View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Purdue E-Pubs

Purdue University
Purdue e-Pubs

ECE Technical Reports

Electrical and Computer Engineering

1-6-2008

Extracting Source Level Program Similarities from Dynamic Behavior

Zheng Zhang
Purdue University, zhang97@purdue.edu

Madhur Gupta
Purdue University, guptam@purdue.edu

Shuo Yang
Purdue University, ayyang@purdue.edu

Guy Lebanon
Purdue University, lebanon@purdue.edu

Y. Charlie Hu
Purdue University, ychu@purdue.edu

See next page for additional authors

Follow this and additional works at: http://docs.lib.purdue.edu/ecetr

Zhang, Zheng; Gupta, Madhur; Yang, Shuo; Lebanon, Guy; Hu, Y. Charlie; and Midkiff, Samuel, "Extracting Source Level Program Similarities from Dynamic Behavior" (2008). *ECE Technical Reports.* Paper 376.
http://docs.lib.purdue.edu/ecetr/376

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

Authors
Zheng Zhang, Madhur Gupta, Shuo Yang, Guy Lebanon, Y. Charlie Hu, and Samuel Midkiff

This article is available at Purdue e-Pubs: http://docs.lib.purdue.edu/ecetr/376

Extracting Source Level Program Similarities from Dynamic Behavior

Zheng Zhang
Madhur Gupta
Shuo Yang
Guy Lebanon
Y. Charlie Hu
Samuel Midkiff
School of ECE
Purdue University

TR-ECE-08-08
January 1, 2008

School of Electrical and Computer Engineering
1285 Electrical Engineering Building
Purdue University
West Lafayette, IN 47907-1285

Contents

Abstract

The vast majority of work on comparing program similarities to detect software piracy either assumes the availability of the program source code (e.g., Moss) or performs a complicated source program transformation to embed carefully designed signatures, or software watermarks, into the binary code. In this paper, we propose a new approach to detecting program similarities that requires neither the availability of the program source nor complicated compile-time watermarking techniques. Furthermore, in contrast to the alternatives, our framework is resistant to standard attacks such as code obfuscation. Our approach exploits the observation that the sequence of system calls performed by a program execution provides a strong signature of the program semantics or functionality, thereby using the inherent properties of a program to identify it. By statistically analyzing sequences of system calls, the relative similarities and differences of program regions can be automatically determined. We have developed a framework that automatically extracts system call sequences, computes the similarities between two binaries via statistical analysis, and maps dynamically similar regions onto textually similar source files. We present several case studies showing the applicability of our framework in pinpointing pirated segments. Our experimental study also shows that directly comparing the binary files of the programs without considering their dynamic behavior is ineffective, and demonstrates strong consistency between the output of our new framework and that of Moss.

1 Introduction

Large software projects incur development costs of tens to hundreds of millions of dollars. For many independent software vendors, the entire business is often dependent on a single program. Loss of the code for these programs can lead to the loss of competitive advantage, and even bankruptcy, for a company, while employee mobility and limited releases of source code to partners make the theft of software and intellectual property an on-going possibility.

The difficulty facing an organization that suspects that its software has been stolen is that it typically only has access to an executable representation of the application containing the stolen software. This presents formidable challenges to determining if a theft has occurred. While disassembly of the executable is technically possible, doing this is likely a violation of the license of the binary, and an unproven (in a court of law) accusation of theft will almost certainly lead to a lawsuit from the accused party, increasing the risk to the organization whose software was stolen. An equally high practical hurdle is that the binary may be tens or hundreds of megabytes in size, making the chore of manually examining the disassembled code nearly impossible. As we show in Section 5.4 automatically examining or comparing binaries is difficult. This is because compiling code at different optimization levels or with different compilers will lead to different executable code, a problem further compounded by linkers which change the binary in different ways depending on the (possibly non-stolen) files being linked against. Binary obfuscators that alter executable code without changing its semantics can further alter the code.

Software watermarking [12, 27, 6] has been proposed as a solution to these problems. Software watermarking techniques add easily recognized data or code segments to a program, or add extra code whose dynamic runtime actions serve as a signature of the code origin. An inherent property of watermarks, however, is that they are something that is added to the program – they are not intrinsic to the program semantics. Therefore, at least in principle, the watermark can always be discovered and removed, or rendered unrecognizable. This is particularly true if the watermarking technique is known. And, in at least some cases, the watermark can be damaged by standard techniques such as program obfuscators.

In this paper, we propose a solution that does not need access to the source code, and that unlike watermarking does not depend on "cosmetic" alterations of the program unrelated to the core program semantics. Instead, our technique relies on intrinsic semantic properties of the programs. These intrinsic properties are the sequence of system calls invoked by a program for some input data. System calls are used because they capture the interaction of the program with the outside world, that is, they capture the permanent side effects of the program. At the same time, it is much harder to corrupt the program's signature by obfuscating system calls than by obfuscating code. Deleting, adding, or reordering system calls would create unwanted effects, degrading the program itself. Such unwanted effects range from making the program incorrect or unstable to slowing it down considerably. By analyzing system calls statistically, we are able to determine regions of similarity in pairs of system call sequences that correspond to regions of two programs built from the same code base.

The principle of using inherent properties of the semantics of a particular version of a program (represented in this paper by sequences of system calls) to identify the code base that all or part of a program is derived from can be generalized to other types of calls. In particular, sequences of calls to `libc` or other standard libraries can be used to identify a program's code base and to measure its similarity to the code base of other programs.

To summarize the contributions of the paper, it presents

- the idea of using intrinsic properties of a program as its digital watermark or signature;
- similarities between system calls and text sequences leading to the adoption of the statistical analysis of n-grams;
- the technique of using system call stack depths as markers in statistical analysis of n-grams;
- global and local similarity measures between system call sequences leading to robust and effective matching;
- results comparing the similarity metrics given by our technique, and the similarity metric output by standard source file comparison tools such as Moss;
- results showing that directly comparing binary files of the programs without considering their dynamic behavior is ineffective.

The rest of the paper is organized as follows. Section 2 describes the key insight behind our work, followed by an an examination of the statistical framework supporting our work. Section 4 analyzes the robustness of our similarity analysis to intentional corruptions by attackers. Section 5 demonstrates experimental results that justify the approach we have taken, and validate our similarity measures. We conclude with a review of related work in Section 6 followed by a discussion and our conclusions.

2 Runtime System Call Sequence as a Characteristic of Program Behavior

System calls invoked during the execution of a software program are semantically significant points indicating the interaction of a program with its environment. They encode important information about the software's functionality and implementation details. In particular, system call sequence (SCS) information represents the services that an application has requested from the underlying operating system. For these reasons, the sequence of system calls invoked by a program for some input data can be used as intrinsic and identifying properties of the program. The key idea of our technique is to use such intrinsic properties of a program to serve as a watermark or signature of the software program. By performing the proper statistical analysis on the system call sequences, we can determine what regions of the program are similar to the regions in another program, and consequently what regions of the two programs are most likely to have been built from the same code base.

In the following, we describe details about two types of SCS information that we extract from a program execution, namely the system call ID sequence (SIS) and the system call stack depth sequence (SDS).

2.1 System Call Sequence: SIS, SDS and Valleys

We consider two attributes of each invoked system call, namely *call name* and *call-stack depth.* Call name is the name of the system call, *e.g.,* `fork()` and `read()`. We map the name to a unique identifier referred to as system call ID. The sequence of ID attributes of a system call sequence (SCS) is referred to as system call ID

```
main(){           foo(){
   ...               ...
   syscall_a;        syscall_c;
   bar();            bar();
   foo();            ...
   syscall_b;        syscall_d;
}                 }


bar(){            tee(){
   syscall_e;        syscall_g;
   tee();            ...
   syscall_f;     }
   tee();
   ...
}
```

Figure 1: A sample program showing system calls.

sequence (SIS). Call-stack depth is the depth of call-stack at the entrance of the system call, *i.e.,* how many layers of function call invocations are associated with this system call. The sequence of call-stack depth attributes of a system call sequence (SCS) is referred to as system call stack depth sequence (SDS). The system call ID reflects the *operation*, more specifically the type of action, of the system call; the call-stack depth reflects the *context* under which the system call was invoked.

Figure 1 shows a sample program, and Figure 2(a) and Figure 2(b) show the SIS and SDS of the execution of the sample program. For instance, the 4th system call `syscall_f()` and the 9th system call `syscall_f()` both have the same ID f, but are invoked under different context. The former one has a stack depth of 2, and the latter one has a stack depth of 3.

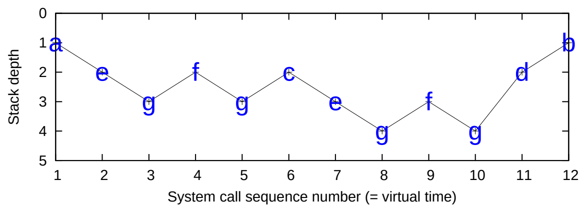
The SDS encodes the hierarchical information among the system call invocations. We further derive valleys from SDS. A *SCS valley* refers to a subsequence of SCS with the first and last system call having a same stack depth d, and stack depths of all intermediate system calls being greater than d. We define the *SCS valley width* of a valley consisting of m system calls as $(m-1)$. SCS valleys are assumed to approximately mark the boundaries of a subroutine invocation and therefore carry significant information. Thus the SCS valley widths are a good indication of subroutines lengths in terms of the number of system calls invoked by them and by their descendants.

Note that a SCS valley does not always exactly map to the life span of a subroutine, nor vice versa. Reverse engineering the subroutine call graph and live spans of subroutines, given only the SCS, is generally a difficult problem. However, since code piracy is more likely to happen at the granularity of subroutines than of individual lines of code, the SCS valleys are more likely to represent pirated code, if piracy happens, compared to other arbitrary subsequences that are not correlated with subroutine life span.

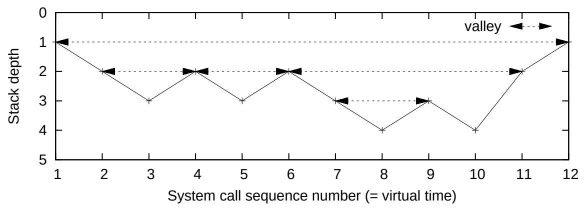
Figure 2(c) shows the SCS of the execution the sample program marked with SCS valleys. Figure 2(d) shows the subroutine call graph of the execution of the sample program. The zig-zag shape of the SCS resembles the

SIS	a	e	g	f	g	c	e	g	f	g	d	b
SDS	1	2	3	2	3	2	3	4	3	4	2	1

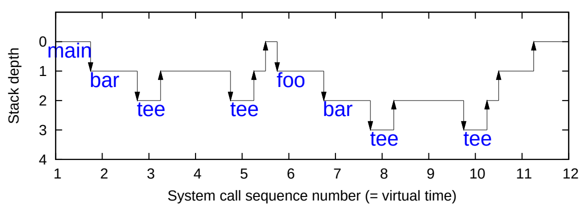
(a) SCS



(b) Visualization of SCS



(c) SCS marked with valleys



(d) Subroutine call graph

Figure 2: The SIS, SDS and valley widths of the sample program of Figure 1.

subroutine call graph. 5 SCS valleys are identified, namely (2,4), (4,6), (7,9), (6,11) and (1,12), having widths 2, 2, 2, 5 and 11 respectively. Trivial valleys such as (3,3) are not considered as valleys.

2.2 Retrieving System Call Sequence

Now we describe our tool for retrieving system call sequence, in particular the SIS and SDS, of a program's execution. Our tool requires neither access to the program's source code nor modification to the operating system kernel. The tool works by trapping the program's system calls through the `ptrace` debugging interface. When a system call is trapped, our tool obtains the current system call name directly from the `ptrace` interface. To obtain the stack depth of the current system call, the tool first obtains the corresponding frame pointer, and then traverses the call-stack chain until reaching the bottom of the program's call-stack. The number of links in the chain is the stack depth. Note that during the traversal, the program's execution is paused by `ptrace`.

We implement our tool on Unix by modifying the `strace` utility. This utility records the system call invoked in a program's execution through the `ptrace` interface. The recorded information associated with each system call originally includes its name, its arguments and its return value. The name is of particular relevance to us. Our modified `strace` records the call stack depth as well, by traversing call stack chain as described above. Since `Ptrace` and `strace` are also available on Microsoft Windows, we expect our methodology of retrieving system call sequences to be extendible to Windows.

3 Similarity Analysis

This section presents the statistical framework used by our method to automatically determine the similarity of program regions. Our discussion in Sections 3.1-3.3 is based only on SIS information. The SDS information is used in the estimation of the prior distribution over n-grams, as described in Section 3.4.

3.1 1-Gram: Histogram of System Calls

Let V be the set of possible system calls. Then with no loss of generality, we identify system calls with their integer IDs, *i.e.,* $V = \{1, \ldots, k\}$. We denote a system call sequence y of length m as $\langle y_1, \ldots, y_m \rangle$, $y_i \in V$.

It is not immediately clear how to relate two SISs x and y to one another. For example, standard norm-based distances are designed for numeric vectors and do not apply for SIS. Two SISs may be of different lengths, and even if they were of the same length, they cannot be related by standard distances such as $\|x - y\| = (\sum |x_i - y_i|^2)^{1/2}$. Another complication arises from the fact that x_i immediately precedes x_{i+1} and should not be considered as an orthogonal dimension.

A reasonable way to summarize a SIS $x = \langle x_1, \ldots, x_m \rangle$ in a way that facilitates distance measurement is to consider the corresponding system call histogram

$$h_1(x) = \frac{1}{m} \left(\sum_{j=1}^{m} \delta_{x_j,1}\,,\ \ldots\ ,\ \sum_{j=1}^{m} \delta_{x_j,k} \right) \in \mathbb{R}^k \tag{1}$$

where $\delta_{k,l} = 1$ if $k = l$, and 0 otherwise. $h_1(x)$ is a vector of length $|V| = k$ that represents the relative frequency of different system calls throughout the SIS. Note that due to the $1/m$ factor in (1), the histogram is normalized and constitutes a probability distribution over system calls. The normalization abstracts away the SIS length and allows direct comparison of varying length SISs by representing them as numeric vectors of a fixed dimensionality k.

The underlying assumption behind the representation (1) is that the sequence $x = \langle x_1, \ldots, x_m \rangle$ was generated by a multinomial distribution [9]

$$p_\theta(x) = \theta_1^{m[h_1(x)]_1} \theta_2^{m[h_1(x)]_2} \cdots \theta_k^{m[h_1(x)]_k}$$

($m[h_1(x)]_i$ is the number of occurrences of system call i in x) governed by an unknown parameter vector $\theta \in \mathbb{R}^k$ subject to $\theta_i \geq 0$ for all i, and $\sum \theta_i = 1$. The multinomial distribution generates sequences $x_1, \ldots, x_m$ where each element $x_j \in V$ is an independent draw from a multivariate Bernoulli distribution assigning result i with probability θ_i. Under the multinomial assumption, we may discard the SIS x, and keep only the summarized information contained in the parameter θ corresponding to the multinomial that generated it. However, the parameter θ corresponding to x is unknown and unobserved, so we resort to representing x by an estimator $\hat{\theta}(x)$ of θ. Indeed, the SIS histogram coincides with the maximum likelihood estimator (MLE) $\hat{\theta}_{\text{mle}}(x) = h_1(x)$. This motivates using $h_1(x)$ as a compact and efficient representation of x, since the MLE enjoys nice statistical properties. In particular, it is well known that $h_1(x)$ is an unbiased, consistent, and efficient estimator of the unknown parameter θ [9]. These properties and others single out $h_1(x)$ as the estimator of choice for θ within classical statistics.

3.2 n-Gram: Incorporating System Call Sequence Information

The main disadvantage of the above representation and of the multinomial distribution assumption is that the assumption ignores position information and system call ordering, and assumes that x_i are independently identically distributed (i.i.d.). For example, the h_1 representations of the two permuted versions of x are the same

$$h_1(\langle 1, 2, 3, 1\rangle) = h_1(\langle 3, 2, 1, 1\rangle) = \left(\frac{1}{2}, \frac{1}{4}, \frac{1}{4}\right).$$

Thus, measuring key frequency information $h_1(x)$ ignores potentially important patterns of consecutive system calls. To address this weakness, we introduce a generalized version of SIS histograms which incorporate measurements of consecutive patterns in a robust manner. To measure interactions of i consecutive system calls, we use the normalized histogram of length-i subsequences in x, denoted by $h_i(x)$. Conforming with standard terminology, we refer to $h_n(x)$ as the n-gram representation of x. For example, the $\langle 3, 1, 3\rangle$-entry of the 3-gram representation of a SIS x is

$$[h_3(x)]_{\langle 3,1,3\rangle} = \frac{1}{m-3+1} \sum_{i=1}^{m-3+1} \delta_{x_i,3}\delta_{x_{i+1},1}\delta_{x_{i+2},3}.$$

The statistical assumption behind the h_n representation is that length n consecutive subsequences in the SIS are generated by a multinomial distribution. As before, the multinomial parameter associated with x is unknown, and we have to estimate it by using the MLE estimator $h_n(x)$.

The order n multinomial assumption and the use of the h_n representation have both advantages and disadvantages associated with the order n being used. Lower order representations do not faithfully capture long range sequential information. On the other hand, higher order representations are extremely high dimensional with the dimensionality growing exponentially. As a result, estimators $h_n(x)$ for large n, despite having nice theoretical properties, may be extremely noisy and very inaccurate. This trade-off between rich representation for large n and low estimation error for small n is commonly called in statistics the bias-variance trade-off. In the limit that corresponds to infinitely long system call sequences $m \to \infty$, even high dimensional estimators would converge to the true values, and we would benefit most from using h_n for large n. However, for finite sequence lengths m, it is likely that the n that optimizes the above trade-off lies somewhere in between the two extremes of $n = 1$ and $n = m$.

Finding an optimal n in terms of the bias-variance trade-off would lead to good estimation accuracy. However, using a single h_n representation is suboptimal since it forces us to represent the data at a specific resolution or level of details. Instead, we use a Bayesian viewpoint which lets us aggregate information across multiple h_n representations of different orders n. This approach has both theoretical and practical benefits over choosing a single h_n. The problem that we turn to next is how to aggregate the h_n representations across different orders.

3.3 Aggregating Multiple n-Grams

In the Bayesian viewpoint, alternative n-order multinomial models are held as correct with certain probabilities $p(n)$, called *prior probabilities* or a *prior distribution*. The Bayesian viewpoint aggregates information across different orders by using basic probability laws and in most cases results in a linear combination with coefficients corresponding to the prior probabilities.

We consider the following cosine similarity measure $\text{sim}(h_n(x), h_n(y))$ between the n-order histogram representations of the SCSs x and y

$$\text{sim}(u, v) = \cos(\sphericalangle(u, v)) = \frac{\sum u_i v_i}{\sqrt{\sum u_i^2}\sqrt{\sum v_i^2}} \in [0, 1]. \tag{2}$$

Above, u_i corresponds to $[h_n(x)]_{\langle a_1,\ldots,a_n\rangle}$ and the summation covers all n-tuples $\langle a_1, \ldots, a_n\rangle \in \{1, \ldots, k\}^n$. Under the Bayesian assumption of randomness in the order n, such a measure becomes a random variable rather than a scalar which is conveniently summarized by its expectation

$$\begin{aligned}\text{sim}(x, y) &= E_{p(n)}\{\text{sim}(h_n(x), h_n(y))\} \\ &= \sum_n p(n)\,\text{sim}(h_n(x), h_n(y)).\end{aligned} \tag{3}$$

Note that since $\text{sim}(h_n(x), h_n(y))$ is in the range $[0, 1]$, then so is $\text{sim}(x, y)$ with 1 corresponding to $x = y$ and 0 corresponding to orthogonal n-gram histograms for all orders (assuming that $p(n) > 0$). This fact, together with the computational efficiency of the cosine-angle similarity has made this similarity one of the most popular similarity measures for text analysis and information retrieval [25, 7]. Therefore, we use this cosine similarity in our similarity analysis.

The prior distribution may be obtained from data or may be obtained by consulting a domain expert. We address this issue further in the next subsection and then consider the computational efficiency of computing the expected similarity measure.

3.4 n-Gram Prior Estimation

Balancing the trade-off between capturing high order interactions and achieving high estimation accuracy, the prior distribution $p(n)$ may be defined in several ways. The most common way of obtaining a prior distribution is by eliciting it from a domain expert or using the prior belief of the person conducting the analysis. Such prior selection is guaranteed to be optimal with respect to the subjective prior belief, but is not guaranteed to yield good results objectively. In this paper, we explore a data driven method of estimating $p(n)$ which removes much of the arbitrariness of a personal belief prior.

In obtaining a good prior distribution $p(n)$, we need to consider the proportion or probability of n-order interactions for different n. A key observation that we made in Section 2 is that the valley-widths are a good indication of subroutine lengths as they reflect the number of system calls invoked by the subroutines or by their descendants. Furthermore, since code piracy is more likely to happen at the granularity of subroutines than of individual lines of code, an n-length valley of the SCS which approximately corresponds to a subroutine in the source code that induces n (consecutive) system calls is likely to exhibit n or lower order interactions in the similarity analysis. Therefore, the proportion of time programs spend in valleys of width n should dictate $p(n)$. To explore this observation, we analyze the frequencies of valleys at different widths and use such a valley-width histogram of the SCS as an estimation of $p(n)$.

However, the valley-width histogram may be noisy and may not reflect accurately the true distribution of subroutine lengths in terms of system calls. A parametric or non-parametric fit to $p(n)$ based on that histogram would smooth out the discontinuities that naturally appear in the data-based histogram. In our case, we used parametric estimation to obtain an estimate for $p_\theta(n)$ based on the valley width histogram. Specifically, we considered several families such as Gaussian, exponential and gamma, and used the maximum likelihood estimator $\hat{\theta}$ in our prior estimate $p_{\hat{\theta}}(n)$. In our experiments, the obtained distribution did not vary much among the three parametric families mentioned above. However, for all three families, the resulting distribution was a rapidly decreasing function with substantially higher values for low n than for high n.

3.5 Local Similarity Measures

The expected similarity $\text{sim}(x, y)$ provides a single number describing the overall similarity of the sequences x and y averaged over different n-gram representations. We call this expected similarity *global similarity*, since it reflects the average similarity throughout two sequences. Often, instead of a single global similarity measure, we are interested in a *local similarity* measure $\text{sim}_{s,t}(x, y)$ around x_s and y_t. This measure can be useful for analyzing system call sequences that correspond to programs that are largely unrelated, but contain small portions which are identical or very similar.

A simple way to measure such local similarity is to restrict the system call sequences x and y to windows of size $2r+1$ around system call indices s and t

$$\text{sim}_{s,t}(x,y) = \text{sim}(\langle x_{s-r}, \dots, x_{s+r}\rangle, \langle y_{t-r}, \dots, y_{t+r}\rangle).$$

The obtained measure $\text{sim}_{s,t}(x,y)$, viewed as a function of s,t, is a bivariate function whose peaks correspond to positions in x,y containing similar system call structures.

3.6 Computational Efficiency in n-Gram Analysis

Computing quantities involving n-order histograms may become infeasible, even for moderately sized m,n,k. For example, a naive computation of $\text{sim}(h_n(x), h_n(y))$ for sequences x and y of lengths m_x and m_y would be of complexity $O(m_x + m_y + k^n)$, due to the registration of all n-grams and the summation in (2) over all possible n-tuples. Such complexity becomes prohibitive for all but the smallest n.

For the case of $\text{sim}(h_n(x), h_n(y))$ being cosine similarity as in (2), we describe below an efficient quadratic time computational procedure. We register all occurrences of the n-tuple $\langle x_1, \dots, x_n\rangle$ in $h_n(y)$ by sliding a window of size n over y and counting the number of matches with $\langle x_1, \dots, x_n\rangle$. We repeat this procedure for n-tuples corresponding to a size n sliding window over x such as $\langle x_{i+1}, \dots, x_{i+n}\rangle$, $i = 1, \dots, m_x - n + 1$. Thus, by going over all combinations of the positions of the two sliding windows we can keep track of all common n-grams. The registration of common n-grams in $h_n(x), h_n(y)$ may be done via hashing or some other efficient representation of sparse data. Subsequently, we proceed by computing the summation over the common n-grams. Overall, the complexity of the above method is quadratic $O(km_xm_y)$.

The same method can be applied for computing the denominator of (2) by considering the norms as inner products $\|h_n(x)\|^2 = \langle h_n(x), h_n(x)\rangle$ and $\|h_n(y)\|^2 = \langle h_n(y), h_n(y)\rangle$. This results in a quadratic $O(k(m_x^2 + m_y^2 + m_xm_y))$ procedure for computing $\text{sim}(h_n(x), h_n(y))$. Computing the expected similarity $\text{sim}(x,y)$ would then take a total of $cO((m_x+m_y)^2)$ assuming that $p(n)$ has constant positive support of size c, *e.g.,* $p(r) = 0$ for $r > c$. The last assumption of bounded support for $p(n)$ is a natural one since $p(n)$ tends to be extremely low for large n and zeroing it would not result in a noticeable difference.

4 Robustness of Similarity Analysis

Comparing two identical SIS results in a similarity measure of $1 = E_{p(n)}\langle h_n(x), h_n(x)\rangle / \|h_n(x)\|^2$ indicating a perfect matching of the identical sequences. In Section 5.3, we show experimentally that when the match is not perfect because only part of the original application code was stolen, or the stolen code forms only part of an application, our technique accurately identifies sections of the application built on stolen code that are similar to or different from the original application. In this section, we provide a theoretical analysis of a different problem – how robust is our method in comparing sequences that have been intentionally corrupted. In particular, we theoretically analyze the optimal similarity detection error rate (defined as the probability of making a wrong prediction) and its dependency on the corruption or noise level.

We consider two models for system call corruption: insertion and replacement. Insertion refers to the modification of a SIS by inserting a number of additional system calls into the original SIS. This behavior may be performed by an attacker in attempt to corrupt the original SIS signature. The inserted calls can be dummy system calls, *i.e.,* system calls having no particular role beyond corrupting the SIS histogram representation, and these calls can be inserted at random places. For example, the SIS $\langle 1,3,2,3,4\rangle$ may be corrupted into $\langle 5,1,3,5,2,5,3,4,5\rangle$ by inserting the dummy system call 5 at random positions. This system call 5 could be `write()` to `/dev/null`. Note, however, that such corruption incurs the side effect of slowing down the program by executing additional redundant system calls. Moreover, in the case of simple insertion of system calls having no external effect, it is possible to collect the parameter information of system calls, and preprocess the SIS data before the similarity analysis by removing such redundant system calls, and constructing the histogram representation over the remaining system call types.

The second corruption process that we consider is replacement. In this case, certain system calls in the SIS are selected and replaced by other system calls achieving the same functionality. As an example, a corruption process could be $\langle 1,3,2,3,4\rangle \mapsto \langle 1,2,2,3,4\rangle$, where the second system call 3 (e.g. `read()`) was replaced by the system call 2 (e.g. `pread()`) which is used to perform equivalent function as the former. Note that in the case of replacement, in contrast to insertion, the SIS maintains its original length.

Mathematically, corruption by insertion is harder to model than corruption by replacement and therefore we focus on the theoretical analysis of the latter. For simplicity, we consider below the case of 1-gram histogram representation which corresponds to an independent identically distributed (i.i.d.) model generating the data. Extending the discussion for higher n-grams is somewhat more complicated, but straightforward.

We thus assume that a system call sequence x corresponding to code base i is generated independently by some distribution $p_i(x) = \prod_{j=1}^{m} p_i(x_j)$. The corruption of the SIS by replacement is modeled by a mixture process $p_i^{(\theta)} = \theta p_0 + (1-\theta)p_i$, where p_0 is the probability governing the call type of the replaced system calls. In other words, the corruption model may be described as going over each system call in the SIS and with probability $1-\theta$ leaving it intact and with probability θ replacing it with a new system call drawn from the replacement distribution p_0. The probability of obtaining a SIS x from code base i (drawn from distribution p_i) after undergoing replacement corruption at level θ is

$$p_i^{(\theta)}(x) = \prod_{j=1}^{m}(\theta p_0(x_j) + (1-\theta)p_i(x_j)). \tag{4}$$

In general, the difficulty of determining the original program corresponding to a SIS becomes more difficult as the corruption level θ increases. In the limit of $\theta \to 1$, the original information is lost and there is no hope of success in determining the correct program beyond simple guessing. In the limit of $\theta \to 0$, there is no corruption and distinguishing the origin of a SIS is limited only by the statistical variations of the different distributions p_i corresponding to different programs ($i = 1, 2$). In the case that the distribution p_i represents a deterministic model (i.e. $p_i(z) = 1$ for some z and 0 otherwise), the error rate or probability of making a wrong prediction will be zero. It is interesting to consider the fundamental limit of our ability to correctly determine the correct code base in the intermediate case $0 < \theta < 1$. This limit, obtained through the use of the Chernoff-Stein theorem, is independent of a specific similarity analysis method and depends in a fundamental way on the corruption level θ, and the program and replacement distributions p_1, p_2, p_0.

4.1 Fundamental Limit of Similarity Analysis under Replacement Corruption

The Chernoff-Stein theorem, reproduced below, will allow us to quantify the relationship between the asymptotic error rate of any similarity based method and the corruption level θ. For more details, consult for example [16] or Chapter 11 of [15].

Proposition 1. *Consider the hypothesis test for the distribution of i.i.d. data $\langle x_1,\ldots,x_m\rangle$ between two alternatives $x_i \sim p_1$ and $x_i \sim p_2$ with a finite KL divergence, defined as*

$$D(p_1||p_2) \overset{\text{def}}{=} \sum_z p_1(z)\log\frac{p_1(z)}{p_2(z)} < \infty.$$

Denoting the acceptance region for hypothesis p_1 by A_m and the probabilities of the two types of error by $\alpha_m = 1 - p_1(A_m)$ and $\beta_m = p_2(A_m)$, we have

$$\lim_{m\to\infty}\frac{1}{m}\log\min_{\alpha_m<c, A_m}\beta_m = -D(p_1||p_2).$$

The acceptance region A_m represents the set of data sequences $\langle x_1, \dots, x_m\rangle$ for which we decide the null hypothesis, i.e., that the sequence originated from p_1. Its complement A_m^c represents the set of sequences for which we decide on the alternative, i.e., that the sequence originated from p_2. The quantities α_m, β_m represent the probabilities of making type I and type II errors (determining p_1 if the correct source is p_2 and determining p_2 if the correct source is p_1). The Chernoff-Stein theorem states that fixing the probability of type I error α_m below some acceptable threshold c, the best possible probability of making a type II error is approximately

$$\beta_m \approx \exp(-mD(p_1||p_2)). \tag{5}$$

Thus, the KL-divergence between the two models $D(p_1||p_2)$ represents the fundamental limit on the exponent (coefficient $a = D(p_1||p_2)$ of the exponential decay function $\exp(-am)$ in Equation (5)) of the best possible type II error rate and thus it represents an inherent measure of difficulty in the task of distinguishing p_1 from p_2.

We apply the Chernoff-Stein Theorem to the problem at hand of similarity analysis under the replacement model by considering the fundamental error exponent, or KL-divergence, between two sources $p_1^{(\theta)}, p_2^{(\theta)}$ representing replacement corruption at level θ. Comparing $D(p_1^{(\theta)}||p_2^{(\theta)})$ to the KL divergence between the two uncorrupted models $D(p_1||p_2)$ gives us a notion of the added difficulty associated with the corruption process (4).

Lemma 1.

$$D(p_1^{(\theta)}||p_2^{(\theta)}) \leq (1-\theta)D(p_1||p_2) \tag{6}$$

Proof. Apply the log-sum inequality for non-negative numbers a_i, b_i, $\sum a_i \log a_i/b_i \geq (\sum a_i) \log \frac{\sum a_i}{\sum b_i}$ (Theorem 2.7.1 in [15]) to each term in the KL divergence to obtain

$$\begin{aligned}&(\theta p_0(z) + (1-\theta)p_1(z)) \log \frac{\theta p_0(z) + (1-\theta)p_1(z)}{\theta p_0(z) + (1-\theta)p_2(z)} \\ &\leq \theta p_0(z) \log \frac{p_0(z)}{p_0(z)} + (1-\theta)p_1(z) \log \frac{p_1(z)}{p_2(z)}.\end{aligned}$$

Summing over z we obtain the required result. □

As a result of Proposition 1 and Lemma 1, we have an upper bound on the exponential rate of decay of the error rate (5) for corrupted data in terms of the corresponding exponent for uncorrupted data and the corruption level θ. Note that the right hand side of Equation (6) does not depend on the replacement distribution p_0 and thus can be applied in a general way without the knowledge of the specific corruption distribution.

Lemma 1 leads to useful observations concerning the amount of corruption needed to substantially reduce the error rate. For example, a corruption level of $\theta = 0.3$ or greater leads to a reduction of the error exponent by no more than 0.7. Such fundamental statements are true for any similarity analysis method and any replacement model p_0 and thus are broadly applicable.

5 Experimental Results

In this section, we present experiments concerning the run-time similarity analysis of different versions of two widely used software systems, namely GCC/ICC compilers and Java virtual machines. Firstly, we show that our global similarities on these softwares match with our expectation that is based on our knowledge of the software version numbers and vendors. We also show the global similarities are consistent with those reported by Moss. Therefore, our proposed similarity measure is suitable for characterizing source level similarity. Secondly, we demonstrate that using the local similarity, our technique is able to locate subroutines that are likely to be based on the same code base. Finally, we consider textual similarity between binaries, a seemly alternative to our proposed behavioral similarity, and demonstrate its ineffectiveness.

Table 1: Benchmarks used in this paper.

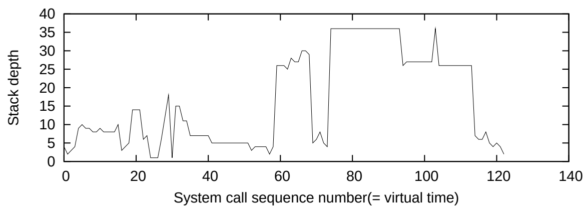
Benchmark Suite	Software
Compilers	(a) GCC-3.2.3 (b) GCC-3.3.0, GCC-3.3.2, GCC-3.3.3, GCC-3.3.5 (c) GCC-3.4.2, GCC-3.4.6 (d) GCC-4.0.3, GCC-4.1.1 (e) ICC-9.0
Java Virtual Machines	(a) IBM JVM-1.3.1, IBM JVM-1.4.2 (b) Sun JVM-1.3.1, Sun JVM-1.4.1, Sun JVM-1.4.2

5.1 Benchmarks

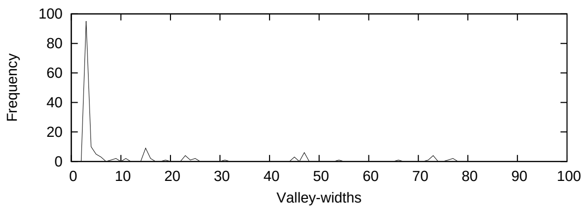
Our benchmarks include GCC/ICC compilers and Java virtual machines (JVM). They are large complex software packages with over a million lines of code, and are the result of an extensive programming effort. Both GCC/ICC and JVM have different versions of implementation provided by different vendors, but offer essentially the same functionality. Therefore, these implementations are good examples for similarity analysis. Both softwares basically follow the common software version scheme in naming the versions. Each version is named using several digits as `major.minor[.revision[.build]]` or `major.minor[.maintenance[.build]]`, where the fields in brackets [ ] are optional. Thus, the numerical gap between two versions is a good indication of the source level similarity. In the following, we describe some relevant facts concerning the benchmarks and the input fed into the benchmarks for similarity analysis.

GCC/ICC Compilers The GNU Compiler Collection (GCC) [1] is a open source project by the GNU Project, producing a set of compilers for various programming languages. Different versions of GCC compilers contain different optimization passes, but their functionality remains essentially the same. The Intel C/C++ compiler (ICC) [3] is Intel's proprietary C/C++ compiler implementation, which achieves the same overall functionality in handling C/C++. The GCC and ICC compilers used in our experiments are listed in Table 1. GCC 3.x.x's evolve from the same code base, but they incrementally update the internal implementations with few, if any, changes to the system call behavior. GCC 4.0.x's and later versions contain the tree SSA intermediate representation infrastructure which GCC 3.x.x's do not contain. Thus we expect that similarities between GCC implementations obey the order of numerical distance between version numbers. In our similarity analysis, the input to the GCC/ICC compilers is a simple hello-world C program to be compiled and linked.

Java Virtual Machines A Java virtual machine (JVM) is a software system implementing a virtual machine that interprets Java byte-code. Different vendors such as Sun [4] and IBM [2] independently implement their JVMs adhering to the common JVM specification. Both specification and implementation version numbers continue to evolve. Two JVMs with the same version numbers by different vendors, for example IBM's JVM-1.3.1 and Sun's JVM-1.3.1, conform to the exact same specification, and therefore achieve the same functionality, but may be quite different in implementation since they were developed by different vendors whose softwares are protected by intellectual property law. The JVMs used in our experiments are listed in Table 1. In our similarity analysis, the input to the JVMs is a simple hello-world Java program already compiled into byte-code.



(a) Stack depth trace of GCC-4.1.1



(b) Frequency of valley-widths (aggregated over all GCCs)

Figure 3: Stack-depth sequence and valley-width histogram for a typical GCC implementation.

5.2 Evaluation of Similarity Measure

We perform similarity analysis on the benchmark programs. As described in Section 2 and Section 3, we use our modified `strace` tool to obtain the SIS and SDS of a sample execution of each program, and compute the similarity between two programs by aggregating multiple n-gram similarities of their SISs using the weight functions $p(n)$ that are obtained by fitting parametric distributions to the valley-width histograms (as described in Section 3.4). Each valley-width histogram is computed by aggregating all valley-widths of various implementations of the same software (specification), and the histogram is shared by all similarity analysis between any two implementations of the software. The methods of parametric fitting of distributions to the valley-width histogram include the maximum likelihood estimator (MLE) for the Gaussian, exponential and uniform distributions. In addition, we also consider the following singular distributions $p(n) = \delta_{n,1}$, $p(n) = \delta_{n,10}$, $p(n) = \delta_{n,30}$ ($\delta_{k,l} = 1$ for $k = l$, and 0 otherwise) which correspond to single n-gram models without any meaningful Bayesian aggregation.

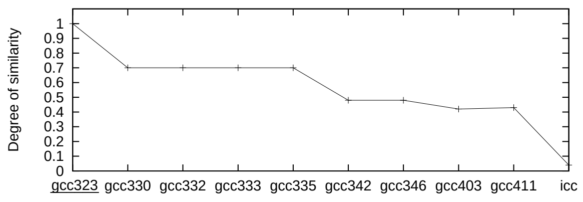
Next, we compare the similarities produced by our analysis with two external sources – our expectation based on version numbers, and the similarities reported by Moss. We focus on global similarity rather than local similarity, because the corresponding global similarities from those two external sources are easier to obtain. Nevertheless, local similarity is the application of our similarity measure on a sliding window, and hence the effectiveness of global similarity implies the effectiveness of local similarity.

Figure 3 shows some intermediate results in our similarity analysis. Figure 3(a) depicts the SDS of GCC-4.1.1, and Figure 3(b) depicts the valley-width histogram derived from various versions of GCC/ICC. The histogram illustrates the expected behavior that lower order n-grams have higher frequency.

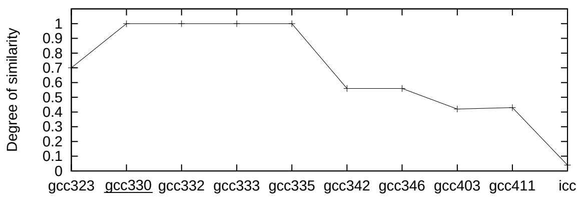
5.2.1 Consistency with Version Number Expectation

In the following, we show that the global similarities conform to the following empirical rule, which we call version number expectation: 1) the similarity between two independent implementations by different vendors is less than that between two implementations by the same vendor; 2) similarities $\text{sim}(x, y)$ between any two implementations x and y by the same vendor obey the ordering of numerical distances between their version numbers, *i.e.,* $|\text{ver}(x) - \text{ver}(y)|$.

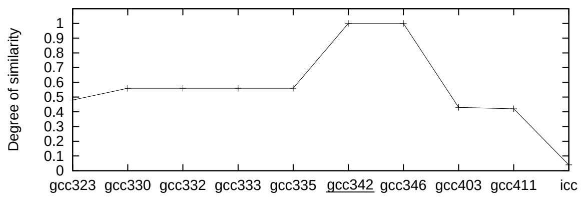
Figure 4 shows the global similarity $\text{sim}(x, b)$ using a Gaussian-fitted prior $p(n)$ for SCSs x and b corresponding to different softwares chosen from the GCC/ICC suite. The baseline b is fixed and x varies in each sub-figure. The software corresponding to x's are shown along the x-axis, and the baseline software corresponding to b is underlined. Along the x-axis, the GCCs are arranged in increasing order of version number, followed by ICC. In each sub-figure, the expected similarity should peak to 1 when x is the baseline b, and monotonously decrease as x goes further from b, both on the left side and on the right side. The similarities in Figure 4 mostly behave as this expectation, with few exceptions in which GCC-4.0.3 is shown slightly less similar to an earlier version than GCC-4.1.1. Also, as expected, the similarity between GCC and ICC is quite low. In fact, it is almost 0, indicating nearly orthogonal n-gram representations of the two vendors. Moreover, we also see that the similarity curves



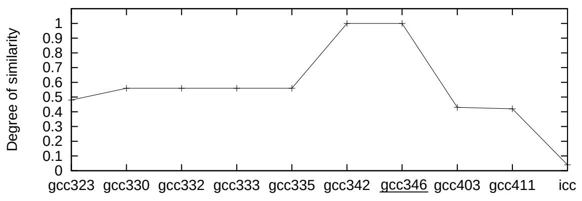
(a) Baseline: GCC 3.2.3



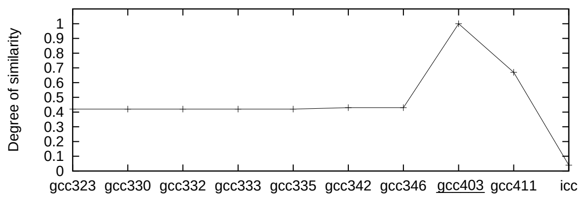
(b) Baseline: GCC 3.3.x



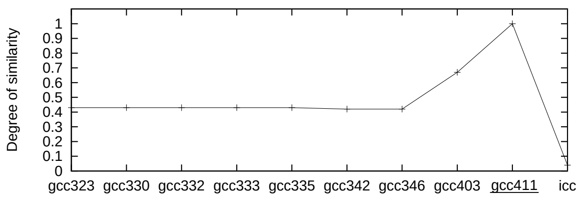
(c) Baseline: GCC 3.4.2



(d) Baseline: GCC 3.4.6



(e) Baseline: GCC 4.0.3



(f) Baseline: GCC 4.1.1

Figure 4: Global similarities of GCC/ICC. Different versions of GCC and ICC-9.0 are compared with a certain baseline software (underlined). The analysis is based on a Gaussian-fitted $p(n)$.

remain flat in the range of GCCs having the same major and minor version number but different revision version number (*e.g.,* 3.3.x or 3.4.x), and jump across different minor versions (*e.g.,*, 3.3.5 and 3.4.2). This trend shows that the modification related to minor version number change is more significant than revision number change. Due to space limitations, the similarities based on other fitting methods are omitted. They exhibit a similar trend as Gaussian.

Figure 5 shows the global similarity $\text{sim}(x, b)$ using a Gaussian-fitted prior $p(n)$ between different JVMs with a baseline Sun JVM-1.3.1. The similarities again behave as expected within the same vendor and across different vendors. Due to space limitations, the similarities using other baselines or based on other fitting methods are omitted. They exhibit a trend similar to that shown in Figure 5.

The above GCC/ICC and the JVM experiments provide partial evidence supporting our claim that our SCS-based similarity measure captures meaningful source-level program similarity. Next, we use Moss to further validate our claim.

5.2.2 Consistency with Moss

Moss [5] is a tool offered as an Internet service that determines the similarity of C, C++, Java, Pascal, Ada, ML, Lisp, or Scheme programs by analyzing their source codes. Moss produces a similarity score between 0 and 1 that corresponds to the percentage of match found between two source codes. Since Moss is a well established source code similarity measure, we compare our SCS-based similarity with the similarity given by Moss. Because Moss requires source code, our comparison study is restricted to the GCCs, whose source code is open.

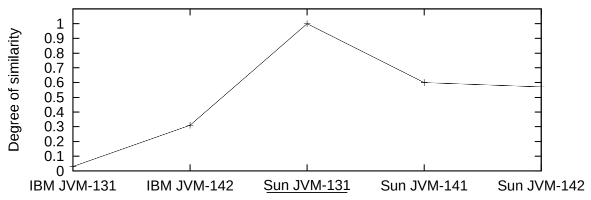


Figure 5: Global similarities of JVMs. Different JVMs are compared with a baseline Sun JVM-1.3.1 (underlined). The analysis is based on a Gaussian-fitted $p(n)$.

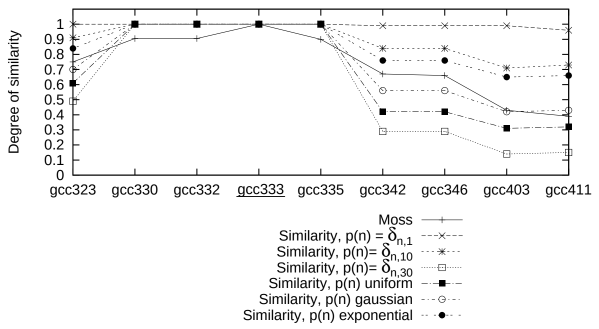


Figure 6: Comparison of Moss and SCS-based similarity measure for different versions of GCC compared with the baseline GCC-3.3.3 (underlined). The SCS analysis uses different priors $p(n)$, as indicated.

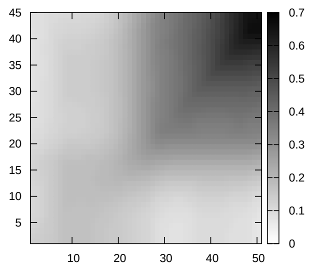
Figure 6 shows the global similarities between various GCC versions and a baseline GCC-3.3.3, as well as the similarities given by Moss. Our analysis uses various fitting methods to derive prior distributions $p(n)$. Due to space limitations, the similarities using other baselines are omitted. They show similar trend as in Figure 6. Overall, the trend given by SCS-based similarity is consistent to the trend provided by Moss. Note the impact of the choice of prior fitting method on the SCS-based similarity curve and on its agreement with the similarity curve output by Moss. Choosing $p(n) = \delta_{n,1}$ results in extremely poor behavior. The Gaussian and exponential fit perform better than the others, and Gaussian performs most closely to Moss. Thus, for the rest of the paper, we use Gaussian as the fitting method.

In summary, our experiments show close agreement between our SCS-based similarity and version number rule, and between the SCS-based similarity measure and Moss. That latter agreement is especially strong when using a prior $p(n)$ fitted by Gaussian mle. As a result, these experiments validate our proposed similarity measure as a meaningful measure for characterizing source-level program similarity.

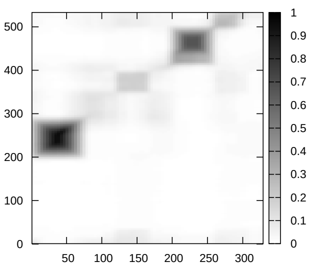
5.3 Source Level Similarity Extraction

Next, we demonstrate the use of local similarity in locating the highly similar and thus possible stolen source code piece. Our benchmarks do not include a case in which one software contains a piece of source code stolen from another software. However, we believe that different versions of a vendor's software contains highly similar source code pieces that remain unchanged across these version. We study this case, as it resembles the real stolen code case, and we show that the local similarity can discover those unchanged code pieces.

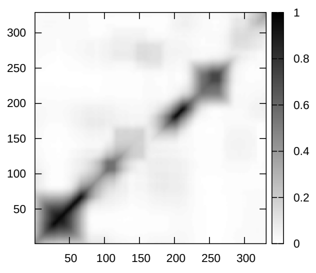
We analyze the local similarity of three pairs of programs, GCC-4.0.3 and GCC-3.4.6, Sun JVM-1.4.1 and Sun JVM-1.3.1, and Sun JVM-1.4.2 and Sun JVM-1.4.1. As discussed in Section 3.5, local similarity between two SCSs x and y is a bivariate function $g(s,t) = \text{sim}_{s,t}(x,y)$. We visualize this bivariate function in Figure 7. The



(a) GCC-4.0.3 vs GCC-3.4.6



(b) Sun JVM-1.4.1 vs Sun JVM-1.3.1



(c) Sun JVM-1.4.2 vs Sun JVM-1.4.1

Figure 7: Local similarity of software pairs. Y-axis corresponds to the SCS index of the first software, and X-axis corresponds to the SCS index of the second software. The prior distribution is Gaussian-fitted.

size of sliding window used in the local similarity analysis is 50 (the length of the SCS was in the range of 100-200). For all three pairs, we see highly similar regions in the contours. For Figure 7(a), the highly similar region is in the right top corner; for Figure 7(b), there are two highly similar regions, one in the right top corner, the other in the left middle; for Figure 7(c), the highly similar regions are along the diagonal. They correspond to the code pieces that are largely unchanged across the two compared version.

We further attempt to map the high similarity regions back to source code. Picking one point inside a high similarity region as a candidate, the task of mapping the high similarity point $(\hat{s}, \hat{t})$ back to the subroutines in the source codes can be automated as follows. We first modify our `strace`-like tool to record the program counter (PC) at which a system call is invoked, in addition to recording the call ID and stack depth. Then we find out the two PCs associated with two system calls $\hat{s}$ and $\hat{t}$ in the two SCSs. Next we re-execute each executable under GDB, and set a break point at the obtained PC. When the execution hits the break point, we obtain the names of the similar subroutine and the source file containing the subroutine from GDB.

As a case study, we perform the above procedure on the pair of GCC-3.4.6 and GCC-4.0.3, since their source codes are available. Figure 7(a) discloses a high similarity region in the top right corner. We choose a point $(\hat{s}, \hat{t})$ where the value of $g(\hat{s}, \hat{t})$ is greater than 0.55. Following the above procedure, we found that the two subroutines `execute()` and `do_spec_1()` in `gcc/gcc.c` in both GCCs correspond to the high similarity region. This observation is confirmed by Moss, which found more than 0.9 similarity between the source codes of these subroutines.

We also choose other points $(\hat{s}, \hat{t})$ of low local similarity $g(\hat{s}, \hat{t})$. Repeating the above procedure, we found subroutines that have the same name but have low similarity. One such subroutine is `gcc_init_libintl()` in `gcc/intl.c`. Moss also found no matches between the source codes of these subroutines, confirming our observation.

In summary, we present an automated procedure for locating code pieces with high similarity and demonstrate usefulness of this procedure by comparing the obtained results with source code similarity measure obtained by Moss.

5.4 Textual Similarity as an Alternative?

The technique based on dynamic behavior presented in this paper uncovers source-level similarities between two programs when only binaries for the programs are available. As discussed in Section 1, the alternative approach of automatically examining the program binaries is challenging in general. Previously, static textual similarity techniques have been shown to be highly effective in discovering similarities files, and variants of these techniques have been applied for the purpose of avoiding storing redundant data [23, 28] and avoiding transferring redundant

Table 2: Textual similarity between different GCC/ICC executables.

Similarity (%)		GCC									ICC
		3.2.3	3.3.0	3.3.2	3.3.3	3.3.5	3.4.2	3.4.6	4.0.3	4.1.1	9.0
GCC	3.2.3	100.0	2.5	2.4	2.4	2.4	0.6	0.6	0.3	0.3	0.0
	3.3.0	2.5	100.0	10.2	9.6	7.3	0.9	0.8	0.4	0.3	0.0
	3.3.2	2.4	10.2	100.0	15.1	10.9	0.9	0.8	0.4	0.3	0.0
	3.3.3	2.4	9.6	15.1	100.0	12.5	0.9	0.9	0.4	0.3	0.0
	3.3.5	2.4	7.3	10.9	12.5	100.0	0.9	0.9	0.4	0.3	0.0
	3.4.2	0.6	0.9	0.9	0.9	0.9	100.0	15.9	1.6	1.1	0.0
	3.4.6	0.6	0.8	0.8	0.9	0.9	15.9	100.0	1.6	1.1	0.0
	4.0.3	0.3	0.4	0.4	0.4	0.4	1.6	1.6	100.0	2.8	0.0
	4.1.1	0.3	0.3	0.3	0.3	0.3	1.1	1.1	2.8	100.0	0.0
ICC	9.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	100.0

data [26]. In this section, we analyze the effectiveness of such techniques when applied to program executables. In the following, we first adapt such techniques to measuring similarities between executables, and then apply them to our benchmark programs.

5.4.1 Textual Similarity between Executables

The basic idea is to treat program executables as regular files, and apply the general file similarity measure that breaks file content into chunks, and compares files by the hashes of their chunks. We use Rabin fingerprinting [29] to define the boundaries of chunks, and SHA-1 to hash the data chunks.

An important parameter in the textual similarity is the size of the chunks, which determines the granularity in which similarity is defined. Choosing too small chunks, for example using a single instruction as a chunk, or choosing too large chunks, for example using the whole file as a chunk, would misrepresent the true program similarity. Using the average subroutine size in the executable is a reasonable choice, because similarity defined in this way approximately reflects the fraction of common subroutines shared between two programs. Therefore, we determine the chunk size from the distribution of subroutine sizes in our benchmark programs.

Once the chunk size C is determined, Rabin fingerprinting is used to break each file F into content-defined chunks of average size C, and SHA-1 is computed for each chunk. Thus the file F is abstracted into a list of SHA-1 values $\{B_1, B_2, \ldots, B_n\}$. The similarity between file $F = \{B_1, B_2, \ldots, B_n\}$ and $F' = \{B'_1, B'_2, \ldots, B'_m\}$ is defined as,

$$tsim(F, F') = \frac{1}{n+m}(\sum_{i=1}^{n} \delta(B_i, F') + \sum_{i=1}^{m} \delta(B'_i, F))$$

where $\delta(B_i, F') = 1$ if B_i is in F', and 0 otherwise.

5.4.2 Results

The same benchmark programs in Table 1 are used in our evaluation. The binaries of JVMs are directly obtained from the vendors' website since their sources are not available. The binaries of GCCs and ICC are built from their sources using compiler gcc-3.4.6 on i386 platform. Using different compilers or different optimization levels to

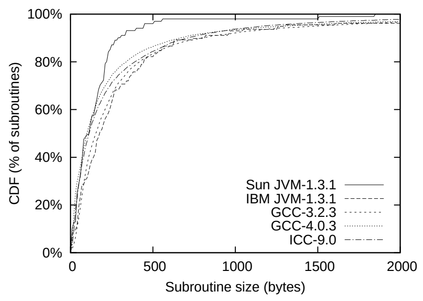


Figure 8: The size of subroutines in benchmark programs.

Table 3: Textual similarity between different JVM executables.

Similarity (%)		Sun			IBM	
		1.3.1	1.4.1	1.4.2	1.3.1	1.4.2
Sun	1.3.1	100.0	3.6	27.6	3.9	2.9
	1.4.1	3.6	100.0	1.5	1.1	0.9
	1.4.2	27.6	1.5	100.0	6.6	5.3
IBM	1.3.1	3.9	1.1	6.6	100.0	11.7
	1.4.2	2.9	0.9	5.3	11.7	100.0

produce the binaries would make the textual similarity scheme perform worse, because different compiler setting may produce totally different binaries even for the same source. We choose to use the same compiler on the same platform, and hence our workload is favorable towards the textual similarity technique, as the results are expected to estimate the best case of similarity detection. All the benchmark programs have functionality placed across multiple binary files. In our similarity comparison, we include for each program one of the binary files containing the main functionality. For JVM, the executable `java` is selected for both Sun and IBM JVMs. For GCC, the C compiler executable `cc1` is selected. For ICC, the functionally equivalent counterpart `mcpcom` is selected.

In our evaluation, the symbol table embedded in an executable is used to identify the subroutines. Note that defining the chunk boundaries using subroutines instead of Rabin fingerprinting is not feasible. The reason is that symbol tables are usually stripped off the executables by the software vendors. Also note that including symbol tables in our experiments makes little difference to our similarity results, because they are small compared to the whole executable.

Figure 8 shows the subroutine sizes in five selected benchmark programs. On average, 85% of the subroutines are smaller than 500 bytes, and 82% are larger than 32 bytes. Therefore, to be conservative, we use 32 bytes as the chunk size to capture the subroutine-level similarity.

Table 2 and Table 3 show the textual similarities between different GCC/ICC executables and JVM executables respectively. These similarities are quite low, even between the versions that only differ in release number. We even see a misleading result that the Sun JVM 1.3.1 is shown less similar to its own JVM 1.4.1 than to IBM JVM 1.3.1. These results show that textual similarity between executables is not suitable for characterizing source-level program similarity.

We further analyze why minor differences at the source level causes low textual similarity. The reason is that the translation from source code to binary code by a compiler, is a non-linear process. For example, subroutines are not translated independently of each other and a change to one subroutine affects the code for other subroutines. For example, if the starting address of a subroutine changes, all the subroutines calling this subroutine, even if their source code remains unchanged, are affected because the call sites now jump to the new address. To validate this conjecture, we disturbed our Rabin fingerprinting program by inserting a dummy subroutine at the beginning, and calculated the textual similarity between the executables before and after the disturbance. Our Rabin fingerprinting program consists of more than 600 lines of C/C++ code. The dummy subroutine simply does a `printf`, and is not invoked anywhere. The similarity between the executable before and after disturbance is 63.2%. After looking closely into the subroutines we obsreve that more than 90% of the subroutines are translated differently at their call sites because of the disturbance. The unchanged subroutines are those that do not call any other subroutines.

In summary, we have studied static textual similarity between program executables, and show its ineffectiveness in characterizing program similarity. Some previous research papers have studied the similarity/redundancy between software distributions using the same technique based on Rabin fingerprinting. Most of them studied the similarity on the source trees, and reported high accuracy [26, 23, 28]. Our study does not contradict these results since even in situations where the source level similarity is high, direct comparison of the binary code may be quite low.

6 Related Work

In this section, we discuss related work on detecting program similarities, on using system call information for tracking program execution, and on statistical analysis of program behavior.

6.1 Detecting Program Similarities

Moss [5] is a widely-used tool to detect plagiarism, primarily plagiarism in programming assignments. It is generally believed that Moss first tokenizes a source code into token sequence and then compares the token sequences between source programs. The implementation details are not released publicly to prevent circumvention. A comparison algorithm is presented in [30]. In circumstances where the source code is not available, Moss and other techniques [10, 31] that detect program plagiarism by various source code comparison approaches fail to work. In contrast, the statistical analysis approach presented in this paper is a black-box approach that does not require direct access to source code; our approach uses runtime information instead of static information to calculate software similarities.

Baker and Mander [8] present an approach to detecting similarities of Java programs represented as bytecode files, without accessing the Java source code. They exploit the fact that bytecodes from different programs conform to the same Java virtual machine application binary interface (ABI), and they use the disassembled bytecode for text analysis. Their approach does not work for native binary executables. Our approach uses black-box based runtime behavior without the difficulty of disassembling a binary. Tools like .RTPatch[1] can be used to update a software binary, but they cannot report how similar two applications are.

Software watermarking [17] is a technique used to combat software piracy by embedding identifying information into a program. Static software watermarking [14] is a technique where a software watermark recognizer examines the code or data segment of the executable program. However, the embedded static watermark can easily be destroyed by semantic-preserving transformations [13]. A dynamic watermark [12] utilizes the dynamic behavior (semantics) of the program, and is more resilient to semantic-preserving transformations. However, the runtime behavior representing the watermark is behavior added to the original (and inherent) program runtime behavior, *e.g.,* [11]. More knowledgeable attackers can extract these "added" features and destroy them with the help of a reverse engineering tool. The statistical approach discussed in this paper calculates the software similarity without the need of adding additional information to the released software, but instead uses information that is inherent to the program execution.

[1] `http://www.pocketsoft.com/product.html`

6.2 Tracking Program Execution using System Call Information

Various forms of system call information have been used as system call signature in different studies. There are basically two types of information, *content*, such as call number and call argument [18], and *context*, such as call site program counter and call stack [19, 20, 21, 24]. We used content as our system call signature and used call stack depth, a context property, to determine the prior distribution determining the aggregation of n-gram information.

System call information has been used by previous studies in various domains, including anomaly detection [18, 19, 24], prediction of access patterns by analyzing the correlation of I/O behavior with program context [20] and power management [21]. In [18], sequences of system call numbers are used to characterize normal program behavior and detect an intrusion. System call sequences are broken into patterns of fixed length, which are learned and stored. In [19], a call graph is built incrementally by inspecting the call stack when system calls are made and is used to characterize normal program behavior. In [24], system call program counters of an executable are first learned by statically analyzing the binary, and then used at run time as a white list to block illegitimate system calls issued by remote injected code, which limits the scope of remote code injection attacks. In [20], system call program counters are used to correlate the program context in which I/O system calls occur and the associated I/O access patterns. Such correlations are then used to predict the I/O access patterns in future occurrences of the same call sites for efficient buffer cache management. In [21], path-based correlation is used to observe a particular sequence of call site signatures leading to each idle period, and then subsequently used to predict future occurrences of that idle period. To the best of our knowledge, we are the first to leverage system call information to extract source level program similarities.

6.3 Statistical Analysis of Program Behavior

Many of the ideas underlying the expected similarity measure and n-gram analysis originated in the speech recognition, statistical text mining, and information retrieval fields. The n-gram representation has been widely used in speech recognition, language modeling, and later in web search engines and text classification. The monographs [25, 22] contain an overview of these fields.

A main difference between the n-gram analysis of SCS and text or speech analysis is that the number of possible system calls is not as large as the number of possible words, and therefore SCS analysis typically deals with a significantly lower dimensionality. This enabled the use of higher order n-grams for analyzing SCS. In text and speech, on the other hand, the extremely high dimensionality restricts the use to all but a few very low n-gram orders (typically $n \leq 3$) using alternative aggregation techniques such as back-off or linear interpolation [25, 22]. The particular use we make of cosine similarity to measure vector similarity is borrowed from a standard term frequency (tf) cosine similarity measure in information retrieval [7]. We chose that particular measure out of a wide variety of alternatives since it leads to the efficient quadratic time computation described in Section 3.6.

7 Conclusions

In this paper, we have proposed a new approach to detecting program similarities that requires neither the availability of the program source nor complicated compile-time watermarking techniques. Our approach exploits the key observation that the sequence of system calls invoked by a program provides a strong signature of the program semantics or functionality.

We have presented a set of statistical techniques that aggregate similarity information across different n-order interactions into a simple similarity measure between 0 and 1. Experiments on compiler systems and Java virtual machines validate the similarity measure, and comparison with the Moss source code similarity measure indicates striking consistency between the two methods. However, in contrast to source code methods like Moss, system call analysis is not sensitive to code obfuscation and does not require the source code itself.

We considered several prior distributions that correspond to different aggregations of n-gram similarities. Our experiments indicate that aggregating information across multiple n-grams is more effective than focusing on a single n-gram. The Moss comparison experiment indicates that a Gaussian fitted maximum likelihood estimator to the valley-width histogram provides the most striking similarity to Moss which leads us to consider this estimator as the preferred prior estimation method.

We also explored how the similarity measure can be used to dynamically map similar program regions onto textually similar source files. A case study shows how this framework can be used to pinpoint pirated segments of a program, and that the regions in the program source identified as similar by the framework correlate with those regions shown to be similar by Moss.

A potential weakness is that an attacker can insert dummy system calls into the program thus mutating or corrupting the system call sequence. However, such system call insertion can be manually detected by investigating the local similarity diagrams which will contain narrow bands separating a series of high peaks. We are currently examining methods to both collect sequences and perform statistical analysis that ignores calls in the code that cannot correspond to calls in the stolen code, either because files cannot be matched under isomorphism of system call arguments (*e.g.,* file names), or the calls are of the types that are unseen in the original source code. In general, insertion of a massive amount of fake system calls will eventually foil the detection mechanism. On the other hand, that is unlikely to happen since it will result in a significant slowdown of the software.

Finally, the principle of using inherent properties of the semantics of a particular version of a program to identify the code can be generalized. In particular, calls to standard library routines can be used instead of system calls. Thus calls to routines within `libc` or standard numerical libraries can be used to identify similar code regions. This will allow our technique to be used on codes that make relatively few system calls, such as numerical codes.

References

[1] GCC, the GNU Compiler Collection. `http://gcc.gnu.org/`.

[2] IBM, Java technology. `http://www.ibm.com/developerworks/java`.

[3] Intel C++ Compiler. `http://en.wikipedia.org/wiki/Intel_C++_Compiler`.

[4] Sun, Java techology. `http://java.sun.com/`.

[5] A. Aiken. Measure of software similarity.

[6] G. Arboit. A method for watermarking java programs via opaque predicates. In *International Conference on Electronic Commerce Research*, 2002.

[7] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

[8] B. S. Baker and U. Manber. Deducing similarities in java sources from bytecodes. In *USENIX Annual Technical Conference*, 1998.

[9] P. J. Bickel and K. A. Doksum. *Mathematical Statistics - Basic Ideas and Selected Topics*. Prentice Hall, 2001.

[10] X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker. Shared information and program plagiarism detection. *IEEE Transactions on Information Theory*, 50, 2004.

[11] C. Collberg, E. Carter, S. Debray, A. Huntwork, C. Linn, and M. Stepp. Dynamic path-based software watermarking. In *ACM Conference on Programming Language Design and Implementation*, 2004.

[12] C. Collberg and C. Thomborson. Software Watermarking: Model and Dynamic Embeddings. In *ACM Symposium on Principles of Programming Languages*, 1999.

[13] C. S. Collberg and C. Thomborson. Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection. *IEEE Transactions on Software Engineering*, 28(8), 2002.

[14] P. Cousot and R. Cousot. An abstract interpretation-based framework for software watermarking. In *ACM symposium on Principles of programming languages*, 2004.

[15] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley-Interscience, second edition, 2006.

[16] I. Csiszar. The method of types. *IEEE Transactions on Information Theory*, 44, 1998.

[17] D. Curran, N. J. Hurley, and M. O Cinneide. Securing java through software watermarking. In *International Conference on Principles and practice of programming in Java*, 2003.

[18] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *IEEE Symposium on Research in Security and Privacy*, 1996.

[19] D. Gao, M. K. Reiter, and D. Song. Gray-box extraction of execution graphs for anomaly detection. In *ACM conference on Computer and Communications Security*, 2004.

[20] C. Gniady, A. R. Butt, and Y. C. Hu. Program-counter-based pattern classification in buffer caching. In *Proc. of OSDI*, 2004.

[21] C. Gniady, Y. C. Hu, , and Y.-H. Lu. Program counter based techniques for dynamic power management. In *Proc. of the 10th International Symposium on High-Performance Computer Architecture*, 2004.

[22] F. Jelinek. *Statistical Methods for Speech Recognition*. MIT Press, 2001.

[23] Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *Proc. USENIX Annual Technical Conference*, 2004.

[24] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, and J. H. Hartman. Protecting Against Unexpected System Calls. In *Proceedings of the 14th Usenix Security Symposium*, 2005.

[25] C. Manning and H. Schutze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[26] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Symposium on Operating Systems Principles*, 2001.

[27] G. Myles. *Using software watermarking to discourage piracy*. ACM Press, 2004.

[28] Calicrates Policroniades and Ian Pratt. Alternatives for detecting redundancy in storage systems data. In *Proc. USENIX Annual Technical Conference*, 2004.

[29] Michael O. Rabin. Fingerprinting by Random Polynomials. Technical report, TR-CSE-03-01, Harvard University, 1981.

[30] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local Algorithms for Document Fingerprinting. In *ACM SIGMOD*, 2003.

[31] M. J. Wise. Detection of Similarities in Student Programs: YAP'ing may be Preferable to Plague'ing. In *SIGCSE technical symposium on Computer science education*, 1992.