DFRWS

DIGITAL FORENSIC RESEARCH CONFERENCE

BinComp: A Stratified Approach to Compiler Provenance Attribution

By

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

Presented At

The Digital Forensic Research Conference

DFRWS 2015 USA Philadelphia, PA (Aug 9th - 13th)

BinComp: A Stratified Approach to Compiler Provenance Attribution

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

Agenda

- Introduction
- Use cases
- Motivating Example
- BinComp Approach
- Evaluation
- Comparison
- Discussion and Limitations

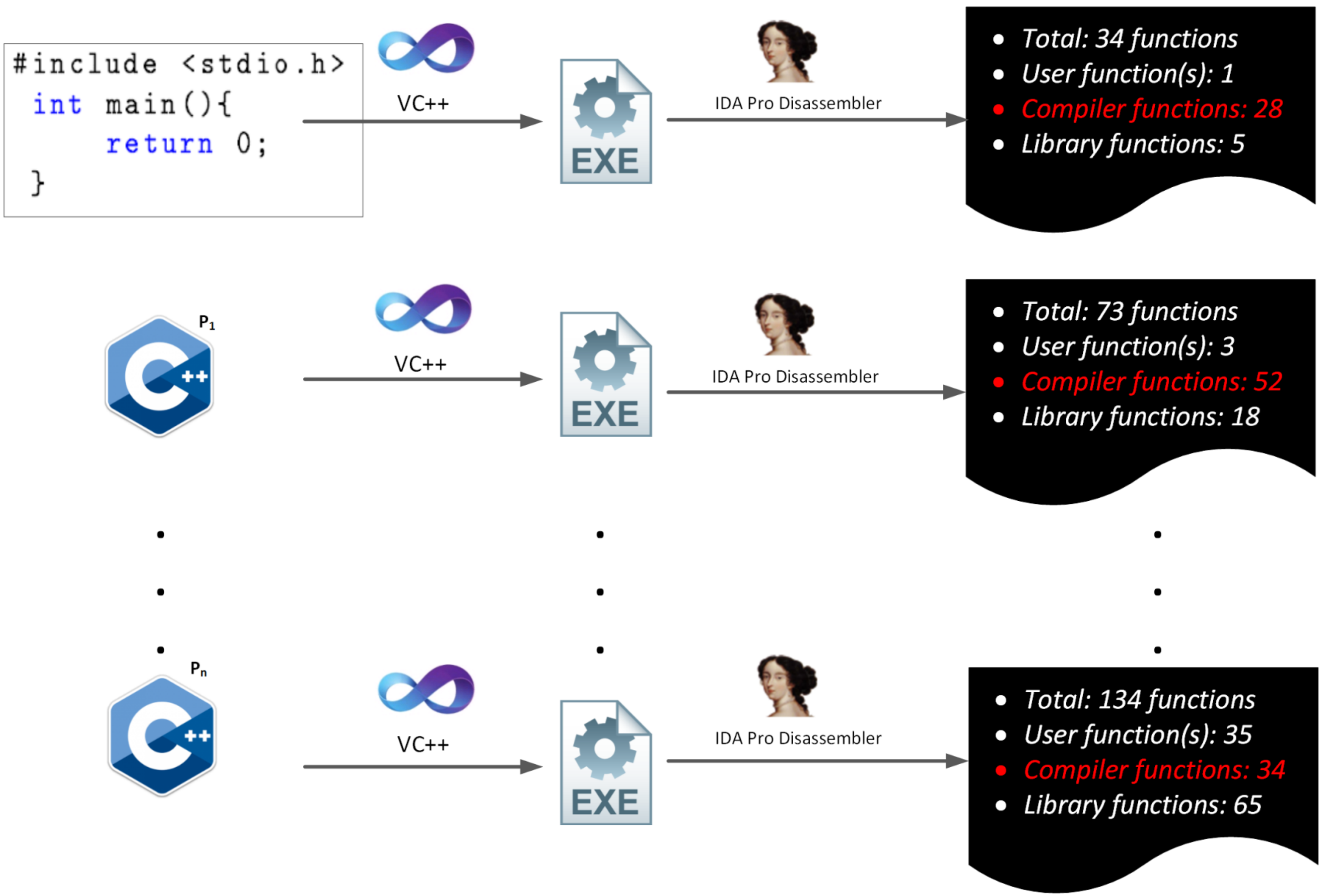
Introduction

- Compiler provenance attribution encompasses
  - Compiler family
  - Compiler version
  - Optimization level
  - Compiler-related functions
- Why compiler provenance attribution?
  - It is a necessary component at pre-processing stage for binary analysis
  - It is important in digital forensics
    - It provides information about the process by which a malware binary is produced

Use Cases

- Compiler identification can be useful for
  - Tool chain malware analysis in binaries
    - Clustering malware samples based on families (assuming each family is using a specific compiler/linker setting)
    - Reasoning about the evolution of malware binaries
  - Library identification, authorship attribution, clone detection, etc.

Motivating Example



Extract Compiler-related Functions

GCC

['___p__environ', '_strlen', '_glob_match', '_rewinddir',
'___mingwthr_run_key_dtors.part.0', '__cexit', '_strncpy', '_signal',
'_memcpy', '_GetCommandLineA@0', '__pei386_runtime_relocator',
'__errno', '___mingw_globfree', '_register_frame_ctor', '_readdir',
'__setmode', '_puts', '_SetUnhandledExceptionFilter@4', …

VC++

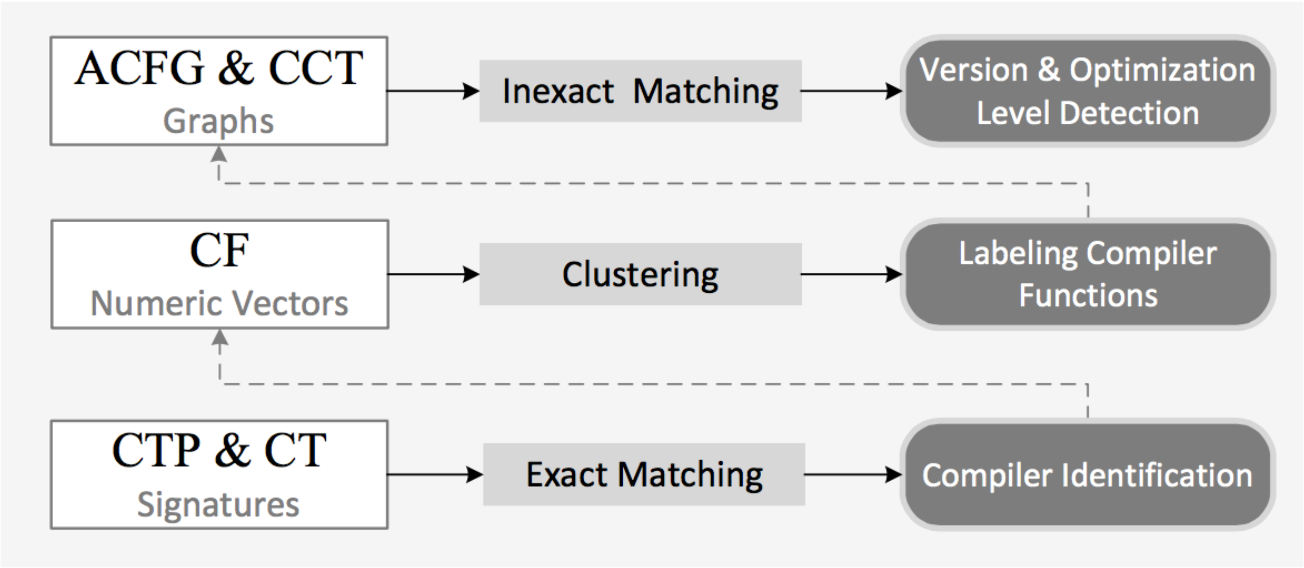
['___tmainCRTStartup', '__ValidateImageBase', '?
__CxxUnhandledExceptionFilter@@YGJPAU_EXCEPTION_POINTERS@@
@Z', '__dllonexit', '_controlfp_s', '_invoke_watson', '_amsg_exit', '?
terminate@@YAXXZ', '__setdefaultprecision', '__SEH_epilog4', '_lock',
'__onexit', 'start', '_pre_cpp_init', '_XcptFilter', '_init, …

ICC

['__cxa_atexit', '.strncat', 'getenv@@GLIBC_2.2.5',
'_ZNSt8ios_base4InitD1Ev', '__intel_cpu_features_init', '.setenv',
'_ZNSt8ios_base4InitC1Ev', '.fprintf', '.catopen',
'catopen@@GLIBC_2.2.5', '.term_proc', 'vsprintf@@GLIBC_2.2.5',
'_start', 'strlen', 'strncpy', 'strchr', …

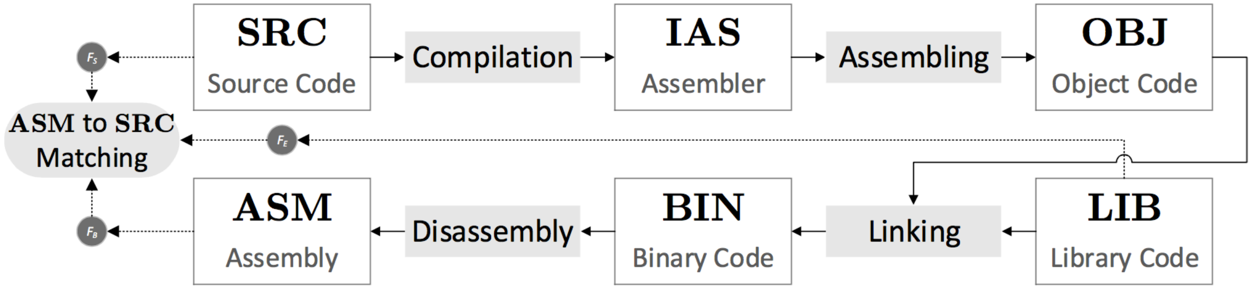
BinComp Approach

- A layered approach
  - Compiler identification (syntax features)
  - Labeling compiler functions (semantic features)
  - Version & optimization level detection (structural & semantic features)



1 st layer: Compiler Identification

- Collecting a set of known source code
- Observing the compiled outputs
- Extracting the features
  - CTP (Compiler Transformation Profile)
  - CT (Compiler Tags)
- Exact matching



1 st layer: Compiler Identification

- CTP (Compiler Transformation Profile)
  - How source-level data and control structures are reflected in the assembly output
    - For example, corresponding assembly code of `if/else` compiled with VS will be `cmp` or `test`, then `jcc`.
- CT (Compiler Tag)
  - Compilers may embed certain tags in the form of strings or constants
    - For example, GCC writes a *.comment* section that contains the GCC version string

2^{nd} layer: Compiler Function Labeling

- Signature generation
  - Extracting CF (Compiler Functions)
    - Numerical vectors such as number of instructions, type of registers, etc.
    - Symbolic vectors: such as function names, function prototypes, etc.
- Signature detection
  - Computing the similarity

n Function Name, **ne** Demangled Name, **i** Imported Function, **k** Call List (From), **ke** Demangled Calls, **ix** Number of Imported Functions, **kx** Number of Calls
c Constant, **s** String, **cx** Number of Constants, **sx** Number of Strings
p Function Prototype, **a** Function Argument, **r** Return Type, **g** Number of a, **b** Size of Arguments, **gx** RES. Number of g
m Number of Instructions, **l** Size of Local Variables, **f** Function Flags, **o** Code References (From), **z** Function Size, **mx** RES. Number of m, **ox** Number of o, **cc** Cyclomatic Complexity, **bx** Number of Basic Blocks
d Dictionary (Malware Tag), **t** API Tag, **dx** Number of d, **tx** Number of t
av Anti-VM, **reg** General Register, **mem** Memory Reference, **bix** [Base +Index], **bid** [Base+Index+Displacement], **imm** Immediate, **ifa** Immediate Far Address, **ina** Immediate Near Address, **fpp** FPP Register, **ctr** Control Register, **dbr** Debug Register, **trr** Trace Register
DTR Data Transfer, **DTO** Data Transfer Address Object, **FLG** Flag Manipulation, **DTC** Data Transfer Conversion, **ATH** Binary Arithmetic, **LGC** Logical, **CTL** Control Transfer, **INO** Input Output, **INT** Interrupt and System, **FLT** Floating, **MSC** Misc.

2nd layer: Compiler Function Labeling

- Example of CF features

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

Instruction Categories.

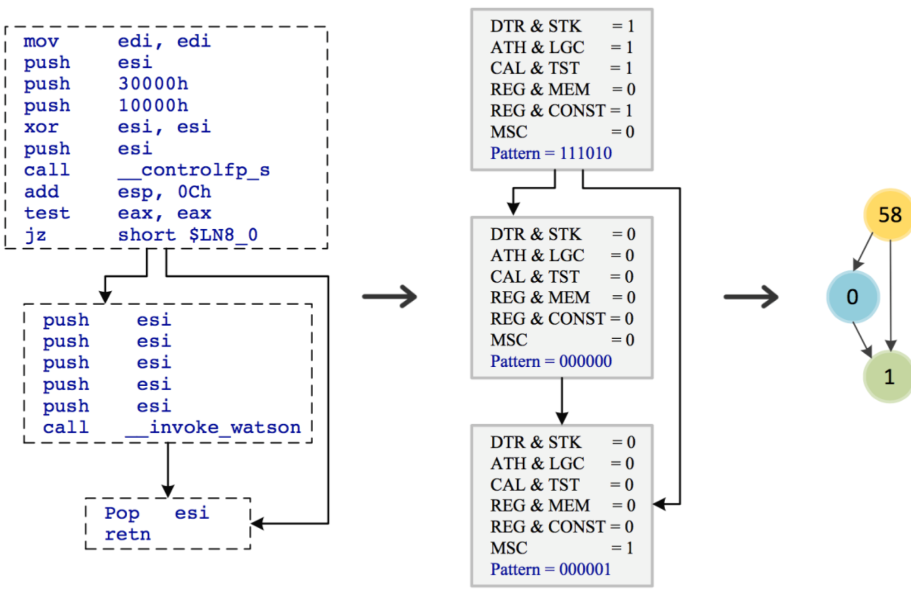
Feature	Description	Feature	Description
DTR	Data Transfer	INA	Indirect Near Address
INO	Input/Output	DTO	Data Transfer Object
FLT	Float Point	FLG	Flag Manipulation
REG	Registers	LGC	Logical Instructions
MEM	Memory	CTL	Control Instructions
IMM	Immediate Value	IFA	Indirect Far Address
INT	Interrupt/System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

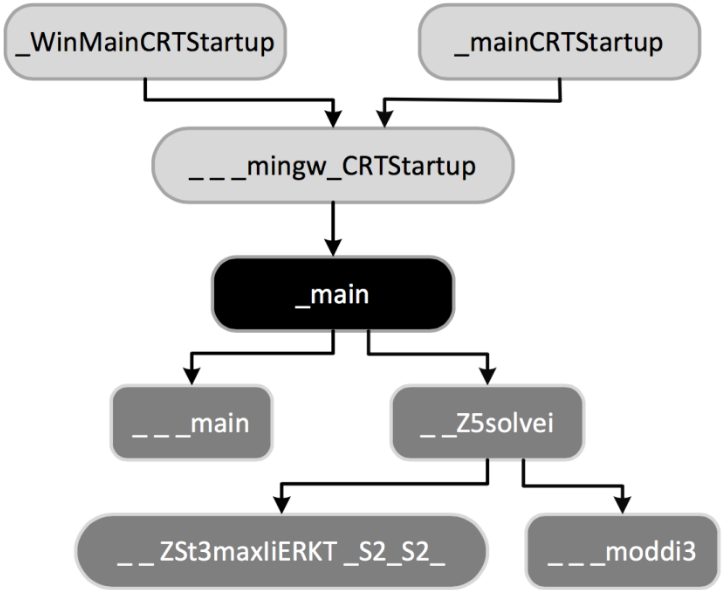
$$d_J(t_i, t_j) = \frac{S(t_i \wedge t_j)}{S(t_i \vee t_j)}$$

t_i, t_j: fingerprint vectors generated from the candidate function pairs

3rd layer: Version & Optimization

- Extracting the features
  - ACFG (Annotated Control Flow Graph)
  - CCT (Compiler Constructor Terminator)





3rd layer: Version & Optimization

- According to our experiments ACFG and CCT are the best features to detect the version and optimization
  - The different versions can affect the ACFG
  - The different optimization levels affect the CCT
    - For instance, the CCT for full optimized is a subset of CCT for the no-optimization code

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

Université Concordia University

Evaluation

- Dataset
  - Four free open-source projects (SQLite, zlib, libpng, and openSSL)
  - Google Code Jam (232 files)
  - Students Code Projects (993 files)

Evaluation

- Results

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%

Accuracy for variations of compiler optimization levels.

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

Comparison

	IDA Pro Disassembler	Rosenblum et al	BinComp
Features	Entry point signatures	Syntax (n-gram)	Syntax, Semantic, Structural
Detection Method	Signature based	Classification	Exact/Inexact matching
Compilers	VS, Delphi, Fortran	GCC, VS, ICC	VS, GCC, ICC, XCODE
Required data set	Small	Large	Small
Scalability	✓	✗	✓
Time efficient	✓	✗	✓
Identifying Ver.	✗	✗	✓
Identifying Opt.	✗	✗	✓

Discussion and Limitations

- BinComp requires few data set and it can be applied for any compiler
- BinComp is efficient in terms of time and scalability
- Limitations
  - The binary code is deobfuscated
  - Only Intel x86/x86-64 architecture is considered

Thank You!

Evaluation

- Accuracy
  - Precision (P)
  - Recall (R)

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

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

- Our application domain is much more sensitive to false positives than false negatives

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

Neighbor hash graph kernel (NHGK)

- Neighbor hash graph kernel (NHGK)
  - Condense the information contained in a neighborhood into a single hash value
  - Label each node in the function call graph
    - each function is characterized by its numerical and symbolic feature vectors
- our method strives to model the composition of functions
  - The neighborhood of a function must be taken into account.
  - We compute a neighborhood hash over all of its direct neighbors in the function call graph

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

  - shr_1: a one-bit shitt right operation
  - $\oplus$: a bit-wise XOR on the binary labels