

Delft University of Technology
Software Engineering Research Group
Technical Report Series

Proceedings
Third International Workshop on
**Code Based
Software Security Assessments**
— CoBaSSA 2007 —

October 31, 2007, Vancouver, Canada
co-located with 14th Working Conference on Reverse Engineering
(WCRE 2007)

Organized by
Leon Moonen and Spiros Mancoridis

Report TUD-SERG-2007-023

TUD-SERG-2007-023

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:

<http://www.se.ewi.tudelft.nl/techreports/>

For more information about the Software Engineering Research Group:

<http://www.se.ewi.tudelft.nl/>

All papers are copyright © 2007 by their respective authors.

This collection was edited by Leon Moonen <Leon.Moonen@computer.org>

Copyright © 2007, Software Engineering Research Group, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology.

All rights reserved. No part of this series may be reproduced in any form or by any means without prior written permission of the publisher.

Table of Contents

Overview & Introduction	2
CoBaSSA 2007 Program	2
Workshop Background	2
Motivation, Topics, and Goals	3
Organizers	3
 Keynote: The Good, the Bad, and the Ugly From 10 Years of Vulnerability Prevention - <i>Crispin Cowan</i>	 4
 Searching for Malware - <i>Ira Baxter</i>	 5
 Information Flow Control and Taint Analysis with Dependence Graphs - <i>Jens Krinke</i>	 6
 Software Security through Targeted Diversification - <i>Nessim Kisserli, Jan Cappaert, Bart Preneel</i>	 10
 Identifying Source Code Authorship - <i>Robert Lange, Jonathan Max-Sohmer, Maxim Shevertalov, Jay Kothari, Spiros Mancoridis</i>	 14

Overview & Introduction

CoBaSSA 2007 Program

08:30	welcome & participant intro
08:45	Keynote: The Good, the Bad, and the Ugly From 10 Years of Vulnerability Prevention <i>Crispin Cowan</i>
09:45	Searching for Malware <i>Ira Baxter</i>
10:10	break
10:25	Information Flow Control and Taint Analysis with Dependence Graphs <i>Jens Krinke</i>
10:50	Software Security through Targeted Diversification <i>Nessim Kisserli, Jan Cappaert, Bart Preneel</i>
11:15	Identifying Source Code Authorship <i>Robert Lange, Jonathan Max-Sohmer, Maxim Shevertalov, Jay Kothari, Spiros Mancoridis</i>
11:40	global discussion
11:55	wrap-up
12:00	end

Workshop Background

Our technological society has become more and more dependent on software that is used to automate everyday processes. This dependence increasingly exposes us to the security threats that originate from malicious software (malware) such as computer viruses and worms and software vulnerability exploits such as remote execution of code or denial of service attacks. Moreover, this exposure is not limited to computer systems but is spreading to common appliances such as mobile phones, PDAs and consumer electronics such as media centers, personal video recorders, etc. since a growing number of these products are made extensible and adaptable by means of embedded software.

The proliferation of malware and exploits requires that action is taken to tackle these issues and evaluate software security to prevent the damage and costs (e.g., data loss, productivity loss, recovery time) that result from security incidents. This calls for measures to assure that a software system has the desired security properties, i.e. that it is free of malware and vulnerabilities. In addition, there is a need for technology for software forensics, for example to detect code authorship or plagiarism.

Workshop History

The First International Workshop on Code Based Software Security Assessments was held in 2005, also co-located with WCRE. CoBaSSA 2005 drew 20 participants from academia and industry/government (approx. 50/50). One of the workshop's results was a top 10 list of open issues in software security research jointly collected, refined and prioritized by workshop participants. The workshop proceedings and results are available from the CoBaSSA 2005 homepage¹. Evaluation indicated a great interest in the return of

¹<http://swer1.tudelft.nl/leon/cobassa2005/>

CoBaSSA as co-located event of WCRE. All participants felt it was worthwhile to bring together the people in our community that work on this particular subject.

Motivation, Topics and Goals

The purpose of this workshop is to bring together practitioners, researchers, academics, and students to discuss the state-of-the-art of software security assessments based on reverse engineering of source or binary code (as opposed to software security assessments that look at the software process that was applied). This includes research on topics like source & binary code analysis techniques for the detection of software vulnerabilities (e.g. detect if code has potential buffer overflow problems) or analysis for the detection of malicious behavior (e.g. detect if code contains an exploit or has viral behavior).

CoBaSSA topics of interest include, but are not limited to:

- Mitigating stack- or heap-based buffer overflow attacks
- Software forensics
- Re-modularizing legacy code for privilege separation
- Race condition detection
- Vulnerabilities in trust management and authentication
- Code tamper-proofing and obfuscation
- Decompilation and disassembly techniques
- Anti-debugging measures
- Copy protection schemes
- Analysis of computer virus and worm code
- (Case studies in) evaluating software vulnerability
- Best practices practices for secure coding

The goal is to share experiences, consolidate successful techniques, collect guidelines, and identify opportunities for future work and collaboration. We will do so by building on the list of open issues in software security research collected, refined and prioritized by participants of CoBaSSA 2005.

Workshop Organizers

CoBaSSA 2007 is organized by:

- Leon Moonen (Delft University of Technology, The Netherlands)
- Spiros Mancoridis (Drexel University, USA)

Keynote:

The Good, the Bad, and the Ugly From 10 Years of Vulnerability Prevention

Crispin Cowan

This talk presents a retrospective montage from 10 years of research into techniques to prevent intrusion by blocking the exploitation of software defects. Topics covered include compiled-in intrusion prevention (StackGuard, FormatGuard, and PointGuard), kernel intrusion prevention (RaceGuard), access controls (AppArmor/SubDomain) and leveraging the community to patch defects (Sardonix and Time to Patch).

The talk draws from the following papers

- "Linux Security Modules: General Security Support for the Linux Kernel". Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Presented at the 11th USENIX Security Symposium, San Francisco, CA, August 2002.
- "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade". Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. DARPA Information Survivability Conference and Expo (DISCEX), Hilton Head Island SC, January 2000. Also presented as an invited talk at SANS 2000, Orlando FL, March 2000.
- "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks". Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Published in the proceedings of the 7th USENIX Security Symposium, January 1998, San Antonio, TX.
- "RaceGuard: Kernel Protection From Temporary File Race Vulnerabilities". Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-Hartman. Presented at the 10th USENIX Security Symposium, Washington DC, August 2001.
- "FormatGuard: Automatic Protection From printf Format String Vulnerabilities". Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. Presented at the 10th USENIX Security Symposium, Washington DC, August 2001.
- "PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities". Crispin Cowan, Steve Beattie, John Johansen and Perry Wagle. Presented at the 12th USENIX Security Symposium, Washington DC, August 4-8, 2003.

which can be downloaded from <http://crispincowan.com/%7Ecrispin/>

Searching for Malware

Ira Baxter (Semantic Designs, Inc)

To control cost, military systems are starting to acquire software components from a variety of suppliers. These components must not be subverted during usage. In a military context, one must even assume that the supplier may have a subverted development process, in which malware might be inserted during development, with serious efforts to hide the malware. Acquired software must be inspected for malware.

This talk will discuss preliminary research on an interactive method for locating malware that combines deep static analysis capability with interactive support for a security analyst. The static analysis capability will build on DMS's ability to parse languages and construct control and data flow. The static analysis capability provides traceability to/from artifacts of interest to the analyst. The analysis provides understanding of key system actions and predicates on which the malware may trigger.

Information Flow Control and Taint Analysis with Dependence Graphs

Jens Krinke
FernUniversität in Hagen
Hagen, Germany

1. Introduction

Ensuring that the integrity of critical computation is not violated by untrusted code or the confidential data is protected is a complex problem for current software systems. We can observe two main directions to approach the problems:

- For critical system, formal approaches are needed. One is (static) *information flow control* which analyzes the software to check if it conforms to some security policy. An example is *noninterference*: secret information does not influence the publicly observable behavior of a system.
- Many informal approaches can be subsumed under *bug detection*. A violation of some security policy can be regarded as a bug and therefore many bug detection approaches do some kind of taint analysis. Data from untrusted sources (e.g. the user) is tainted and is not allowed to reach exploitable functions like system calls vulnerable to buffer overruns.

There is a large overlap in both directions. Let's consider taint analysis as described in the later direction: Taint analysis is just a special case of information flow control and noninterference (tainted information does not influence vulnerable parts of the system). Moreover, it is a very simple form as it just has a security policy with two levels "tainted" and "untainted".

There is a major difference in both directions: The first is conservative while the later is optimistic. This means that the formal approaches do not allow false negatives: If the noninterference check does not reveal a violation, it is guaranteed that the analyzed code does not contain leaks. On the other hand, bug detecting approaches prefer an optimistic approach that allows false negatives. This has two reasons: First, this reduces the amount of false positives, and second, practicable solutions (as in tools) often require unsound approaches.

It is not surprising that both directions have almost disjoint research communities. For example, the recent comprehensive survey by Sabelfeld and Myers [11] about information flow security does include 147 references, but none

of them is related to taint analysis for bug detection. However, a closer examination reveals that both communities can benefit from each other. Especially as both communities make heavy use of program analysis for their approaches.

For information flow control, many approaches use program analysis in terms of special type systems that ensure noninterference or other security policies for well-typed programs. Such type systems are usually extensions or modifications of real world languages, examples are Jif by Myers et al. [6, 7], similar to Java, and Flow Caml by Simonet and Pottier [10, 12], similar to Objective Caml. However, as Zdancewic [15] has observed, information-flow based enforcement mechanisms have not been widely used.

In contrast, taint analysis for bug detection is more and more used for the analysis of real systems. Many of such approaches are based on a long record of research on program analysis for optimization and reengineering [1]. Usually, type systems are only one of the used analyses (most often used for efficient pointer analysis) and other, more complex representations are used. For example, the taint analysis approaches from Pistoia et al. [9] use program slicing [13] or the one from Livshits and Lam [4, 5] that uses IPSSA, an interprocedural variant of gated SSA [2, 8]. Wlandler discusses some more tools.

It seems that the bug detection approaches are somewhat more successful than the information flow control approaches. From our point of view, this is related to major difference that has not been pointed out by Zdancewic as a challenge [15]: Information flow control approaches and tools like Jif and Flow Caml rely on dedicated languages that support information flow control while the bug detection approaches analyze real world programs in standard languages like C or Java with additional security-related information provided. This enables security analyses on already existing software, whereas in information flow control approaches one needs to reimplement the system.

In the following we will present our approach for information flow control that is based on dependence graphs, using results from both major directions discussed above. It will enable more flexible security policies than taint analysis without for real world languages like C or Java, thus having the advantages of both directions.

2. Security levels and declassification

The simple two-level security policy of taint analysis is sufficient for simple problems, but in practice one wants more detailed information about security levels of individual statements. Thus theoretical models for IFC utilize a *lattice* $\mathcal{L} = (L, \sqcup, \sqcap)$ of security levels, the simplest consisting just of two security levels *High* and *Low*. We provide a specification option for the lattice, and an option to mark some (or all) statements with their security level. The approach is based on the program dependence graph representation [3]. Program statements or expressions are the graph nodes. A data dependence edge $x \rightarrow y$ means that statement x assigns a variable which is used in statement y (without being reassigned underway). A control dependence edge $x \rightarrow y$ means that the mere execution of y depends on the value of the expression x (which is typically a condition in an if- or while-statement). A path $x \rightarrow^* y$ means that information can flow from x to y ; if there is no path, it is guaranteed that there is no information flow. In particular, all statements influencing y (the so-called *backward slice*) are easily computed as $BS(y) = \{x \mid x \rightarrow^* y\}$. If there is no PDG path from a to b , it is guaranteed there is no information flow from a to b . This is true for all information flow which is not caused by hidden physical side channels such as timing leaks.

The security level of statement with PDG node x is written $S(x)$, and confidentiality requires that an information receiver must have at least the security level of any sender. In PDGs, this implies $\forall y \in \text{pred}(x) : S(x) \geq S(y)$ which ensures $S(y) \rightsquigarrow S(x)$. The dual condition for integrity is $\forall y \in \text{pred}(x) : S(x) \leq S(y)$. However, this assumes that every statement resp. node has a security level specified, which is not realistic. We want to specify *provided* as well as *required* security levels not for all statements, but for certain selected statements only. The provided security level specifies that a statement sends information with the provided security level and the required security level specifies that only information with a *smaller* security level may reach that statement. The provided security levels are defined by a partial function $P : N \rightarrow L$, where N is the set of nodes resp. statements of the programs. Thus, $l = P(s)$ specifies the statement's security level. The required security levels are defined similarly as a partial function $R : N \rightarrow L$. Thus, $P(s)$ specifies the security level of the information generated at s and $R(s)$ specifies the maximal allowed security level of the information reaching s . Information with security level l that is generated at some node x in the dependence graph, is propagated along the dependences and should not reach another node a which has a required security level which is smaller than l . Thus a program represented as a dependence graph does not violate confidentiality, iff

$$\forall a \in \text{dom}(R) : \forall x \in BS(a) \cap \text{dom}(P) : P(x) \leq R(a)$$

```

1 class PasswordFile {
2   private String[] names /*P:confidential*/
3   private String[] passwords; /*P:secret*/
4   public boolean check(String user,
5     String password /*P:confidential*/) {
6     boolean match = false;
7     for (int i=0; i<names.length; i++) {
8       if (names[i]==user
9         && passwords[i]==password) {
10        match = true;
11        break;
12      }
13    }
14    return match; /*R:public*/
15  }
16 }

```

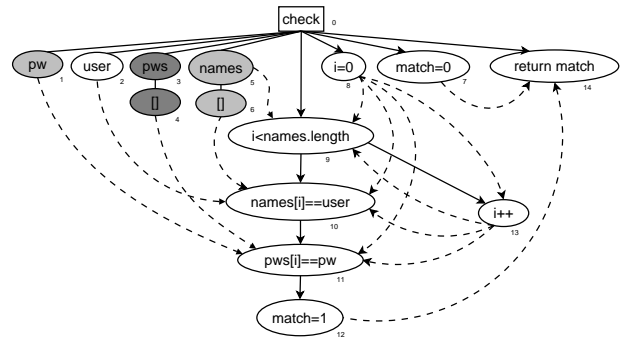


Figure 1. A Java password checker with its PDG

i.e. the backward slice from a node a with a required security level $R(a)$ must not contain a node x that has a higher security level $P(x)$.

Usually, the number of nodes that have a specified security level is low, e.g. points of output. Therefore, the above criterion can easily be transformed into an algorithm that checks a program for confidentiality:

PDG-Based Confidentiality Check. *For every node in the dependence graph that has a required security level specified, compute the backward slice, and check that no node in the slice has a higher provided security level specified.*

Checking each node separately allows a simple yet powerful diagnosis in the case of a security violation: If a node x in the backward slice $BS(a)$ has a provided security level that is too large ($P(x) > R(a)$), the responsible nodes can be computed by a chop $CH(x, a)$. The chop computes all nodes that are part of path from node x to node a , thus it contains all nodes that may be involved in the propagation from x 's security level to a .

As an example, consider the PDG for a password checking program (Figure 1). We choose a three-level secu-

urity lattice: *public*, *confidential*, and *secret* where *public* \rightsquigarrow *confidential* \rightsquigarrow *secret*. The list of passwords is *secret*, thus $P(3) = P(4) = \textit{secret}$. The list of names and the parameter *password* is *confidential*, because they should never be visible to a user. Thus, $P(1) = P(5) = P(6) = \textit{confidential}$. Figure 1 shows the annotated PDG: The security levels are depicted through white for *public*, light gray for *confidential*, and gray for *secret*. According to the criterion, we require that no confidential or secret information flows out of the method, thus we require the return statement to have a required security level of *public* ($R(14) = \textit{public}$). A backward slice for node 14 will reveal that nodes 1 and 3–6 are included in the slice and have a higher security level, thus a security violation is revealed.

In practice the above approach is too simple because in some situations one might accept that information with a higher security level flows to a “lower” channel. A typical example is the password checking method presented earlier: The result of the method will eventually be used to access the user’s private data or to output an error message that the login was not successful. Of course, that areas will not have the same security level (*secret*) as the list of passwords. *Declassification* allows to lower the security level of incoming information at specified points. In the example, declassification would reduce the security level at the return node 14 to security level *public* such that the result of the password check can be used in low security areas.

We model declassification by specifying certain PDG nodes to be declassification nodes: Let D be the set of declassification nodes. A declassification node $x \in D$ has to have a required and a provided security level: $r = R(x)$ and $p = P(x)$. Information reaching x with a maximal security level r is lowered (declassified) down to p . Now a path from node y to a with $P(y) > R(a)$ is not a violation, if there is a declassification node x on the path with $P(y) \leq R(x)$ and $P(x) \leq R(a)$ (assuming that there is no other declassification node on that path).

The above slicing solution no longer works with declassification, as information flow with declassification is no longer transitive and slicing is based on transitive information flow. This can easily be solved: First, we compute the backward slice for every node a with a required security level specified. By definition of a slice, no node outside this slice can have any influence on a . The subsequent analysis (see below) will only consider nodes and edges that are part of this slice, i.e. the dependence graph is reduced to the subgraph represented by the initial slice for a . This avoids spurious dependencies and false alarms caused by potential security violations outside the backward slice of a .

The analysis will then compute the actual required security level for every node in the (sliced) program dependence graph by a backward analysis. The actual required security level of a node is the maximal security level that may reach

the node without causing a security violation at the criterion node under observation.

The actual incoming (required) security level $S_{IN}(x)$ for a statement x is computed from the outgoing security levels of its successors $y \in \textit{succ}(x)$:

$$S_{IN}(x) = \begin{cases} \top & \text{if } \textit{succ}(x) = \emptyset \\ \prod_{y \in \textit{succ}(x)} S_{OUT}(y) & \text{otherwise} \end{cases}$$

At nodes without declassification, the outgoing security level is simply the incoming security level: $S_{OUT}(x) = S_{IN}(x)$. At declassification nodes $x \in D$ with a declassification from $R(x)$ down to $P(x)$, S is replaced with the new required level: $S_{OUT}(x) = R(x)$. Thus

$$S_{OUT}(x) = \begin{cases} R(x) & \text{if } x \in D \\ S_{IN}(x) & \text{otherwise} \end{cases}$$

These equations are data flow analysis equations and can be iteratively solved using a standard algorithm, with a proper initialization of S_{OUT} . The initialization is done based on the criterion node a under observation, i.e. it is checked that no security violation occurs due to $R(a)$:

$$S_{OUT}(x) = \begin{cases} R(a) & \text{if } x = a \\ R(x) & \text{if } x \in D \\ \top & \text{otherwise} \end{cases}$$

Due to the monotonicity of the computation and the limited height of the security level lattice, a minimal fixed point for S_{IN} is guaranteed to exist and can be computed using a standard iteration. The computed S_{IN} have then to be checked for confidentiality:

Confidentiality Check With Declassification. *For every node a in the dependence graph that has a required security level specified which is not a declassification node, compute the incoming security levels $S_{IN}(x)$ of all statements x in its backward slice and check the following property:*

$$\forall x \in \textit{dom}(P) \cap BS(a) : P(x) \leq S_{IN}(x) \quad (1)$$

Thus, for any $l = P(x)$ such that $l \not\leq S_{IN}(x)$ we have a confidentiality violation at x because $l \not\rightsquigarrow S_{IN}(x)$ (the security level l is not allowed to influence the required level of $S_{IN}(x)$). Note that it is $\not\leq$ and not $>$ because l and $S_{IN}(x)$ might not be comparable. Because the slice $BS(a)$ has been computed first, the confidentiality check can be done during the dataflow analysis: If the current node has a provided security level $P(x)$, a computed required level $S_{IN} \not\geq P(x)$ is a violation. Declassification nodes themselves are not considered as information sinks in the above check, even though they have to have a required security level.

Let us return to the example in Figure 1 and assume $R(14) = \textit{public}$. The computation to check confidentiality for criterion node 14 will start with a backward slice $BS(14)$. Because node 14 can be reached from every node

in the PDG, the slice will contain the complete PDG, thus it is not reduced. The subsequent computation of the actual required security levels will result in $S_{IN}(x) = public$ for all nodes x of the example. The confidentiality check will reveal violations at nodes 1 and 3–6 because for these nodes, the specified provided level is higher than the computed actual required.

Now assume the node 14 is a declassification node $secret \rightarrow public$: $14 \in D$, $R(14) = secret$, $P(14) = public$. The computation of the actual required security levels will result in $S_{IN}(x) = secret$ for $x < 14$. The confidentiality check will no longer reveal a security violation, which may be desirable depending on the security policy, since only a negligible amount of information leaks from password checking.

3. Taint analysis

The above presented approach has been implemented and we are currently evaluating it for confidentiality security policies. While preparing the evaluation for integrity security policies (to which taint analysis belongs), we discovered a disadvantage of the proposed approach. For taint analysis, we are able to specify the sources of untrusted data (e.g. user inputs) as with a provided security level of *tainted* and to specify the vulnerable functions to have a required security level of *untainted*. However, many operations or functions do a declassification from *tainted* down to *untainted*. For example, consider taint variable analysis for buffer overrun protection. The call of `snprintf` from the C-library prints its arguments to a buffer. However, it is guaranteed by `snprintf` that the buffer will not overrun. Thus, `snprintf` declassifies a security level of *tainted* in its arguments down to *untainted* in the buffer. Such a declassification can easily be specified within our approach by specifying the assignment to the buffer inside `snprintf` as declassification node.

However, there are other operations that turn tainted variables into untainted ones. An example is a simple check of the buffer with the user input against a maximal size (“if (`length(s) < MAX`)”). In the true branch, `s` is *untainted* and in the false branch, it is *tainted*. Such a declassification cannot easily be specified within our approach: Any use of variable `s` inside the true branch will have a direct data dependence to the *tainted* assignment to `s` and will result in a generated level of *tainted*.

We are currently investigating more flexible way to specify declassifications. One approach would be to use Wilander’s pattern matching on dependence graphs [14]. For the above example, the declassification would be specified as “a check of the length of a variable against a maximal size will declassify all accesses to the variable in the true branch to *untainted*”. As SSA form is needed here to prevent that the

check and the access are different instances of the same variable, another approach that is investigated by is to use gated SSA form.

References

- [1] B. Chess and G. McGraw. Static analysis for security. *IEEE Security & Privacy*, November/December 2004.
- [2] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst.*, 13(4):451–490, 1991.
- [3] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang. Syst.*, 9(3):319–349, July 1987.
- [4] V. B. Livshits and M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 317–326, 2003.
- [5] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX Security Symposium*, 2005.
- [6] A. C. Myers. Jflow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, 1999.
- [7] A. C. Myers, S. Chong, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow. <http://www.cornell.edu/jif>, 1999.
- [8] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 257–271, 1990.
- [9] M. Pistoia, R. J. Flynn, L. Koved, and V. C. Sreedhar. Interprocedural analysis for privileged code placement and tainted variable detection. In *ECOOP 2005 – 19th European Conference on Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, 2005.
- [10] F. Pottier and V. Simonet. Information flow inference for ml. *ACM Trans. Prog. Lang. Syst.*, 25(1):117–158, 2003.
- [11] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), Jan. 2003.
- [12] V. Simonet. Flow caml. <http://cristal.inria.fr/~simonet/soft/flowcaml/>.
- [13] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, July 1984.
- [14] J. Wilander and P. Fak. Pattern matching security properties of code using dependence graphs. In *Proceedings of the First International Workshop on Code Based Software Security Assessments*, pages 5–8, 2005.
- [15] S. Zdancewic. Challenges for information-flow security. In *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence (PLID'04)*, 2004.

Software Security through Targeted Diversification

Nessim Kisserli Jan Cappaert Bart Preneel

Katholieke Universiteit Leuven, Dept. of Electrical Engineering – ESAT Kasteelpark Arenberg 10, B-3001
Heverlee, Belgium

{nessim.kisserli, jan.cappaert, bart.preneel}@esat.kuleuven.be

Abstract—Despite current software protection techniques, applications are still analysed, tampered with, and abused on a large scale. Crackers¹ compensate for each new protection technique by adapting their analysis and tampering tools. This paper presents a low-cost mechanism to effectively protect software against global tampering attacks. By introducing diversity per programme instance, we illustrate how to defeat various patching methods using inlined code snippets. We propose an efficient technique for creating the snippets based on genetic programming ideas, and illustrate how our approach might trigger a small-scale arms race between defending and attacking parties, each forced to evolve in order to “stay in the game”.

I. INTRODUCTION

Piracy has plagued software vendors for years and still continues to do so. Efforts to thwart it have largely failed due to the inherently open architecture of current computing systems and the prevalence of a healthy software monoculture. Attempts to address the former can be seen in the current development of Trusted Protection Modules (TPMs), the latter impediment however, despite being acknowledged [10], remains mostly unaddressed.

Table I illustrates the cracking process’ various stages as described in [12] and the typical countermeasures employed by software vendors to counter each. We note the lack of any widespread protection mechanism targeting the automation stage and propose a low-overhead solution based on the following sober reflections:

- Users will always attempt to analyse software protection mechanisms, out of academic curiosity or otherwise.
- It is virtually impossible to prevent a user from modifying software and processes on a machine they control –given current architectures.
- It is impossible to police the Internet and remove the myriad sites currently hosting pirated software.

Software patches contain sufficient information to locate and replace a set of *critical instructions* within a programme. We propose to make automated patching sufficiently unreliable, and consider our approach successful if crackers:

- Are forced to distribute cracked full-binaries, or
- Develop automated patches whose size approaches that of the full-binary.

¹We use the term for individuals who strip copy-protection mechanisms from commercial software, enabling its unfettered use. They are also colloquially referred to in some circles as *warez d00dz* [13]

Both cases cause sites hosting pirated software to incur substantially higher bandwidth and storage costs. Additionally, the former allows software developers to leverage watermarking and other origin identification techniques such as [16].

This paper is structured as follows. Section II introduces *snippets*, the building blocks of our protection scheme. Section III explores increasingly sophisticated existing as well as hypothetical automated patching techniques, describing specialised snippets to counter each. We share our experiences generating such snippets using genetic programming in Section IV, and present related research and ideas for further work in sections V and VI respectively.

II. SNIPPETS

A snippet is a series of one or more assembly instructions designed to be inserted within an existing assembly programme which we refer to as *the host*. We distinguish three types of snippets based on the net effect of their execution on a host’s state. Informally, two instances of a programme are said to be in the same state if the following conditions hold:

- Their instruction pointers refer to the same instruction.
- Their stack pointers refer to the same stack offset.
- The contents of their respective registers are identical, including the flags register.

A snippet is termed *harmless* if executing it in a host at state σ_1 yields a new state σ_2 , identical to σ_1 . Intuitively such snippets exhibit similar properties to redundant code. Non-harmless snippets are further distinguished into semi-harmless and harmful ones. A *Semi-harmless* snippet is one which under certain well defined conditions, we call *insertion conditions*, can be rendered harmless. The task of locating in a host *insertion points* at which a snippet’s insertion conditions are satisfied is carried out by an *insertion function*. Snippets with a non-empty set of insertion conditions for which no insertion function can be found are *harmful*.

A. Immunity from code compaction

Being extraneous code, we must ensure the snippets are not readily identified using code compaction tools. There is a paucity of work on code compaction of compiled, stripped binaries as most available research focuses on compiler-generated parse trees with full access to authoritative source code, e.g. [5]. Our snippets can take the following forms:

- Redundant code: Code whose execution has no effect on the overall output of a programme.

TABLE I
THE CRACKING PROCESS AND TYPICAL COUNTERMEASURES

Stage	Purpose	Industry adopted protection mechanism
Analysis	Determine and locate protection mechanisms	Obfuscation
Tampering	Disable protection mechanisms	Tamper resistance
Automation	Apply the tampering to other instances of the software with minimal user knowledge and interaction	None
Distribution	Provide others with the ability to obtain cracked software	Legal threats + Termination of hosting

- Dead code: Code whose computed results are unused.
- Unreachable code: Code to which there is no control flow path. Determining whether an arbitrary code snippet is reachable is considered undecidable.

The use of various obfuscation concepts in our snippets (such as opaque predicates [3] for branch-confidentiality) can help reduce the threat of detection. While the remainder of this paper classifies snippets according to various criteria, we assume them all immune to detection by code compaction tools.

III. THE ARMS RACE

While the battle lines between crackers and software vendors are better defined in the *analysis* phase of the cracking process, they are no less present in the *automation* stage. In this section we systematically examine automated patching techniques in order of increasing complexity. For each method, we discuss required properties of both snippets and insertion functions to counter automation, further escalating the arms race.

As we approach the limits of sequential instruction-based comparisons, we explore more “exotic” methods such as graph-based structural programme analysis. While the patching methods in sections III-A and III-B are widely used by the cracking community, those outlined in sections III-C and III-D are, to the best of our knowledge, currently not.

A. Offset patches

1) *How they work:* As their name suggests, offset patches overwrite a number of bytes at a fixed file offset. They may employ various techniques to maximise successful patching, such as comparing the binary’s checksum against a “known good value” or verifying the instruction at the specified offset is the expected one. Their continued successful use by crackers is testimony to the inherent weakness of today’s software monoculture on the one hand, and the failure of current protection schemes to address all stages of the cracking process on the other.

2) *Defeating them:* Any modification to the offset of the critical instructions will suffice to defeat an offset patch. The main requirement is that the snippet be inserted before the critical instructions. Such snippets, which need not exhibit any special properties, we call *basic snippets*, and are in fact a superset of the more specialised harmless snippets introduced later.

B. Pattern searching patches

1) *How they work:* Pattern searching patches locate and replace specific byte patterns in a binary. They may also employ similar success maximising techniques to offset patches.

2) *Defeating them:* There are two main approaches to defeating pattern matching cracks:

- Destroy the pattern being searched for.
- Duplicate the pattern, inducing multiple false positives.

The first technique requires interleaving snippets with critical instructions. In the most extreme case, no two consecutive native instructions are allowed to remain in the critical section. Harmless snippets, by definition, can be inserted between any two instructions without disrupting the host’s functionality. A targeted insertion function is required for inserting the snippets between the critical section’s instructions. We call such snippets which actively destroy patterns in their host *parasitic snippets*.

The second technique requires the creation of snippets containing sequences of instructions identical to the critical ones. However, these instructions are most likely to be harmful and must first be “neutralised”. Use of opaque predicates can guarantee such instructions are always skipped (i.e. rendered dead code). We call these snippets which imitate their host *mimic snippets*.

We note that both approaches can be employed simultaneously for increased effectiveness.

C. Collusion-based patches

1) *How they work:* Collusion attacks refer to ones in which multiple parties share information about a protection scheme in order to defeat it. At its simplest, such an attack takes the form of a file comparison between two or more diversified instances of a programme to establish their commonality.

2) *Defeating them:* We propose a new kind of snippet for defeating such collusion attacks, the *poisoned snippet*, with the following properties.

- It is composed of two consecutive logical parts S_1 and S_2 . Crucially, when combined they form the harmless snippet $S = S_1 || S_2$ (where $||$ denotes concatenation).
- Taken separately, both parts are most likely harmful.
- A *generation* of poisoned snippets all share the exact same logical part S_x .

Poisoned snippets require the following insertion condition:

- All snippets in a generation are inserted adjacently to the same native instruction (we call a *border instruction*) in all instances of the diversified programme.

The above condition effectively renders the most likely harmful S_x -part of the poisoned snippet indistinguishable from

native instructions by including itself in the Longest Common Substring (LCS) spanning the border instruction.

D. Structural Analysis-based patches

1) *How they work:* Comparative structural analysis is generally cast as a graph matching optimisation problem. Two differing but similar executables E_1 and E_2 are represented as graphs G_1 and G_2 and an optimal isomorphism between the two is sought.

Nodes from each graph representing the same element in E_1 and E_2 , called *fixed points* in [6], are used to map each function in E_1 onto its counterpart in E_2 using function signatures (such as return, number and type of formal parameters). The same procedure is iterated for each function's basic blocks, then for each basic block's individual instructions until a partial best-fit isomorphism between G_1 and G_2 is achieved. The resulting bijection maps elements of E_1 to their semantic equivalents in E_2 . Recently both [6] and [15] have used structural analysis to highlight changes between different patch-levels of a binary.

It is still not clear whether this type of patch will be feasible. It incurs higher storage and bandwidth usage to access the graph of a known-good patched instance, and may be less economic than simply distributing a fully cracked binary.

2) *Defeating them:* Structural analysis based patching can be attacked in two ways:

- Obfuscating a programme's control and data flow, complicating initial graph construction.
- Minimising iteratively discoverable fixed points across programme instances.

The first approach can leverage existing obfuscation and anti-disassembly techniques. The latter requires the creation of *decoy snippets* providing fake fixed points upon which graphs are mapped onto each other. They follow the same principle as mimic snippets, but are structurally more complex as they must mimic both function signatures and control flow properties. To be effective, the insertion function must be coupled with targeted modifications to the host's original code. This is what we propose to explore in more detail in the diversifying compiler mentioned in section VI.

IV. SNIPPET GENERATION

Manually crafting snippets for our protection scheme presents a costly endeavour given their required number and specificity. Rather, we rely on genetic programming techniques to automate the task, finding the stochastic and evolutionary elements of the approach particularly appropriate.

In this section we share some of our experiences evolving various aspects of the snippet species we introduced in section III. We discuss our main fitness function, different *genetic operators* used, and illustrate some of the problems faced. Due to space constraints, we assume familiarity with the basic workings of a genetic algorithm.

A. Genetic operators

Genetic operators are the main drivers of diversification in genetic computing and are broadly divided into so called sexual and asexual types. The former traditionally combine traits from two parents to produce an offspring, while the latter mutate one individual into another.

a) *Mutation:* Asexual reproduction was limited in our experiments to infrequent mutations in which one of the following occurred:

- Two random instructions in a snippet were swapped.
- A register was substituted for another, globally within a snippet. For example, all references to **eax** changed to **ebx**.

The latter form was introduced in an attempt to curb the observed general destructiveness of the first mutation, and to allow for a limited template-like replication of harmless individuals.

b) *Reproduction:* We experimented with two operators, the classical *N-point crossover* and a more suitable, snippet-friendly *Insertion* operator. The former divides two snippets X and Y into n random parts X_1, \dots, X_n and Y_1, \dots, Y_n respectively, recombining them into two new child snippets $C_1 = X_1 || Y_2 || \dots || X_{n-1} || Y_n$ and $C_2 = Y_1 || X_2 || \dots || Y_{n-1} || X_n$. The operator was found to be extremely destructive in the majority of cases, including $n = 2$.

Starting from the observation that harmless snippets can be safely embedded between any two instructions, we developed the less destructive *insertion operator*. Here, one of the two parent snippets is chosen and randomly split into Y_1 and Y_2 . The remaining parent, X, is then inserted between the two parts, creating a new child snippet $C = Y_1 || X || Y_2$.

While particularly well suited at duplicating mimic snippets and overall less destructive, the insertion operator produces increasingly longer snippets which must be artificially constrained.

B. Snippet Simulation

In order to establish the effect of snippets on a host programme's state, we model various IA32 assembly instructions and a generic x86 little-endian compatible execution environment as follows:

- Several general purpose 32-bit registers modelled with bit-level precision.
- An extended 32-bit flags register ϕ .
- A programme stack and accompanying stack pointer ρ .

We initialise our pre-snippet execution environment σ_1 as follows:

- For each bit i in the extended flags register, set $\phi[i] = 0$.
- For each simulated register, reg set $\sigma_1(reg)[i] = reg[31], reg[30], \dots, reg[0]$.
- Initialise the stack's pointer ρ_{σ_1} to 0.

We then symbolically evaluate, with bit-level precision, the effect of each assembly instruction on our environment. While tracking changes to the stack is relatively straight forward, we rely on a bit vector *decision procedure* for the individual bits of

each register (we use the function rich STP [9] from Stanford university).

At the end of a snippet's execution our environment is in state σ_2 . Recall from section II that a harmless snippet is one which does not modify its original environment, i.e. one for which $\sigma_1 == \sigma_2$. Practically, for a harmless snippet:

- The stack pointer $\rho_{\sigma_2} == 0$.
- For each simulated register, *reg*, the symbolic value $\sigma_2(\text{reg})[i] == \sigma_1(\text{reg})[i]$ for $i=0$ to 31.

C. Flags

Up to now we have carefully avoided mentioning the flags register. This is the most problematic aspect of symbolic evaluation. The issue can be sidestepped by restricting ourselves to only modelling those instructions which do not set any flags, such as **push** and **pop**. This is not a practical solution however. Our current approach adds an appropriate restriction to the snippet's insertion condition set for each simulated instruction which may trigger a flag. Most such snippets are semi-harmless, requiring an adequate insertion function. However, certain snippets which make use of constant values or opaque predicates may escape such restrictions if the simulator is able to assert flag setting-conditions are not met.

1) *Insertion Conditions*: The following fragment for example, risks overwriting the Zero Flag (ZF) upon equality and the Carry Flag (CF) if $ebx > eax$:²

```
cmp eax ebx
jnz L1
...
```

The insertion conditions are thus:

- $\phi[ZF]$ is not live.
- $\phi[CF]$ is not live.

This is most likely the case right before a native **cmp** instruction.

V. RELATED RESEARCH

Diversification has been leveraged in many security solutions. Cohen explored the feasibility of using similar techniques to ours to increase operating systems' resistance to attacks in [2]. Forrest et al. sketched a multi-level holistic system diversification technique, including instruction block reordering and use of nonfunctional code in binaries [8]. More recently, Anckaert et al. discussed the logistics of providing updates to tailored software instances [1], and although conceptually similar to our idea, no details of the tailoring scheme were provided.

More generally, Address Space Layout Randomisation [14] (ASLR), System Call diversification, and Instruction Set Emulation have all been used as countermeasures to certain types of memory corruption and code-injection attacks. Cox et al. formalised the use of diversification and equivalent execution in their N-variant system [4], targeting similar classes of attacks. Finally, El-Khalil et al. used functionally equivalent instructions for steganographic purposes [7].

²For simplicity we assume unsigned operands.

VI. CONCLUSIONS AND FURTHER WORK

Besides modelling additional assembly instructions and exploring new opaque predicates, the problem of the flags register must be better addressed. We would like to incorporate our snippets into a diversifying compiler in order to influence and better exploit the layout of the native assembly. By producing keyed one-to-many mappings between the high-level *native instructions* and their assembly, snippets can be made harder to distinguish. Currently we lack automated insertion functions. A compiler lends itself fairly naturally to this task. Adding such functionality to the LLVM compiler framework [11] is therefore our next goal.

This paper introduced programme diversifying snippets, a light-weight software protection scheme designed to thwart patches relied on for low-overhead, mass distribution of pirated software. We presented several types of snippets targeting distinct patching approaches, and showed the feasibility of automating their creation using genetic programming techniques.

REFERENCES

- [1] B. Anckaert, B. De Sutter, and K. De Bosschere. Software Piracy Prevention Through Diversity. In *Proceedings of the 4th ACM workshop on Digital rights management* pp. 63–71, Washington DC, 2004.
- [2] F. Cohen. Operating system protection through program evolution. *Computers and Security*, 12(6):56–584, 1993.
- [3] C. Collberg, C. Thomborson, and D. Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Principles of Programming Languages 1998*, San Diego, CA, January 1998.
- [4] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *The 15th USENIX Security Symposium*, pp. 10–120, 2006.
- [5] S. K. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Language and Systems*, Ed. 22(2), March 2000.
- [6] T. Dullien and R. Rolles. Graph-based comparison of executable objects. *Symposium sur la Securite des Technologies de l'Information et des Communications*, 2005.
- [7] R. El Khalil and A. D. Keromytis. Hydan: Hiding Information Binaries. In *Proceedings of the 6th International Conference on Information and Communications Security*, pp. 187–199, October 2004, Malaga, Spain.
- [8] S. Forrest, A. Somayaji, and D. H. Ackley. Building Diverse Computer Systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pp. 67–72, 1997.
- [9] V. Ganesh and D. L. Dill. A Decision Procedure for Bit-Vectors and Arrays. *Computer Aided Verification*, Berlin, Germany, July 2007.
- [10] D. Geer et al. CyberInsecurity: The Cost of Monopoly, 2003.
- [11] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, Palo Alto, CA, 2004.
- [12] A. Main and P.C. van Oorschot. Software Protection and Application Security: Understanding the Battleground. *International Course on State of the Art and Evolution of Computer Security and Industrial Cryptography*, Heverlee, Belgium, June 2003.
- [13] E. Raymond(ed), The New Hacker's Dictionary, MIT Press, 1991.
- [14] The PaX Team. <http://pax.grsecurity.net/docs/aslr.txt>
- [15] T. Sabin. Comparing binaries with graph isomorphisms. *BindView RAZOR Team*, 2004. http://www.bindview.com/Services/Razor/Papers/2004/comparing_binaries.cfm
- [16] Julien P. Stern, G. Hachez, F. Koeune, and J. Quisquater. Robust Object Watermarking: Application to Code. *Information Hiding '99, volume 1768 of Lectures Notes in Computer Science* pp. 368–378, Dresden, Germany, 2000.

Identifying Source Code Authorship

Robert Lange, Jonathan Max-Sohmer, Maxim Shevertalov, Jay Kothari, Spiros Mancoridis

Software Engineering Research Group

Department of Computer Science

College of Engineering

Drexel University

3141 Chestnut St.

Philadelphia, PA 19104

Email: {jmaxsohmer, rlange, max, jhk39, spiros}@drexel.edu

Abstract—

I. INTRODUCTION

Stylometry, which is the application of the study of linguistic style, is often used to analyze the differences in the styles of authors of literary works. Researchers have identified about 1,000 characteristics, or style markers, such as word length, to analyze literary works [1]. Linguistics investigators have used stylometry to distinguish the authorship of written works by capturing, examining, and analyzing style markers [2]. Like naturally-evolved human languages, programming languages allow developers to express certain constructs and ideas in different ways. The differences in the way developers express

their ideas can be captured in their programming styles, which in turn can be used for author identification.

In the context of programming languages, it has been shown that capturing style in source code can help in determining authorship. In our previous work, we examined source code as a text document and identified certain peculiarities in the styles of software developers that persisted across different projects. We used those styles to determine the authorship of the source code. Using various text-based style markers such as line length and 4-character sequence distributions, we developed and matched profiles of authors [3]. We were able to use those profiles to determine authorship of unidentified code with a success rate over 80% on a testing data set of

24 open source projects, each authored by a single developer. We also experimented with identifying the authors of whole projects, rather than individual files, using developer profiles built from histograms of text-based source code metrics [4]. In this study, we used a genetic algorithm to find the best combinations of metrics to perform the identification.

II. SERG SOFTWARE FORENSICS TOOL

From our work, we have developed an application for identifying the authorship of source code files by classifying them against data gathered from files with known authors. Our application allows for the use of various classifiers, filters and metrics in order to provide a high level of adaptability and configuration to increase speed and accuracy. In addition to the library built into the tool, our application also allows for the addition of user-written plugins that conform to our interfaces for the extension of the tool's capabilities.

Although other tools for code authorship identification do exist, our application is unique in how it accomplishes the task. The primary difference is the use of machine learning to train our classifier; this allows the classifier to evolve as new information is added.

A. Release Targets

Our tool has two modes of operation which are targeted to specific uses. The "research" mode is used to test our algorithms and compare their results both in accuracy and speed. We use this to compare the effectiveness of different configurations so we can select the one best suited to our needs. It also helps us evaluate new algorithms. For this mode, we use one set of source files of known authorship for our training samples and another set to play the role of the "unknown" files so we can determine how accurately our tool is classifying them. Once we choose a configuration, the tool exports a file which is used for the configuration of the "production" mode.

The production mode for our tool loads its configuration from the file generated by the research mode to choose its options; most end-users do not have the domain-specific knowledge necessary to choose the best configuration so we hide the configuration step from the user in this mode and instead use the one from the file. This makes for a simpler interface which is better suited for the real-world application of identifying code authorship. Also, because the files being tested truly are unknown in this scenario, the unknown files are not grouped according to their actual authors but instead entered as a single set.

This white paper will focus primarily on the "research" mode of operation.

III. PROCESS COMPONENTS

The process of using our tool can be split into two major sub-processes: Preparation and execution.

A. Preparation

1) *Metrics*: In order to obtain characteristic data for the source code files, we run the selected set of metric extractors against each file and record the occurrences of specific metrics. Each metric extractor runs in sequence against the source files and the combined metrics from each file are stored for analysis. Thirteen metric extractors are packaged with the application, while more may be added via automated plugin loading.

The included metrics count the proportion of occurrences of textual and stylistic characteristics in the source code, including brace positioning, call chains, naming characteristics, leading tabs and character sequences. For a detailed explanation of the metrics in use, please refer to Kothari[3] and Lange[4]. The current metrics are intended for the analysis of Java source files; metrics specific to other programming languages could be added as plugins.

2) *Filters*: While using all of the data gathered from the known source files typically yields accurate results, the aggregated data about the files is huge and can take extremely large amounts of time and memory to classify. Because of the intended use for our tool, it would be unacceptable to have the initial classification of known source files require several days to run; we therefore employ the use of filters to narrow our data set down to a more manageable size.

The difficulty here is choosing which metrics to keep and which to discard; we need to select the most relevant and useful data from our huge data set. Discarding important data can greatly and adversely affect the accuracy of the tests, but at the same time we do need to eliminate a significant portion in order to bring the running time down to acceptable levels.

We use pluggable filtering functions to analyze the data and choose the most relevant metrics for further consideration. Depending on the filter implementation the running time can be cut substantially and, for some filtering methods, the discarding of irrelevant data can actually improve the accuracy of our final test results.

In addition to a "dummy" filter which allows the user to keep all data, we have included a filter which uses information entropy to choose the most relevant data. We use information entropy to remove metrics which have low between-author variability and high within-author variability[3]. This guarantees that the metrics in the filtered data set are only those which are good for classification. The use of the entropy filter does increase the pre-classification execution time, however we observed that it decreased classification time. We have also noticed substantial improvements in the accuracy of the unknown code classification when employing the entropy filter. However, the entropy filter does not appear to scale well when applied to very large data sets.

3) *Classifiers*: Once the data about our sample files have been gathered and the metrics filtered, we use the remaining metrics to create a classification database which records the attributes for each developer for comparison against the unknown files. We have chosen to use the open source WEKA framework [5] for the actual classification of the data as it

already has numerous classification algorithms built in and can be used as a library for our program; our tool gathers and prepares the data for the WEKA kit and interprets the results. We have chosen a small handful of Bayesian classifiers for use in our tool, but any of the WEKA classifiers could easily be added to the pool.

The use of different classifiers can have a substantial impact on the accuracy of our analysis; this version of the tool lets us compare the results from different classifiers so we can choose the best one to suit our needs.

B. Execution

Once the sample data is prepared, we use the selected metrics and the classifier to classify the unknown files by comparing their characteristic features to those of the sample files and picking the best match. The filter from the preparation is not applied directly to the testing files; instead, any metrics present in the test files that were dropped from the learning files by the filter are simply ignored as they can't be used for comparison.

1) *Results*: The test results are displayed in a tiered format on three tables. Each of the lower level tables contains more information about the results of the selected item in the table above it.

The overview displays lists for each developer profile used for testing the actual name of the profile, which profile it most looks like (these would match if the classification was correct), the number of correctly classified files for the profile and the total number of files tested.

The user can select a profile in the list to view the individual results of each file in the profile. The individual file information displays the file's name and location on the filesystem as well as the developer to which it actually belonged, the developer it most resembles, and the confidence of the classification. Selecting a file from that list will display in the bottom table the probability of that file belonging to each developer.

The production mode of the application skips the overview table as there is no known membership for the testing files in the scope of its use.

IV. NOTABLE FEATURES

Our tool implements several features to make its use easier and more convenient.

A. Multiple Data Sets

The research mode allows for multiple data sets to be managed simultaneously. This is especially useful for comparing the speed and effectiveness of various configurations used on the same learning files. Multiple data sets can be compiled sequentially with a single click, which allows the user to walk away from the program while it processes the data. Because each data set could take hours to build depending on its size and configuration, this provides a nice alternative to babysitting the application as it runs each test or wasting time by not starting another data set building immediately after the first one is completed.

B. Saving and Loading Data

To speed up the preparation time, progress for various different entities can be saved at each step of the preparation process. Individual developer profiles can be saved to a file for loading later, as can entire data sets. A data set can be saved at any time provided there's been a change since its last save; if the data set has already been built, that will be stored with the saved data set. Similarly, the most recent results of a test run are stored with the data set so they can be viewed later without rerunning the tests if the state is saved after the initial run.

In production mode, the session state is saved rather than an individual data set as the user does not work concurrently with multiple data sets.

C. Plugin Framework

The application supports Java (or Java-wrapped) plugins for both metrics and filters. External plugins must extend the abstract superclass for the plugin type and correctly implement its abstract methods.

V. LIMITATIONS

The following are two known limitations in the use of our tool:

A. Filesystem

Ideally, our tool would load the entire contents of each training file and store this with the data set. However, with large training sets this becomes prohibitively expensive in terms of memory usage and the size of saved data sets on disk. It would similarly affect individual saved profiles, which would also become bloated on the filesystem. In smaller test scenarios this would not be so much of an issue, but when dealing with a learning set of dozens of profiles each containing hundreds of learning files, it adds up and can become problematic.

In compromise, we have the files accessed by reference to their location on disk. While this considerably reduces space concerns, it limits the usefulness of saved data sets when transferred to other systems; unless the files are located in exactly the same path on the new system, the files cannot be accessed.

One important note is that, once a data set's classifier has been built, the metrics which have been extracted and maintained are saved with the data set directly. This is possible because a reasonably useful filter reduces the size of the retained data to a reasonable level.

VI. FUTURE WORK

There are three main areas of planned future work for our tool. The tool itself will be used to evaluate the effectiveness of future research and development.

A. Pluggable Classifiers

The current plugin framework only supports external filters and metric extractors; support for classifier plugins has not yet been implemented. Once this has been completed, any classifier which correctly implements WEKA's classifier interface will be usable with our tool, at which point the WEKA classifiers could be loaded from the plugins folder instead of explicitly invoked as they are currently.

B. Filter Research

Information entropy has proved to be an extremely useful means of paring down the metrics data, but it becomes prohibitively expensive to run on a very large dataset. Further investigation into data filtering is needed to improve the performance and usefulness of our application. Although the primary aim with the data filtering is an improvement to the speed of the application, an ideal filter also would greatly increase the accuracy of the classifications by removing .

Once we have additional filter plugins we can experiment with running multiple filters either sequentially or in parallel as well as the various combinations and how to use the combined results.

C. Metrics Research

We currently have a number of stylistic and textual analysis metric extractors. Metrics research will be divided into two major areas.

1) *Expanding Metric Library*: Current metrics extract information from the source code that can be obtained by textual analysis, lexical analysis, and parsing of context-free grammars. There are still a variety of stylistic and textual characteristics which could be measured and possibly used for classification. Furthermore, semantic analysis of source code is necessary to generate more advanced metrics. We intend to explore these further and expand our repertoire of useful metrics.

2) *Exploring New Types of Metrics*: Currently we are limited to source code analysis for identifying software authors. However, we cannot be assured of access to source code in some of the domains of application for our tool. Malware, for example, is often distributed in binary executable form, so identifying malware authors may require analysis of execution characteristics instead of code characteristics.

REFERENCES

- [1] F. Can and J. M. Patton. Change of writing style with time. *Computers and the Humanities*, 38(1):61–82, 2004.
- [2] D. I. Holmes. Authorship attribution. *Computers and the Humanities*, 28:87–106, 1994.
- [3] J. Kothari, M. Shevertalov, E. Stehle, and S. Mancoridis. A probabilistic approach to source code authorship identification. In *Proceedings of International Conference on Information Technology: New Generations*. IEEE, 2007.
- [4] R. Lange and S. Mancoridis. Using code metric histograms and genetic algorithms to perform author identification for software forensics. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, New York, NY, USA, 2007. ACM Press.
- [5] I. Witten, E. Frank, L. Trigg, M. Hall, G. Holmes, and S. Cunningham. Weka: Practical Machine Learning Tools and Techniques with Java Implementations. *ICONIP/ANZIIS/ANNES*, pages 192–196, 1999.

TUD-SERG-2007-023
ISSN 1872-5392

