A Dynamic Birthmark for Java

David Schuler, Valentin Dallmeier, and Christian Lindig

Universität des Saarlandes, 66 123 Saarbrücken, Germany
{ds,dallmeier,lindig}@cs.uni-sb.de

Abstract. Code theft is a threat for companies that considers code as a core asset. A birthmark can help them to prove code theft by identifying intrinsic properties of a program. Two programs with the same birthmark are likely to share a common origin. Birthmarking works in particular for code that was not protected by tamper-resistant copyright notices that otherwise could prove ownership.

We propose a dynamic birthmark for Java that observes how a program uses objects provided by the Java Standard API. Such a birthmark is difficult to foil because it captures the observable semantics of a program. In an evaluation, our API Birthmark reliably identified XML parsers and PNG readers before and after obfuscating them with state-of-the-art obfuscation tools. These rendered existing birthmarks ineffective, such as the Whole-Program-Path Birthmark by Myles and Collberg.

1 Introduction

Code represents for many companies a core asset that needs to be protected. However, code theft is difficult to prove: for over three years now, the SCO Group and IBM battle in court over code that allegedly belongs to SCO but was distributed by IBM as part of Linux. To protect code, companies may use *watermarking* (Collberg and Thomborson, 1999). Watermarking embeds a copyright notice into a program that is hard to detect and to remove but easy to reveal by the code owner. Even without such precaution, a company still may employ *birthmarking* after a suspected code theft. A birthmark identifies *intrinsic properties* of executable programs that are hard to change but easy to validate. While not a proof, similar birthmarks suggest a common origin.

Birthmarks can be split into two categories. *Static* birthmarks extract properties of program code. Many of them are vulnerable to simple program transformations, like code motion or renaming of registers. *Dynamic* birthmarks capture the *behavior* of a program and are much less affected by code transformations. For example, the Whole-Program-Path Birthmark by Myles and Collberg captures the control flow of a program. Nevertheless, such birthmarks may still be susceptible to obfuscations.

Rather than focusing on program behavior in isolation, we are proposing a dynamic birthmark that captures *observable semantics*: our API Birthmark observes how a program uses the Java Platform Standard API[1]. It builds on

[1] which we also call *Java Standard API*, *Java API*, or simply *API*.

two observations. First, Java programs are shipped as bytecode that still relies on functionality provided by the Java Runtime Environment. The Java Runtime Environment (which includes the Java Virtual Machine and packages like `java.io`) resides on the target machine. It is therefore not under the control of an application developer and hard to manipulate. Second, to have an effect, a Java program *must use the API* since opening a file or creating a window requires primitive services provided only by the Java Runtime Environment.

Here is an example of how the API Birthmark works: it found that the Xerces XML parser instantiates objects from classes `Vector` and `Stack` (both part of the API). These objects receive, among others, the following sequences of five method calls:

`Vector`	$\hookleftarrow$	⟨`removeAllElements`, `addElement`, `addElement`, `size`, `elementAt`⟩
`Stack`	$\hookleftarrow$	⟨`size`, `push`, `push`, `isEmpty`, `pop`⟩
`Stack`	$\hookleftarrow$	⟨`Stack`, `removeAllElements`, `size`, `removeAllElements`, `size`⟩

These call sequences are highly characteristic for Xerces. None of the other five XML parsers that we looked at showed these call sequences. At the same time, even after we obfuscated Xerces with Sandmark and Zelix KlassMaster—two bytecode obfuscators—we could still retrieve these sequences. Their presence strongly suggests that we are running code from the Xerces implementation.
The contribution of our paper is threefold:

- The API Birthmark is the first birthmark that captures fine-grained *observable semantics.*
- The API Birthmark is *credible* and *resilient* to the best commercial and academic obfuscators. We present the most *thorough evaluation* of a birthmark to date and demonstrate the *viability of birthmarking.*
- The API Birthmark is *more robust* and *more scalable* than the Whole-Program-Path (WPP) birthmark by Myles and Collberg (2004).

The remainder of this paper introduces the idea behind the API Birthmark (Section 2), its implementation (Section 3), and evaluation (Section 4). As an alternative scenario to program theft, we look into the detection of library theft (Section 5). Next, we present a comparison between our API Birthmark and the WPP birthmark (Section 6). Finally, we discuss possible attacks against the API Birthmark (Section 7), related work (Section 8), and our conclusions (Section 9).

2 The API Birthmark

The API Birthmark observes how a program interacts at runtime with objects from the Java Standard API. Such a birthmark is called *dynamic* because it depends on the program *and* its actual input. Two programs can be compared using their birthmarks: similar birthmarks are taken as evidence that the two programs are also similar.

2.1 Capturing API Calls

To capture a program's API usage, the birthmark observes method calls that are issued by objects from the program and received by objects from the API[2]. A sequence of method calls is called a trace.

Definition 1 (Call and Call Trace) *A method call $C.m$ is a pair of a class C and a method m invoked on an instance of C. A finite sequence of method calls is called a trace T, which is denoted by $T = \langle C_1.m_1, C_2.m_2, \ldots, C_n.m_n\rangle$.*

We obtain such a trace by letting every object from the API collect the trace of calls invoked from objects outside the API. We denote the trace of calls received by an object o with $T(o)$.

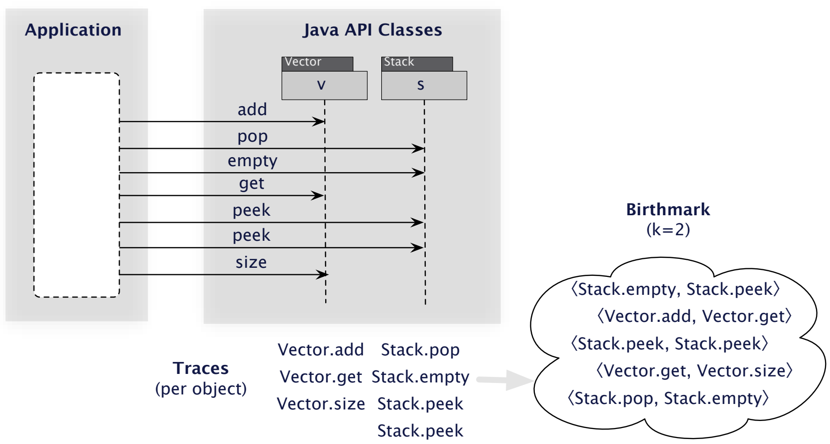


Fig. 1. A birthmark is a set of short call sequences received by API objects.

For example, Figure 1 shows a schematic sequence of method calls. The user program is shown on the left, and objects from the Java Standard API are shown on the right. Each object from the Java API collects the trace indicated below it, for example the trace for `Stack` object `s` is ⟨`Stack.pop, Stack.empty, Stack.peek, Stack.peek`⟩

Traces collected on a per-object basis are difficult to compare across program runs because they are huge. As a solution, we abstract each trace into a compact *call-sequence set.*

Definition 2 (Call-Sequence Set) *The* call-sequence set *for a trace T is defined as the set of all k-long substrings of T:*

$$S(T, k) = \{w \mid w \textit{ is a substring of } T \textit{ and } |w| = k\}$$

[2] Method calls between API objects are ignored.

The call-sequence set abstraction is compact and easy to compare. The size of such a set is bound by window size k and the number m methods in a class: at most m^k k-grams exist. However, this limit is almost never reached in practice. Loops in programs cause highly repetitive call sequences which in turn let call-sequence sets saturate quickly.

The idea of chopping up a trace using a sliding window to enable comparison was used by us previously (Dallmeier et al., 2005) for defect localization. We took this inspiration from Forrest et al. (1997)'s article on intrusion detection. Similar techniques have been used to detect similarities in source and other files (Manber, 1994; Schleimer et al., 2003).

2.2 The Birthmark

Computing one call-sequence set per API object leaves us still with many such sets per program run. To enable comparison between program runs, we define the birthmark as the union of all call-sequence sets from API objects. The API Birthmark for the situation in Figure 1 is the set shown on the right side of the figure.

Definition 3 (Birthmark) *A birthmark $B(P, I, k)$ is the union of all k-long call sequences observed by API objects during the run of program P with input I.*

$$B(P, I, k) = \bigcup_{o} S(T(o), k) \quad \textit{where } \textit{class}(o) \in \textit{API}$$

The central operation on birthmarks is comparing them for similarity. We compare two birthmarks A and B by computing the ratio of sequences found in both birthmarks versus the total number of sequences.

Definition 4 (Birthmark Similarity) *The similarity s of birthmarks A and B is defined as*

$$s(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Similarity is symmetric and yields a value from interval $[0, 1]$, where 0 indicates disjoint birthmarks and 1 indicates identical birthmarks.

The quality of a birthmark depends crucially on the definition of similarity. A good birthmark should *detect copies* of programs, as well as indicate low similarity between independently written programs but must be resilient to program obfuscation. We are discussing these points below in Section 4 where we evaluate the API Birthmark.

3 Birthmark Implementation

To extract a birthmark from a program, we first statically instrument the bytecode of the program as well as the bytecode of the Java API classes and then run the program. The instrumentation detects for each API object the methods invoked from the program. From this information the birthmark is computed at runtime in memory and written to a file when the program terminates.

3.1 Program and API Instrumentation

Instrumenting bytecode rather than source code is essential for birthmarking commercial code for which source code might be unavailable. We chose to instrument code prior to running it (rather than while it is running) because it works in the presence of custom class loaders and it ensures that no code is missed.

Our implementation is based on *method interposition* (Jones, 1993), a technique commonly used for tracing system calls. The key idea is to replace each API call site in the user program with a call to a proxy method that was added to the API class, which requires instrumentation of both the API and the program itself. Using method interposition, we capture all method calls from the user program to the API, whereas API-to-API calls remain unaltered.

```
class Vector {
  // original method
  void add(Object element) {...}
  // overloaded proxy method
  void add(Object element, CallerInfo caller) {
    CalleeInfo callee = new CalleeInfo(⟨Vector.add⟩, objectID);
    Tracer.addCall(caller, callee);
    add(element);
  }
}
```

Fig. 2. API instrumentation overloads each API method with a proxy method. A method `m`'s identifier is denoted by $\langle m \rangle$. The concrete object is identified by the newly introduced field objectID, which is initialized in the constructor. Instrumentation happens in bytecode but is here shown in source code.

```
class Main {
  public void processArgs(String[] args) {
    Vector v = new Vector();
    // was: v.add(args[0]);
    CallerInfo caller = new CallerInfo(⟨Main.procesArgs⟩);
    v.add(args[0], caller);
  }
}
```

Fig. 3. Program instrumentation redirects API calls to proxy methods by augmenting them with caller information.

An example of API instrumentation is illustrated in Figure 2. The `Vector.add` method is overloaded with a proxy method that takes an additional parameter of type `CallerInfo`. This parameter makes the method signature unique and provides information about the caller of the method. The implementation of the proxy updates the `Tracer` with information about the call and invokes the actual implementation of `add`.

Instrumentation of program classes is illustrated in Figure 3: the original invocation of `Vector.add` is replaced by code that creates an object identifying the caller plus the code that invokes the proxy method. This instrumentation is done for all invocations of API methods in the program code, such that they are redirected to proxy methods. Calls that originate inside the API remain unaltered and are not considered for the birthmark.

3.2 Computing the Birthmark

Proxy methods and call rewriting together provide the `Tracer` class with all information required for birthmark computation. The class maintains a trace window for every API object created during the run and a set of call sequences observed so far. Whenever a proxy method is invoked, we determine the receiver object of the call and update its trace window. If a new sequence was generated by the call, it is added to the call-sequence set. Upon shutdown of the virtual machine, the call-sequence set is written to disk.

Computing call-sequence sets at runtime reduces the amount of data that must be kept in memory and is more efficient than writing a raw trace of method invocations. A potential drawback of this approach is that extracting birthmarks for different window sizes requires running the program multiple times.

3.3 Robustness and Validation

In our evaluation of the API Birthmark, we have tested our implementation with several large Java projects ranging from 7 to over 900 classes. Our implementation was able to instrument all tested programs. However, the instrumented version of `javac` from the SPEC JVM'98 benchmark expects different input than the original one (cf. Section 3.4).

The correctness of the instrumentation was validated using the bytecode verifier of the virtual machine and by comparing the results of instrumented and unchanged program runs. To validate the correctness of birthmark extraction, we used a set of randomly generated programs for which the birthmark could be computed statically. Comparing expected and observed birthmarks revealed some subtle bugs that we fixed, such that we are now confident that our implementation extracts the correct birthmark.

3.4 Overhead

In order to assess the runtime overhead of the instrumentation, we compared execution times for programs from the SPEC JVM'98 benchmark suite (SPEC, 1998). The suite is a collection of programs commonly used to measure the performance of a Java virtual machine. We took seven programs and compared execution times for instrumented and unaltered versions. The results are summarized in Table 1.

We excluded program `javac` because it compiles programs against the API (in our case modified) of the JVM it runs on. This would require changing the input data, which leads to incomparable results.

Table 1. Runtime overhead of the API Birthmark for the SPEC JVM 98 benchmark.

	check	compress	jess	db	mpegaudio	mtrt	jack
Original (sec)	0.18	10.79	4.25	19.08	6.65	2.85	5.36
Instrumented (sec)	0.54	12.60	44.98	252.80	20.36	9.38	78.87
Overhead (factor)	3.00	1.17	10.58	13.25	3.06	3.29	14.71

The runtime overhead introduced by the birthmark computation ranges from a factor 1.17 for `compress` to 14.71 for `jack`. While considerable, the overhead is acceptable for a birthmark. A birthmark is only employed in a suspected case of program theft and has no impact on the production version of a program. This is in contrast to watermarking where overhead affects every program execution.

4 Evaluation

The primary purpose of a birthmark is to *detect copies* of a program. It therefore should indicate high similarity between identical programs. But to be *credible*, it should also indicate low similarity between independently written programs. In addition, a birthmark should be *resilient* to semantics-preserving program transformations: a birthmark should find program P and its transformed variant P' to be copies.

Two programs P_A and P_B with birthmarks A and B are *classified* according to their birthmark's similarity $s(A, B)$ and a bound ϵ:

$$s(A,B)\begin{cases}\geq 1-\epsilon & P_A \text{ and } P_B \text{ are classified as copies, or}\\ \leq \epsilon & P_A \text{ and } P_B \text{ are classified as independent, or}\\ \text{otherwise} & \text{no classification, it is inconclusive.}\end{cases}$$

The quality of a birthmark is characterized by the number of wrong classifications (unclassified programs, and programs incorrectly classified as copies or independent) for a given ϵ. A value of $\epsilon = 0.2$ was used by Myles (2006); smaller values are desirable but may lead to more false classifications.

4.1 Evaluation Setup

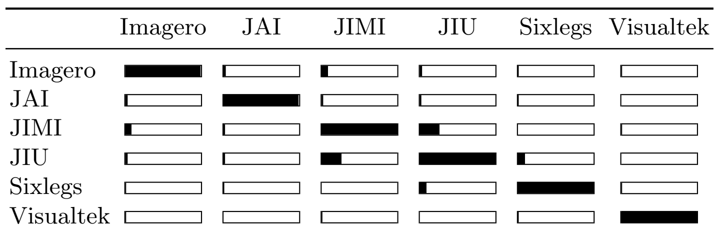
To evaluate our API Birthmark, we analyzed the birthmarks for a group of programs providing similar functionality. The first group consists of six programs

Table 2. Evaluation Subjects.

PNG Library				XML Parser			
	Version	Classes	Bytecode[1]		Version	Classes	Bytecode[1]
Imagero	1.80	916	1038	Aelfred	Saxon 7.0	7	59
JAI	1.1.2_01	476	3276	Crimson	1.1.3	145	347
JIMI	1.0	324	741	OracleV2[3]		343	1193
JIU	0.13	230	787	Piccolo	1.04	87	315
Sixlegs	2.0-rc3	39	74	Xerces	2.6.1	723	1791
Visualtek[2]		12	40	XP	0.5	97	176

[1] in Kilobytes [2] part of Genographer 1.6 [3] part of XDK 9.2.0.6.0

Table 3. Similarity between PNG readers ($k = 5$). Similarity from 0 (distinct) to 1 (identical) is indicated by the length of a black bar; e.g. 0.75 corresponds to .



which read PNG images and was used during development of the birthmark. The programs have been thoroughly exercised over many and varied inputs of about 100 images from the PNG Suite (van Schaik, 1996). The second group consists of six XML parsers, and was used to check the results obtained for the PNG readers; we used the SAXBench to run them (Oren and Slominski, 2002). Table 2 provides additional details for our evaluation subjects.

We conducted the experiments with sequence length $k = 5$, which provides a good trade-off between significance of sequences and runtime overhead (see Section 4.5 for a discussion). Furthermore, we filtered calls to commonly used classes (namely `java.lang.Object`, `java.lang.String`, and `java.lang.StringBuffer`) as these classes rarely indicate special behavior.

4.2 Detection of Copies

The ability to detect genuine copies is the most crucial ability of a birthmark. For an evaluation we executed each program twice with the same input and compared the birthmarks of these two runs.

The diagonal of Table 3 shows similarities between identical PNG readers, indicated by horizontal bars. The API Birthmark generally found perfect similarity of 1.0; except for Imagero and JAI, where it found a similarity of 0.99. This is caused by threads: objects that are shared across threads produce different call sequences when thread schedules differ among program runs. The corresponding results for the XML parsers are shown in the diagonal of Table 4. For this group similarity is perfect.

From the results for both program groups we are able to conclude that the API Birthmark can be used reliably to detect program copies for $\epsilon = 0.2$.

4.3 Credibility

To evaluate the credibility of the API Birthmark, we compared birthmarks pairwise within each program group. Here we expect the birthmark to find low similarity between distinct programs.

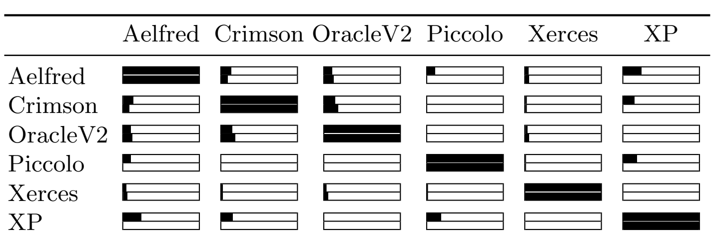
Table 3 shows the results for comparing PNG readers. The bar graphs in the table give the two programs according to the API Birthmark. For example, comparing Sixlegs with JIU yields a similarity of 0.09. The highest similarity measured for distinct programs was 0.26 for JIU and Sixlegs.

The pairwise similarity between XML parsers is captured in Table 4. The table indicates two values presented as horizontal bars for each pair. The top bar gives the similarity for the API Birthmark. Compared to the results for PNG readers in Table 3, the similarity found for distinct programs is higher but still very good in most cases. For $\epsilon = 0.2$, we have one case that is inconclusive for the PNG readers, and one for the XML parsers.

All XML parsers provide access to the parsed data via the SAX interface, which is part of the Java API. Since our test setup uses the default handler for SAX events (which is also part of the API), the sequence of calls to the default handler was very similar for all parsers. If we ignore these sequences for our birthmark, most similarity values for distinct programs drop noticeably (e.g. from 0.24 to 0.00 for Aelfred compared to XP). The values for filtered SAX calls can be found in Table 4, represented as the bottom bar for each pair of programs. With filtered calls, we have no more inconclusive cases ($\epsilon = 0.2$).

These improvements highlight a possibility to fine-tune our approach. The default handler is an example for a class whose methods can only be called in a certain sequence. By ignoring these classes for our birthmark we can further improve credibility. However, it may be difficult to identify these classes.

Table 4. XML parsers ($k = 5$). Similarity for each pair is indicated by two bars: one with SAX calls filtered out (bottom) and one without filtering (top).



4.4 Resilience

In order to disguise the origin of a program, a thief may apply semantics-preserving transformations. A birthmark must be resilient to such transformations, i.e. the birthmarks for the modified and the original program should be equal.

The preferred method to evaluate a birthmark's resilience (Tamada et al. (2004a), Myles and Collberg (2004)) is by using so-called obfuscators to simulate attacks. An obfuscator applies semantics-preserving transformations to a program to harden it against reverse engineering; it produces a semantically equivalent but not identical program.

In our evaluation we conducted a study with two obfuscators. Sandmark (Collberg et al., 2003) is an academic framework that implements 33 different obfuscation techniques, of which eight were stable enough to handle all our subjects. Zelix KlassMaster is a commercial obfuscator with a focus on minimal performance overhead, which we chose because of its reputation of being the strongest commercial obfuscator (Lai, 2001).

For our study, we used 11 different versions of each program (six PNG readers, six XML parsers): an unmodified version, nine Sandmark-obfuscated versions (one for each stable obfuscation technique and one with all techniques applied successively), and a Zelix-obfuscated version. We ran each version with the same input, extracted their birthmarks, and compared each obfuscated birthmark against the unmodified one.

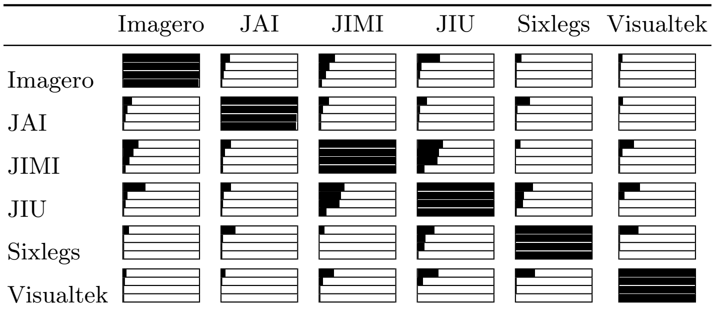
Except for Imagero and JAI, all birthmarks of obfuscated programs were identical to that of the original program. The deviations for JAI and Imagero are due to multiple threads running concurrently (cf. Section 4.2). Our results therefore indicate that the API Birthmark is resilient against transformations as applied by state-of-the-art obfuscators. For $\epsilon = 0.2$ we find no misclassification or inconclusiveness.

Code obfuscators apply program transformations like renaming, class splitting, or method merging. These are likely to have an effect on an application's static code properties or its control flow but not on its interaction with the Java API. Birthmarks that observe these static or dynamic properties are thus much more likely to be affected by these obfuscation techniques. On the other hand, the interaction with the Java API as it is observed by the API Birthmark is much harder to manipulate. We are thus confident that the resilience that we observed is no coincidence. Still—see Section 7 for a discussion of potential attacks against the API Birthmark.

4.5 Various Sequence Lengths

In our previous experiments we used a window size of five. However, it is also possible to compute the API Birthmark for different sequence lengths. In order to investigate the impact of the sequence length on credibility, we compared the birthmarks resulting from different sequence lengths for the PNG readers. Table 5 provides the results for sequence lengths 1, 3, 5, and 8.

Table 5. Pairwise similarity between PNG readers for various sequence lengths (k). The topmost bar of each cell gives the results for sequence length 1 and the following ones for 3, 5, and 8.



Shorter sequences cause birthmark similarity between distinct programs to increase. However, for increasing sequence lengths the multi-threaded programs (Imagero and JAI) decreases. We have chosen five as default sequence length since moving to longer sequences does not seem to offer much benefit; it provides a good trade-off between the ability to separate distinct programs and runtime overhead. This observation is backed up by our previous experience with call-sequence sets (Dallmeier et al., 2005).

5 Program Theft vs. Library Theft

The previous section evaluated the scenario of whole-program theft. Another scenario is library theft, where a stolen library is used as part of a new program. To complement our evaluation, we investigated the ability of the API Birthmark to detect whether a program uses a given library.

Detecting that a program incorporates a certain library with the API Birthmark is a challenge for two reasons: first, as the library is only a part of a program, API call sequences unrelated to the library introduce noise. Second, it may be impossible to find test cases for the program that use the library in the same way as the original library. In a limited experiment we looked at both issues.

5.1 Detecting a Library

Our first experimental setup includes four programs that each uses a programmatic interface of a PNG reader from Section 4. We compared the birthmarks for each of these four programs to the birthmarks of all six PNG libraries. The likelihood that a program uses a library is high if the birthmark contains many sequences that are also part of the library's birthmark. We measure this likelihood as the number of library birthmark sequences that also occur during the execution of the program.

Table 6. Detection of libraries. The actual library is marked with a double frame.

	PNG Reader Library					
Application	Imagero	JAI	JIMI	JIU	Sixlegs	Visualtek
DAOI						
ImageJ						
Jitac						
JSky						

The results of our evaluation are given in Table 6. For each program the similarity for the library actually used is marked with a double frame. Indeed, in all cases we found this library had the highest similarity across all six libraries. The amount of identical sequences is higher than 67%, whereas sequences for libraries that were not used constitute 36% or less. Many sequences occurring in both birthmarks is a good indicator that a program uses a certain library.

5.2 Impact of Input

In our second experiment we measured the similarity between two programs executed with different input. We split the input data in two halves (of about 50 images each), ran the PNG readers once with each halve as input, and compared the birthmarks from these runs. The results are given in Table 7.

Table 7. Similarity between programs executed with different input ($k = 5$).

	Imagero	JAI	JIMI	JIU	Sixlegs	Visualtek
Imagero						
JAI						
JIMI						
JIU						
Sixlegs						
Visualtek						

Similarities between identical programs (on the diagonal in Table 7) are lower than for two runs with the same input (Table 3). Similarities are still higher than between unrelated programs and thus give a strong hint when a certain program was used. Our birthmark is thus not overly sensitive to input.

6 Comparison to WPP Birthmark

The Whole-Program-Path (WPP) Birthmark by Myles and Collberg (2004) is the most recently proposed dynamic birthmarking technique. Whole-program paths are a technique used to compact a program's dynamic control flow (Larus, 1999). We used the implementation of the WPP Birthmark available in the Sandmark tool and compared it to the API Birthmark. For the comparison we could only use the Sixlegs image reader (comprised of 39 classes), since the implementation of the WPP Birthmark cannot handle larger programs. From this image reader we produced an obfuscated version using Zelix KlassMaster. While both birthmarks can be used to identify the original program, *only the API Birthmark is able to identify the obfuscated version*: between original and obfuscated copy the WPP Birthmark indicated a similarity of 0.08, whereas the API Birthmark indicated 1.0. For now, our API Birthmark is therefore both *more scalable* and *more resilient* than the WPP Birthmark.

7 Attacking the API Birthmark

If the API Birthmark became popular, attackers were likely to take counter measures. A simple attack against the API Birthmark would be to add additional sequences. For example, by introducing new API objects and calling methods on them. As this would preserve existing characteristics, only noise would be added.

A more sophisticated attack could manipulate existing sequences—for example by introducing new calls on existing objects. This is limited to calls that have no effect. Another option would be to add new calls together with calls that undo the effect of the former. Finding such methods or sequences is a problem in itself and hard to automate.

Another possible attack is to incorporate parts of the API implementation into the program. Then these parts could be obfuscated like the rest of the program, rendering many calls to the API obsolete. Through the remaining calls, this obfuscation would tie the program to a specific Java implementation. Also, calls to API methods expect regular API types, such that conversion code would be needed.

However, all these attacks introduce code and runtime overhead. Sophisticated attacks like incorporating the API also require manual work that diminishes the economic advantage of an attacker.

8 Related Work

Clone detection aims to find similarity between source code fragments for software maintenance. Advanced techniques (as for example proposed by Krinke (2001)) abstract from the textual representation of source code but still assume its availability, work statically, and do not assume the presence of sophisticated obfuscation techniques.

Birthmarking is related to fingerprinting and watermarking (Collberg and Thomborson, 1999), two other methods to detect software theft. Both work by embedding a copyright notice into an executable prior to its release. Extracting the copyright notice from a watermarked or fingerprinted program therefore constitutes a proof about its origin. Birthmarking may be applied without prior preparation but offers just a strong hint about a program's origin, not a proof. Of all three methods—watermarking, fingerprinting, and birthmarking—watermarking is understood best and only few birthmarks have been proposed to date.

All embedded marks may be extracted statically or dynamically. Collberg and Myles have shown that static marks tend to be vulnerable to basic program transformations. In particular, the static birthmark proposed by Tamada et al. (2004a) was shown to be susceptible to known transformations in Myles and Collberg (2005). The same paper introduced a k-gram based static birthmark that collects instruction sequences —similar to the technique that we use to collect calls dynamically. Their evaluation shows that, while superior to Tamada's birthmark, the k-gram static birthmark is still strongly affected by some known obfuscations.

Tamada et al. (2004b) presented a dynamic birthmark that is based on the raw trace of calls to the Windows API. Like our API Birthmark, the birthmark is in a position to capture the observable semantics of a program. However, the paper offers no substantial evaluation of its applicability, credibility, or resilience. These omissions are filled by this paper; furthermore, we have detailed how to efficiently cope with the volume of data resulting from observing calls and how to take advantage of the object-oriented nature of Java.

For the WPP Birthmark by Myles and Collberg (2004) we have shown that our API Birthmark is more resilient against obfuscators (cf. Section 6).

9 Conclusions

Our API Birthmark captures how a Java program uses objects from the Java API at runtime. We have shown in the first substantial evaluation of a birthmark (with the SPEC JVM'98 benchmark, six PNG readers, and six XML parsers as subjects), that this interaction is highly characteristic for a program. It is also efficient to compute and immune to today's program obfuscation techniques. The birthmark therefore can reliably identify the origin of code.

Unlike prior work, the API Birthmark does not capture a program's behavior in isolation. Instead, it captures the interaction with its environment and, hence, its observable semantics. It is thus much harder to foil by obfuscation techniques that solely change the inner workings of a program and, as a consequence, it is more robust. In particular, we have shown that the API Birthmark scales better and is more robust than the Whole-Program-Path Birthmark.

Our future work will concentrate on the detection of library theft. For this, as well as additional details about this paper, please refer to:

`http://www.st.cs.uni-sb.de/birthmarking/`

Acknowledgements Tom Zimmermann, Silvia Breu and Andreas Zeller gave valuable feedback on earlier revisions of this paper.

Bibliography

Christian Collberg, Ginger Myles, and Andrew Huntwork. Sandmark — A tool for software protection research. *IEEE Security & Privacy*, 4(1):40–49, 2003.

Christian Collberg and Clark Thomborson. Software watermarking: models and dynamic embeddings. In *Proc. of the Symp. on Principles of Programming Languages 99*, pages 311–324. ACM Press, 1999.

Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for Java. In Andrew P. Black, editor, *Proc. of 19th European Conf. on Object-Oriented Programming*, number 3586 in LNCS, pages 528–550. Springer, 2005.

Stephanie Forrest, Steven A. Hofmeyr, and Anil Somayaji. Computer immunology. *Communications of the ACM*, 40(10):88–96, 1997.

Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proc. of the 14th ACM Symp. on Operating System Principles*, pages 80–93. ACM Press, 1993.

Jens Krinke. Identifying similar code with program dependence graphs. *Working Conf. on Reverse Engineering (WCRE)*, pages 301–309, 2001.

Hongying (Jenny) Lai. A comparative survey of Java obfuscators available on the internet. Student summer project 415.780, University of Auckland, Computer Science Department, 2001. `http://www.cs.auckland.ac.nz/~cthombor/Students/hlai/`.

James R. Larus. Whole program paths. In *Proc. of the ACM SIGPLAN 1999 Conf. on Programming Language Design and Implementation*, pages 259–269. ACM Press, 1999.

Udi Manber. Finding similar files in a large file system. In *Proc. of the USENIX Winter 1994 Technical Conf.*, pages 1–10. Usenix Association, 1994.

Ginger Myles. *Software Theft Detection Through Program Identification*. PhD thesis, University of Arizona, Department of Computer Science, 2006.

Ginger Myles and Christian S. Collberg. Detecting software theft via whole program path birthmarks. In Kan Zhang and Yuliang Zheng, editors, *Proc. of the 7th Int. Conf. on Information Security*, volume 3225 of *LNCS*, pages 404–415. Springer, 2004.

Ginger Myles and Christian S. Collberg. K-gram based software birthmarks. In Hisham Haddad, Lorie M. Liebrock, Andrea Omicini, and Roger L. Wainwright, editors, *Proc. of the 2005 ACM Symp. on Applied Computing*, pages 314–318. ACM, 2005.

Yuval Oren and Aleksander Slominski. SAXBench, 2002. URL `http://piccolo.sourceforge.net/bench.html`.

Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proc. of the 2003 ACM SIGMOD Int. Conf. on Management of Data*, pages 76–85. ACM Press, 2003.

SPEC. SPEC JVM 98 benchmark suite. Standard Performance Evaluation Corporation, 1998.

Haruaki Tamada, Masahide Nakamura, Akito Monden, and Ken ichi Matsumoto. Design and evaluation of birthmarks for detecting theft of Java programs. In *Proc. of the IASTED Int. Conf. on Software Engineering*, pages 569–575, 2004a. Innsbruck, Austria.

Haruaki Tamada, Keiji Okamoto, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. Dynamic software birthmarks to detect the theft of Windows applications. In *Proc. Int. Symp. on Future Software Technology 2004*, 2004b.

Willem van Schaik. PNG Suite, 1996. URL `http://www.schaik.com/pngsuite/pngsuite.html`.

Zelix Pty Ltd. Zelix ClassMaster. `http://www.zelix.com`, 2006. Java bytecode obfuscator.