G. Myles and C. Collberg. Detecting software theft via whole program path birthmarks. In K. Zhang and Y. Zheng, editors, *Information Security*, volume 3225 of *Lecture Notes in Computer Science*, pages 404–415. Springer Berlin / Heidelberg, 2004.

[16] G. Myles and C. Collberg. K-gram based software birthmarks. In *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, pages 314–318, New York, NY, USA, 2005. ACM.

[17] H. Park, S. Choi, H.-I. Lim, and T. Han. Detecting Java Theft Based on Static API Trace Birthmark. In *Proceedings of the 3rd International Workshop on Security: Advances in Information and Computer Security*, IWSEC '08, pages 121–135, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] H. Park, H. Lim, S. Choi, and T. Han. Detecting Common Modules in Java Packages Based on Static Object Trace Birthmark. *The Computer Journal*, 54(1):108–124, 2011.

[19] D. Schuler, V. Dallmeier, and C. Lindig. A Dynamic Birthmark for Java. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 274–283, New York, NY, USA, 2007. ACM.

[20] H. Tamada, M. Nakamura, and A. Monden. Design and evaluation of birthmarks for detecting theft of Java programs. In *Proceedings of IASTED International Conference on Software Engineering*, pages 569–575, 2004.

[21] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Behavior based software theft detection. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 280–290, New York, NY, USA, 2009. ACM.

[22] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Detecting Software Theft via System Call Based Birthmarks. In *Computer Security Applications Conference, 2009. ACSAC '09. Annual*, pages 149–158, 2009.