BinComp: A Stratified Approach to Compiler Provenance Attribution

By

Saed Alrabaee, Paria Shirani, Mourad Debbabi, Ashkan Rahimian and Lingyu Wang

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2015 USA

Philadelphia, PA (Aug 9th - 13th)

DFRWS is dedicated to the sharing of knowledge and ideas about digital forensics research. Ever since it organized the first open workshop devoted to digital forensics in 2001, DFRWS continues to bring academics and practitioners together in an informal environment.

As a non-profit, volunteer organization, DFRWS sponsors technical working groups, annual conferences and challenges to help drive the direction of research and development.

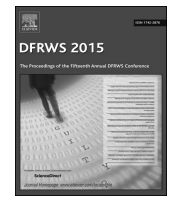
http:/dfrws.org

Digital Investigation 14 (2015) S146–S155

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin



DFRWS 2015 US

BinComp: A stratified approach to compiler provenance Attribution☆

Ashkan Rahimian, Paria Shirani, Saed Alrbaee*, Lingyu Wang, Mourad Debbabi

Computer Security Laboratory, Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Quebec, Canada

Keywords:
Compiler provenance
Reverse engineering
Binary program analysis
Digital forensics
Programming analysis

A B S T R A C T

Compiler provenance encompasses numerous pieces of information, such as the compiler family, compiler version, optimization level, and compiler-related functions. The extraction of such information is imperative for various binary analysis applications, such as function fingerprinting, clone detection, and authorship attribution. It is thus important to develop an efficient and automated approach for extracting compiler provenance. In this study, we present *BinComp*, a practical approach which, analyzes the syntax, structure, and semantics of disassembled functions to extract compiler provenance. *BinComp* has a stratified architecture with three layers. The first layer applies a supervised compilation process to a set of known programs to model the default code transformation of compilers. The second layer employs an intersection process that disassembles functions across compiled binaries to extract statistical features (e.g., numerical values) from common compiler/linker-inserted functions. This layer labels the compiler-related functions. The third layer extracts semantic features from the labeled compiler-related functions to identify the compiler version and the optimization level. Our experimental results demonstrate that *BinComp* is efficient in terms of both computational resources and time.

© 2015 The Authors. Published by Elsevier Ltd on behalf of DFRWS. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

Introduction

Program binaries are often the focus of forensic investigations, covering numerous issues ranging from copyright infringement to malware analysis. Program binary analysis can be extremely challenging due to the absence of high level information, which is found in source code, and the myriad variations in compilers, versions, and optimization levels (Balakrishnan & Reps, 2010). The term compiler provenance refers to information about the compiler family, compiler version, optimization level, and compiler-related functions. Extracting compiler provenance is important in digital forensics, as it provides crucial information about the process by which a malware binary is produced.

Few studies have been conducted on extracting compiler provenance (Jacobson et al., 2011; Rosenblum et al., 2010, 2011a). The existing body of work can be considered as a series of pioneering efforts, beginning with labeling functions in stripped binaries (Jacobson et al., 2011), followed by identifying the source compiler of program binaries (Rosenblum et al., 2010), and culminating in recovering the toolchain provenance of binary code (Rosenblum et al., 2011a). Inspired by such efforts, we provide improvements by addressing the following issues: 1) the features used in the aforementioned works follow generic templates, which may not always provide meaningful information about compilers; 2) large data sets are

☆ This research is the result of a fruitful collaboration between the Computer Security Laboratory (CSL) of Concordia University, Defence Research and Development Canada (DRDC) Valcartier and Google under a DND/NSERC Research Partnership Program.

* Corresponding author.
E-mail address: s_alraba@encs.concordia.ca (S. Alrbaee).

http://dx.doi.org/10.1016/j.diin.2015.05.015
1742-2876/© 2015 The Authors. Published by Elsevier Ltd on behalf of DFRWS. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

usually necessary for the training phase; and 3) significant computational power and time are often needed to rank the features. More recently, the compiler was identified in Toderici and Stamp (2013) based on a hidden Markov model that is intended for malware detection. However, it does not extract information regarding the compiler version or optimization level. Furthermore, there are certain tools that are capable of identifying compilers (e.g., IDA Pro (IDA Pro multi-processor disassembler and debugger,), PEiD (The PEiD tool,), and RDG (The RDG Packer Detector)); however, these tools typically suffer from various drawbacks. For instance, most of these tools apply an exact matching algorithm, which may fail when even a slight difference between signatures is present.

Motivation

Our motivation is twofold:

- Existing techniques (Rosenblum et al., 2010, 2011a) rely on generic feature templates and feature ranking. This usually leads to large amounts of irrelevant features and consequently results in prohibitive time and computational complexity. Moreover, top-ranked features are not necessarily related to the structure or semantics of compiler-related functions. In contrast, *BinComp* uses syntactic, semantic, and structural features to extract compiler provenance in a more efficient manner.
- Existing techniques (Rosenblum et al., 2010, 2011a) do not explicitly label compiler-related functions, which provides vital help to various tasks in binary analysis, such as authorship attribution (Alrabaee et al., 2014; Rosenblum et al., 2011b), function recognition (Alrabaee et al., 2015; Ruttenberg et al., 2014; Stojanovic et al., 2014), and clone detection (Edler et al., 2014; Farhadi et al., 2014), in which filtering out compiler-related functions is a critical pre-processing step for reducing false positives. In contrast, *BinComp* allows labeling compiler-related functions.

Our approach is based on the hypothesis that compiler-helper functions are preserved throughout the compilation process. These functions can be used to identify the source compiler of the binary. The stratified architecture of *BinComp* consists of three layers. The first layer extracts syntax features, namely, Compiler Transformation Profile (CTP), and compiler tags (CT). The second layer extracts Compiler Function (CF) features, which are represented as symbolic and numerical feature vectors used to label the compiler-related functions. This layer provides a list of compiler-helper functions. The third layer extracts semantic features, namely, the Compiler Constructor Terminator (CCT) graph and the Annotated Control Flow Graph (ACFG). This layer identifies the version and the optimization level from the helper functions. The architecture of *BinComp* is illustrated in Fig. 1. Each layer relies on a different detection technique: Layer 1 employs signatures to perform exact matching; Layer 2 uses the distances between the numerical vectors to label compiler-related functions; and Layer 3 extracts semantic graphs that are represented as feature vectors, after which hashing is applied to control the granularity level over the program components.

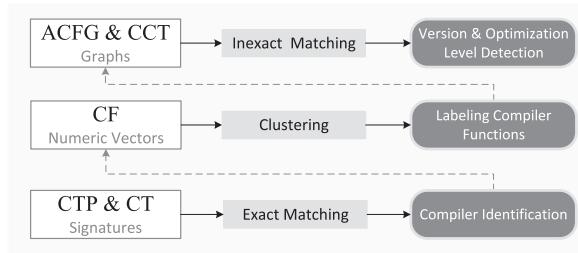


Fig. 1. BinComp architecture.

The novelty of the proposed approach is that it combines prior knowledge of compiler code transformations with an incremental learning component. This serves to adjust source compiler features as well as to update them when compiler-related functions undergo changes due to new versions or updates. Our experimental results show that *BinComp* identifies compiler families effectively, and that the approach is efficient in terms of both time and computational resources. This makes *BinComp* a practical approach for binary analysis.

Contribution

Our paper makes following contributions:

- *BinComp* introduces a stratified approach which simultaneously achieves several goals, including compiler identification, compiler-related function labeling, compiler version detection, and optimization level recognition.
- The signatures used in most tools depend on meta-data or other details of program headers. This can be problematic as such information might be unavailable in stripped binaries or might be easily altered. However, the signature used in *BinComp* relies on the characteristics of the binary, which are available even in stripped form.
- We evaluate *BinComp* on a large set of real-world binaries across several compiler families, versions, and optimization levels. Our results show that compiler provenance can be extracted with high accuracy; *BinComp* identifies the compiler with an accuracy that is above 0.90, the version with an average accuracy of 0.86 and the optimization level with accuracy of over 0.90.

The remainder of this paper is organized as follows: Section Motivating Example provides a motivating example. Section Evaluation of the ECP Approach introduces the re-evaluation of the technique used by Rosenblum et al. Section BinComp Approach details the main methodology. Section Evaluations evaluates the proposed approach and provides comparisons against existing work. Section Conclusion presents some concluding remarks on this work together with a discussion of future research.

Motivating example

Consider the simple program shown in Listing 1. Compilation of this program with Visual Studio 2010 in release mode results in 34 functions. These include one user function (main), five library functions (e.g., iostream), and 28 compiler-related functions. Subsequently, we compile the same example with Visual Studio 2012. Upon disassembling the binary, we find 73 functions: three user functions, 18 library functions, and 52 compiler-related functions. By checking the compiler-related functions, we find that 25 functions remain the same. In addition, we conduct the same simple experiment with a slightly complex program, and we find the same set of 25 compiler-related functions. To obtain the exact list of these functions, we intersect the sets of disassembled compiled binaries. However, we cannot rely merely on the names of the functions, as in most cases functions are given generic names (i.e., sub-xxx) by the disassembler. Therefore, after listing the helper functions (common compiler-related functions), two feature vectors are created for each function, as detailed in *BinComp* Approach Section. These feature vectors are used to ascribe a profile to each compiler.

Listing 1: Do Nothing C++ Program

```
#include <stdio.h>
 int main(){
     return 0;
 }
```

In addition, we extract the compiler effect (compiler transformation profile) from the user functions, since different compilers may generate distinct binary representations for analogous source code. Several parameters, such as compiler families, mapping profiles of operations to instructions, conditional/loop structures, optimization options, etc., govern the way compilers transform code from high-level source to low-level machine code. Listing 2 shows a simple C program consisting of a single function call to printf. Listings 3, 4, and 5 display the compiled versions with MSVC (Windows), GCC (Linux), and Clang (OS X), respectively. It is evident that each compiler generates a different sequence of instructions for the source program. Given a list of assembly functions (similar to listings 3, 4, and 5) from a target disassembly, we will show how to identify the most likely compiler with which the binary is built.

Listing 2: A Simple C Program

```
#include <stdio.h>
 int main(){
     printf("Minimal␣Program.\n");
     return 0x1234;
 }
```

Listing 3: Compilation with MSVC on Windows

```
main
    push    ebp
    mov     ebp, esp
    push    offset aMinimalProg
    call    ds:__imp__printf
    add     esp, 4
    mov     eax, 1234h
    pop     ebp
    retn
```

Listing 4: Compilation with GCC on Linux

```
main
    lea     0x4(esp),ecx
    and     $0xfffffff0,esp
    push   -0x4(ecx)
    push    ebp
    mov     esp,ebp
    push    ecx
    sub     $0x4,esp
    mov    $0x8048460,(esp)
    call    80482d4 <puts@plt>
    mov     $0x1234,eax
    add     $0x4,esp
    pop     ecx
    pop     ebp
    lea     -0x4(ecx),esp
    ret nop
```

Listing 5: Compilation with XCode on OS X

```
main
    push   ebp


 mov    esp,ebp
 push   ebx
 sub    $0x14,esp
 call   0x00001fd6
 pop    ebx
 lea    0x0000001a(ebx),eax
 mov    eax,(esp)
 call   0x00003005
 mov    $0x00001234,eax
 add    $0x14,esp
 pop    ebx
 leave ret
```

Evaluation of the *ECP* approach

ECP models compiler identification as a structured classification problem and labels each byte of the binaries with the information whether it is compiled with one or two compilers (statically linked code). The authors use wildcards idiom features, which are defined as short sequences of instructions that neglect details such as literal arguments and memory offsets. After extracting the features from a large number of binaries, redundant features can be found, which are related to the user functions or the architecture. Therefore, the authors consider top-ranked

features based on the results of mutual information computation between the features and compiler classes. They train the linear-chain model's parameters to assign high probabilities to correct compiler classes. The proposed technique is performed on three sets of binaries containing code from three compilers: GNU C Compiler (GCC), Intel C Compiler (ICC), and Visual Studio (VS). The data set is collected for GCC and VS on Linux and Microsoft Windows workstations, respectively. In addition, they have compiled various open-source software packages with the ICC compiler.

Most of these data sets are not publicly available. We have collected different source files from various years of the Google Code Jam (referred to as the G data set) (The Google Code Jam,) (2008–2014). We have also gathered several source code samples from our university (referred to as the U data set). Our experimental corpus comprises three sets of binaries: GNU C Compiler (GCC), Intel C Compiler (ICC), and Visual Studio compiler (VS). We disassemble each program binary using IDA Pro (IDA Pro multi-processor disassembler and debugger,) disassembler. Furthermore, we test their approach with an additional compiler, Clang (XCode).

ECP evaluation results

We generate our data set after considering all applicable combinations of compilers, various data set combinations, and different optimization levels. In what follows, we discuss our results based on a specific process of validation. First, we split the training data into ten sets, reserving one set as a testing set. We then train a classifier on the remaining sets and evaluate its accuracy on the testing set. In addition, we perform various experiments, as shown in Fig. 2: (i) changing the number of files; (ii) modifying the threshold value of the ranked features; and (iii) mixing various percentages of data sets to observe the effect of data set diversity. Finally, we measure the time efficiency by showing the relationship between the number of features and the total elapsed time.

The *ECP* approach can attain relatively high accuracy, as shown in Fig. 2 (a), where the accuracy for the G data set is between 0.89 and 0.98. This is higher than the accuracy obtained using the U data set (min = 0.82, max = 0.95). By analyzing the source code, we find out that the user contribution in the U data set is greater than that of the G data set. The U data set is also more complex than the G data set, as it consists of classes, methods, etc. In addition, we studied how changing the threshold value for the number of top-ranked features affects accuracy. We find out that the accuracy of their method may depend on the choice of the threshold value, as shown in Fig. 2 (b). For instance, the best threshold values for GCC and XCode compilers are 16,000 and 18,000, respectively. In practice, finding the best threshold for a given compiler and data set combination may be a task that is both time-consuming and prone to errors.

Finally, we perform an experiment to study the effects of data set diversity on accuracy, as illustrated in Fig. 2 (c). We consider different percentages of mixed data sets (e.g., 100% G data set, and 20% G - 80% U data sets). As shown, the accuracy decreases when the diversity of the data set increases, especially when the percentage of university projects increases, since the user contribution in the U data set is higher than that of the G data set. We measure the time efficiency of the *ECP* approach by calculating the total required time for feature extraction and feature ranking. Fig. 2 (d) shows the total time versus the number of features for different experiments.

Discussion and limitation

The *ECP* approach represents pioneering effort on compiler identification and may attain relatively high accuracy. This approach can also identify the compilers of binaries containing a mixture of code from multiple compilers, such as statically linked library code. However, a few limitations could be observed. First, as the number of features increases, the running time may increase even more rapidly. Second, the features of specific compilers may only become apparent after examining a large number of binaries. Third, the accuracy may depend on both the data set and the choice of threshold.

The *BinComp* approach

We present the different layers of our solution.

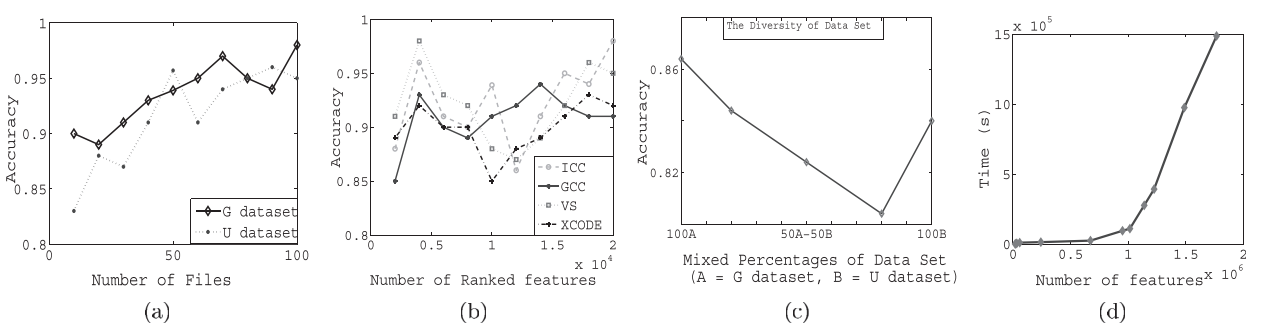


Fig. 2. Re-evaluation results: (a) Relation between accuracy and number of files (b) Relation between accuracy and threshold of ranked features (c) Diversity in datasets; (d) Time versus number of features.

First layer: Compiler identification

The purpose of in this layer is to derive unique signatures for compilers by collecting a set of known source code and observing the compiled outputs. We tailor *BinComp* to C-family compilers due to their popularity and widespread adoption, especially in the development of malicious programs (Lindorfer et al., 2012). The steps used in this layer are displayed in Fig. 3. As can be seen, we use the artifacts of each compilation step to generate specific types of features.

1. **Pre-processing**: The input of this step is the original source code (SRC), and the output is a pre-processed source file that includes the contents of header files imported into the source code. The symbolic constants are replaced with their values as a result of this process. To support the signature generation process, we extract the values of constants and the list of source-level functions along with their prototypes.
2. **Compilation**: The expanded source file generated in Step 1 is fed into the compilation step, which results in a platform-dependent assembler file (IAS). In support of the signature generation process, the list of assembly functions is extracted.
3. **Assembly**: We use the disassembled version of the object file (OBJ) —the output of this step— to extract the relevant features. To support the signature generation process, we match each assembly-level function with the corresponding source-level function obtained from Step 1. At this point, we are able to establish a mapping between assembly code profiles (sequences) and the source code. This mapping provides us with clues about the compiler code transformation process and syntactic styles (compiler transformation profile, as described later in *BinComp* Approach Section).
4. **Linking**: The linking step deals with the integration of the library object code (LIB) with the user object code (OBJ) built in Step 3 to produce the final binary file (BIN).
5. **Disassembly**: We utilize the disassembled version of the binary file to extract compiler tags (compiler tags are described later in *BinComp* Approach Section).

First layer: feature description

Compiler transformation profile (CTP)

CTP is a syntactic feature that highlights how certain syntactic source-level data and control structures (such as conditional/unconditional jumps, loops, arrays, structs, etc.) are reflected in the assembly output of compilers. The compiler transformation profile captures the code transformation process and maps the chosen source-level statements to a sequence of assembly-level instructions in order to profile the code transformation for each compiler. For instance, the assembly code translation of a simple if/else construct involves a comparison instruction followed by a conditional jump at the assembly level, which can be stated as if/else: cmp or test, then jcc, in binaries compiled by VS.

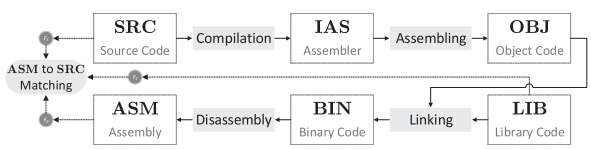


Fig. 3. Compilation steps.

Compiler tags (CT)

Compilers may embed in output binaries certain tags in the form of strings or constants. For instance, GCC-compiled binaries carry a tag that survives even after the symbol stripping process. Similarly, programs compiled with VS have an XOR-encoded value in the file header section that indicates the compiler version. These values can therefore be considered as compiler indicators. We apply a file parsing mechanism (PE/COFF/ELF) to obtain such values.

First layer: Detection method

All extracted features in this layer are recorded as strings and used for exact matching. The features in this layer, including the compiler transformation profile and compiler tags, captures the behavior of the compiler to access primitive and composite data structures and code structures (e.g., control statements, loop structures, register utilization, and memory access). We combine them into a single signature entity Sig_{c_i} that is used to identify the compiler c_i.

Second layer: Compiler functions labeling

The aim in this layer is to label compiler-related functions. To support our goal, we select a pair of source programs P_1 and P_2 and direct them through the supervised compilation process using the compiler c_i. The outcome of this process includes two sets of disassembled functions F_{P_1} and F_{P_2} as well as their corresponding symbolic $\overrightarrow{V}_s$ and numeric $\overrightarrow{V}_n$ feature vectors for each function. We perform a similarity measurement based on the numerical values of feature vectors. We use a standard clustering algorithm (k-means) (Farnstrom et al.,) to group functions with similar attributes $(v_{n_1}, \ldots, v_{n_z})$ into k clusters $S = S_1, \ldots, S_k$ and $(k \leq |F_{P_1}| + |F_{P_2}|)$ to minimize the intra-cluster sum of squares. In the following equation, μ_i denotes the mean of the numerical feature values of each cluster.

$$\arg\min_{S} \sum_{i=1}^{k} \sum_{v_j \in S_i} \|v_j - \mu_i\|^2 \tag{1}$$

The parameter k may be estimated either by following standard practices in k-means clustering (Farnstrom et al.,), or by beginning with one cluster and continually dividing the clusters until the points assigned to each cluster have a Gaussian distribution as described in Hamerly and Elkan (2004). If function names are available in symbolic vectors (which is the case for debug binaries), we would have the option to intersect functions based on their symbolic names. In this case, we compute

$list(set(\overrightarrow{V}_{P_1}.names()) \cap set(\overrightarrow{V}_{P_2}.names()))$, where the $list()$ function returns the list of commonly disassembled functions and the $names()$ method returns the property name of each function.

Second layer: Feature description

Compiler functions (CF)

Compiler function features are extracted in two steps: i) intersect a set of disassembled files, and ii) extract symbolic and numerical vectors for each compiler function. Given a target binary B (with unknown compiler) and its set of disassembled functions $F_B = \{f_1, \ldots, f_m\}$, which are classified into two groups of functions (compiler-related functions and user-related functions), we extract symbolic feature vectors $V_s = \{V_{s_1}, V_{s_2}, \ldots, V_{s_z}\}$ and numerical feature vectors $V_n = \{V_{n_1}, V_{n_2}, \ldots, V_{n_z}\}$ from each disassembled compiler/linker-related function $f_i \in F_B$. We group the instructions according to the instruction categories as shown in Table 1. The numerical vectors are generated using a function fingerprinting technique (Rahimian et al., 2013), which encodes and quantifies the syntactic and structural features of binary functions. Table 2 shows an example of numerical vectors.

Second layer: Detection method

We introduce a function fingerprinting approach for the generation and detection of abstract representations of assembly functions. We consider a program P to be comprised of a set of functions $F = \{f_1, \ldots, f_m\}$. A feature extraction function $X : F \rightarrow S$ maps each function $f_i \in F$ to a set of features $s_i \in S$, where F is the set of compiler/linker-related functions and S is the set of all possible features:

$$s_i = X(f_i) \tag{2}$$

We define $G : S \rightarrow T$ as a fingerprint computation function that takes the set of features to generate the fingerprints. Each fingerprint $t_i \in T$ represents the encoded characteristics of function f_i:

$$t_i = G(s_i) = G(X(f_i)) \tag{3}$$

At a different level of abstraction, this function is similar to a hash function since it generates a fixed length output for an arbitrary length assembly function and compresses the information. However, this function computes the output based on a normalized form of assembly instructions. This function can be interpreted as a semantic hash function, which builds a bit vector of length m from the subset of features S that appear in f_i. The mechanics of the hash function can be balanced according to the required level of strength (number of feature bits) and the size of the feature space.

$$t_i \in domain(S) \rightarrow \{0,1\}^m \tag{4}$$

For function fingerprinting, we calculate the similarity between pairs of functions. It is also necessary to determine the similarity between a target function and a group of functions (e.g., incremental clustering). The similarity function M assigns a score to a pair of candidate fingerprints using a similarity metric. The similarity score can be computed at the level of features (i.e., $M' : F \times F \rightarrow \mathbb{R}^+$) or at the level of fingerprints (i.e., $M : T \times T \rightarrow \mathbb{R}^+$). There are various metrics for the computation of vector similarity scores (e.g., Jaccard similarity). Let t_i and t_j be two fingerprint vectors generated from the candidate function pairs (f_i, f_j), respectively. The Jaccard similarity can be calculated using the following equation (Gascon et al., 2013):

$$d_J(t_i, t_j) = \frac{S(t_i \wedge t_j)}{S(t_i \vee t_j)} \tag{5}$$

where the $S()$ function counts the number of ones in the fingerprint bit vector.

Third layer: Version and optimization recognition

The aim in this layer is to detect the version and the optimization level of the binary code. During the training phase, we observe that two features are slightly distinct: the annotated control flow graph (ACFG) and the compiler constructor terminator (CCT) graph. In such cases, we use neighborhood hashing (Gascon et al., 2013) to provide a level of granularity over the program components and features used that is more refined than graph encoding.

Third layer: Feature description

Annotated control flow graph (ACFG)

We present a new scheme, the Annotated Control Flow Graph (ACFG), to efficiently detect the version and optimization levels. It is an abstracted version of the CFG, which generalizes specific types of CFG features according to multiple criteria. We build an annotated control flow graph

Table 1
Instruction Categories.

Feature	Description	Feature	Description
DTR	Data Transfer	INA	Indirect Near Address
INO	Input/Output	DTO	Data Transfer Object
FLT	Float Point	FLG	Flag Manipulation
REG	Registers	LGC	Logical Instructions
MEM	Memory	CTL	Control Instructions
IMM	Immediate Value	IFA	Indirect Far Address
INT	Interrupt/System	ATH	Arithmetic Instructions

Table 2
Profiling compiler functions based on numerical vectors.

COMP	OPL	Symbolic function ID	DTR, DTO, FLG, ATH, LGC, CTL, INO, INT, FLT, REG, MEM, IMM, IFA, INA
VS	OP2	@__security_check_cookie@4	0, 0, 0, 0, 0, 4, 0, 0, 0, 001, 001, 000, 000, 002
VS	OP2	___tmainCRTStartup	3, 0, 1, 2, 1, 6, 0, 0, 0, 077, 028, 017, 000, 025
GCC	OP0	___mingw_CRTStartup	3, 1, 1, 3, 6, 7, 0, 0, 0, 216, 015, 068, 000, 071
GCC	OP0	___gcc_register_frame	3, 0, 1, 1, 0, 3, 0, 0, 0, 029, 001, 014, 000, 009

for each disassembled compiler-related function according to the type of operations that take place in each basic block. Each node in this graph represents a basic block, and each edge connects basic block B_1 to basic block B_2 if basic block B_2 can directly follow basic block B_1 in execution. The method for extracting an ACFG can be summarized in the following steps: First, we consider a CFG as an input and compute the frequency of opcodes across instruction groups. Second, each assembly instruction can be categorized according to the mnemonic groups and types of operands. As one category, the x86 instructions can be classified into six groups, as shown in Table 3. Third, we treat the ACFG as a structural feature and complement it with subgraph encoding values.

Definition 1. *An Annotated Control Flow Graph* $ACFG = (N,E,\zeta,\gamma)$ *is an attributed graph where N is a set of nodes each representing a basic block,* $E \subseteq (N \times N)$ *is a set of edges which represents a control flow statement,* γ *is a function that groups the instructions, and* ζ *is a function which colors nodes according to the type of operations that take place.*

ACFG construction

We use one compiler function (e.g., setprecision) to illustrate the steps involved in translating CFG to ACFG, as shown in Fig 4. The first part on the left side represents the standard control flow graph. As illustrated in the middle part, we convert each basic block in accordance with Table 3. Finally, function ζ converts these values to a specific color.

Consider the last basic block in Fig. 4 which contains the two instructions pop esi and retn. Function γ groups the two operators to STK and MSC, and the esi operand to REG. To color the nodes, ζ intersects the opcode and operands according to Table 3. For instance, DTR & STK = 0 & 1 = 0, while MSC = 1. Each basic block ends up with a six-bit value (e.g. 000001) which will be represented in a new graph by a decimal value equivalent to its binary. The ACFG captures the control flow semantics of a program, which contains more information and hence provides more accuracy than a CFG. An ACFG can convert equivalent compiler functions to a similar ACFG because the compiler functions are semantically similar. In contrast, a CFG deals only with the structure of the function; since different functions may have the same structure, this fact makes compiler function identification more difficult.

Compiler constructor terminator (CCT)

One important application of call graphs and control flow graphs is in the recognition of compiler-related call sequences. Each compiler exhibits a relatively unique signature for initialization, startup, and termination code. In both the initialization and termination part of the program, there are compiler functions that call other functions or are called by other functions. The order of these functions, number of nodes, flow, and other characteristics are some of the signatures that we use for the compiler constructor terminator (CCT) feature.

During the execution of the system, the call graph of the application is traversed until it reaches the main function. Every sequence of function calls taking place before and after the main function (initialization and terminators) is recorded in the compiler constructor terminator (CCT) of each compiler. Fig. 5 shows an example of a GCC compiler constructor terminator.

Table 3
Patterns for annotation.

Feature	Description	Example
DTR & STK	Data Transfer and Stack	push, mov, etc
ATH & LGC	Arithmetic and Logical	add, xor, etc
CAL & TST	Call and Test	call, cmp, etc
REG & MEM	Register and Memory	esi, [esi+4]
REG & CONST	Register and Constant	esi, 30
MSC	Milestones	MEM and Const

Third layer: Detection method

In certain applications, it may be necessary to match subsets of multiple graphs. For instance, we consider the initialization part of a call graph as a startup signature for compiler identification (compiler constructor terminator profile). In such cases, neighborhood hashing provides a more refined level of granularity compared to graph encoding. In order to fingerprint adjacent nodes, a neighbor hash graph kernel (NHGK) can be applied to subsets of the call graph (Gascon et al., 2013). The function *G* maps the set of annotated graphs *AG* to a bit vector of length *m*.

$$G : AG \rightarrow \{0,1\}^m \tag{6}$$

The neighborhood hash value h for a function f_i and its set of neighbor functions N_{f_i} can be computed using the following formula [8]:

$$h(f_i) = shr_1(G(f_i)) \oplus \left(\oplus_{f_j \in N_{f_i}} G(f_j) \right) \tag{7}$$

where shr_1 denotes a one-bit shift right operation and $\oplus$ indicates the XOR function.

Evaluation

Data set

Gathering a data corpus for the evaluation of compiler provenance extraction is challenging. For example, despite the fact that collecting code from open-source projects may be attractive, the source files usually have numerous dependencies which complicates the compilation process. Nonetheless, we choose four free open-source projects and use them to test *BinComp*. In addition, we have gathered programs written for the Google Code Jam (The Google Code Jam,) as well as university projects from a programming course at our university. All of our data sets are publicly accessible in (The data set,).

We generate the binaries that make up our data set by compiling the source code with all applicable combinations of compiler versions and optimization levels (O0 and O2), as shown in Table 4. Our data set consists of 1177 files, 232 of which belong to the Google Code Jam data set, 933 of

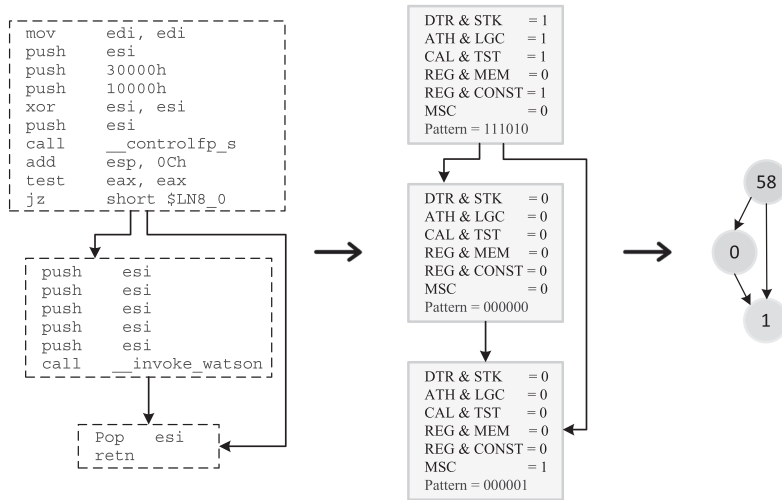


Fig. 4. ACFG construction.

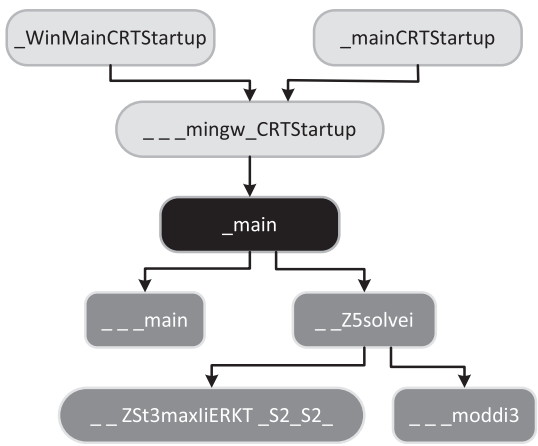


Fig. 5. Sample of GCC compiler constructor terminator.

which belong to the Students Code Projects, and 12 of which belong to the Open Source Projects.

Evaluation results

We evaluate our compiler provenance approach using the aforementioned data sets. We split the training data into ten sets, reserving one set as a testing set, and using nine sets as training sets to evaluate our approach; we repeat this process numerous times. To evaluation our appraoch, we use *precision* (P) and *recall* (R) as follows.

$$P = \frac{TP}{TP + FP} \tag{8}$$

$$R = \frac{TP}{TP + FN} \tag{9}$$

Since our application domain is much more sensitive to false positives than false negatives, we use the F-measure as follows.

$$F_{0.5} = 1.25 \cdot \frac{PR}{0.25P + R} \tag{10}$$

Our results of $F_{0.5}$ measure is summarized in Table 5.

As depicted separately in Fig. 6(a–d), *BinComp* can detect the VS compiler with an average accuracy of 0.97, while *ECP* average accuracy is 0.93. The main explanation for this difference lies in the type of features used in the technique; *BinComp* uses different kind of features (syntactical, structural, and semantical), whereas *ECP* uses only syntactical features. In addition, we record the overall run time and compare *BinComp* and *ECP* approaches in terms of time complexity, as shown in Fig. 7.

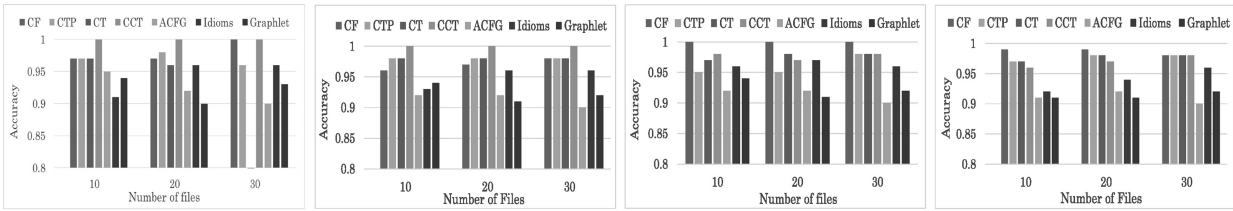
We test *BinComp* using different variations of compiler versions and optimization levels, as shown in Table 6 and Table 7. Table 6 indicates that identifying the version of

Table 4
Data set compilation settings.

Compiler	Version	Optimization
GCC	3.4	O0
	4.4	O2
ICC	10	O0
	11	O2
VS	2010	O0
	2012	O2
XCODE	5.1	O0
	6.1	O2s

Table 5
$F_{0.5}$ results.

Feature	$F_{0.5}$(Data set = 500 files)	$F_{0.5}$(Data set = 1000 files)
Idioms	0.789	0.812
Graphlet	0.602	0.62
CF	0.72	0.745
CTP	0.694	0.708
CCT	0.807	0.877
CT	0.689	0.70
ACFG	0.634	0.671



(a) VS (b) ICC (c) GCC (d) CLang

Fig. 6. The accuracy against different compilers.

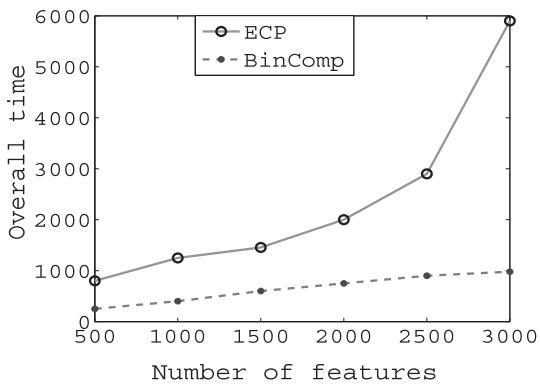


Fig. 7. Time efficiency.

compiler is significantly more difficult than recognizing the compiler and optimization levels. For instance, the accuracy to identify the version of XCode compiler is below 0.80. In addition, we observe that the features of VS and XCode compilers are slightly different when we change either the version or the optimization level, which makes the detection process more challenging. However, the GCC and ICC compilers produce more various code amongst versions compared to the VS and XCode compilers.

We find that up to 75% of the functions in our data set are identical when generated by Visual Studio 2010 or 2012 with the same optimization level. In other words, the code generator in Visual Studio has remained relatively stable between these versions, which offers an explanation for the low accuracy for VS version detection. On the other hand, we found that up to 85% of the functions of XCode are identical among two versions. However, we have observed changes in our proposed features for the GCC and ICC compilers which allow us to detect the version and optimization level by measuring the differences in the features.

A comparison between *BinComp*, *ECP*, and IDA Pro based on four open source projects named SQLite, libpng, zlib, and OpenSSL are illustrated in Fig. 8. We compile SQLite and zlib with VS 2010 (O2), and the compiler is successfully identified by these three approaches; whereas libpng and OpenSSL are compiled with GCC (O2), which IDA Pro is not able to identify the source compiler of these binaries. *BinComp* and *ECP* provide more accurate results in identifying the VS compiler, while IDA cannot identify the GCC compiler.

Table 6
Accuracy for variations of compiler versions.

Compiler	Version	Accuracy
GCC	3.4.x	86%
	4.4.x	89%
ICC	10.x	83%
	11.x	90%
VS	2010	70%
	2012	71%
XCode	5.x	78%
	6.1	74%

Table 7
Accuracy for variations of compiler optimization levels.

Compiler	Optimization	Average Accuracy
GCC	O0, O2	91%
ICC	O0, O2	89%
VS	O0, O2	95%

Discussion and limitations

We evaluate compiler provenance recovery on different data sets across several compiler families, versions, and optimization levels. We check the applicability of the features used in *BinComp* and the features used in the existing techniques (e.g., idioms, Graphlet) to identify the compiler, version, and optimization level based on our observations during the experiments. For instance, CT has the same pattern in the binaries that are compiled with the same compiler. Furthermore, we formulate a template called compiler transformation profile (CTP) that maps the instructions underlying a binary according to the most likely compiler family by which it was generated. We use such features as a unique signature (fingerprint) to identify a

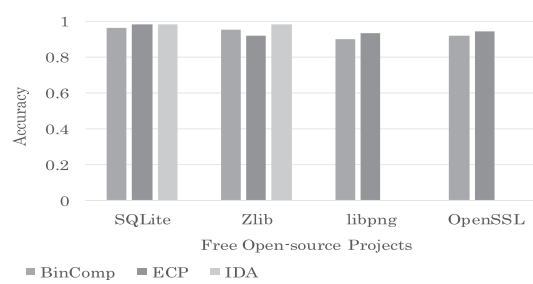


Fig. 8. Comparison the accuracy of BinComp, ECP, and IDA Pro on free open-source projects.

Table 8
Feature abstraction levels: instruction-level (inst), control flow, and external library (ext) interaction properties level.

Features	Code property		
	Inst.	Control flow	Ext.
Idioms	*		
Graphlet	*	*	*
CF	*		*
CTP	*		
CCT		*	*
CT	*		
ACFG	*	*	*

binary source compiler. We use other features (e.g., ACFG, and CCT) to identify the version and optimization level. The variations in compiler provenance lead to variations in the structure of the binary code. We introduce features that capture detailed provenance at function-level granularity, allowing us to recover the optimization levels used to produce binary code. We design ACFG and compiler constructor terminator (CCT) features to explicitly capture such changes and identify the version.

BinComp uses different features that cover various aspects of the binary. Each feature captures one or more of the code instruction-level (inst), control flow, or external library (ext) interaction properties, as shown in Table 8. The features used in *BinComp* cover all aspects of compiler transformations and behaviors.

BinComp still suffers from some limitations. For instance, like most existing methods, *BinComp* works under the assumption that the binary code is already de-obfuscated. In practice, the de-obfuscation of malware can be very demanding. How to extract compiler provenance directly over obfuscated code is an important but very challenging future avenue of research. *BinComp* also assumes the binary code is not stripped. In addition, for this work, only Intel x86/x86-64 architecture is considered, and we evaluate *BinComp* only over compiled C++ programs.

Conclusion

We have presented a technique called *BinComp* for accurately and automatically recovering the compiler provenance of program binaries using syntactical, semantical, and structural features to capture the compiler behavior. *BinComp* extracted specific features from program binaries, which enabled us to build representative and meaningful features to describe each compiler characteristics better. Our results show that compiler provenance can be extracted accurately. Moreover, the results indicate that the approach was efficient in terms of computational resources and time requirements, and could thus be considered as a practical approach for real-world binary analysis.

Acknowledgments

The authors thank the anonymous reviewers for their valuable comments. They are also thankful to Dr. Vassil Roussev for his constructive comments. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsoring organizations.

References

Alrabaee S, Saleem N, Preda S, Wang L, Debbabi M. OBA2: an onion approach to binary code authorship attribution. Digit Investig 2014: S94–103. Elsevier.

Alrabaee S, Shirani P, Wang L, Debbabi M. SIGMA: a semantic integrated graph matching approach for identifying reused functions in binary code. Digit Investig 2015;12:S61–71.

Balakrishnan G, Reps T. Wysinwyx: what you see is not what you execute. ACM Trans Program Lang Syst (TOPLAS) 2010;32(6) [ACM].

Edler K, Franke T, Bhandarkar P, Dasgupta A. Exploiting function similarity for code size reduction. In: Proceedings of the 2014 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems; 2014. p. 85–94 [ACM].

Farhadi M, Fung B, Charland P, Debbabi M. BinClone: detecting code clones in malware. In: Software Security Reliability, 2014 Eighth International Conference on. IEEE; 2014. p. 78–87.

F. Farnstrom, J. Lewis, and C. Elkan, Scalability for clustering algorithms revisited, ACM SIGKDD Explor Newsl, Vol 21, 51–57.

Gascon H, Yamaguchi F, Arp D, Rieck K. Structural detection of android malware using embedded call graphs. In: Proceedings of the 2013 ACM workshop on Artificial intelligence and security; 2013. p. 45–54 [ACM].

Hamerly G, Elkan C. Learning the k in A > means. In: Advances in neural information processing systems16; 2004. p. 281.

IDA Pro multi-processor disassembler and debugger, Available from: https://www.hex-rays.com/products/ida/, [accessed 09.06.14].

Jacobson E, Rosenblum N, Miller B. Labeling library functions in stripped binaries. In: The 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools (SIGSOFT '11); 2011. p. 1–8 [ACM].

Lindorfer M, Di Federico A, Maggi F, Comparetti PM, Zanero S. Lines of malicious code: insights into the malicious software industry. In: Proceedings of the 28th Annual Computer Security Applications Conference; 2012, December. p. 349–58 [ACM].

Rahimian A, Charland P, Preda S, Debbabi M. RESource: a framework for online matching of assembly with open source code. In: Foundations and Practice of Security (FPS 2013). Springer Berlin Heidelberg; 2013. p. 211–26.

Rosenblum N, Miller B, Zhu X. Extracting compiler provenance from program binaries. In: The 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (SIGSOFT '10); 2010. p. 21–8. ACM.

Rosenblum N, Miller B, Zhu X. Recovering the toolchain provenance of binary code. In: The 2011 International Symposium on Software Testing and Analysis; 2011. p. 100–10 [ACM].

Rosenblum N, Zhu X, Miller B. Who wrote this code? Identifying the authors of program binaries. In: Computer security-ESORICS. Springer Berlin Heidelberg; 2011. p. 172–89.

Ruttenberg B, Miles C, Kellogg L, Notani V, Howard M, LeDoux C, et al. Identifying shared software components to support malware forensics. In: Detection of Intrusions and Malware, and Vulnerability Assessment. Springer International Publishing; 2014. p. 21–40.

Stojanovic S, Radivojevic Z, Cvetanovic M. Approach for estimating similarity between procedures in differently compiled binaries, information and software technology. Elseiver; 2014.

The data set. Available from: https://github.com/BinSigma/BinComp/tree/master/Dataset, [accessed 30.04.15].

The Google Code Jam. Available from: https://code.google.com/codejam, [accessed 27.10.14].

The PEiD tool. Available from: http://www.woodmann.com/collaborative/tools/index.php/PEiD, [accessed 14.08.14].

The RDG Packer Detector. Available from: http://www.woodmann.com/collaborative/tools/index.php/RDG_Packer_Detector, [accessed 14.08.14].

Toderici A, Stamp M. Chi-squared distance and metamorphic virus detection. J Comput Virol Hacking Tech 2013;9(0):1–14. Springer.