AFRL-AFOSR-VA-TR-2020-0092

Towards Fundamental and Binary-Centric Techniques for Kernal Malware Defense

Bhavani Thuraisingham
UNIVERSITY OF TEXAS AT DALLAS

12/05/2019
Final Report

Air Force Research Laboratory
AF Office Of Scientific Research (AFOSR)/ RTA2
Arlington, Virginia 22203
Air Force Materiel Command

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
14-07-2020	Final Performance	01 Sep 2014 to 31 Aug 2019

4. TITLE AND SUBTITLE
Towards Fundamental and Binary-Centric Techniques for Kernal Malware Defense

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA9550-14-1-0119

5c. PROGRAM ELEMENT NUMBER
61102F

6. AUTHOR(S)
Bhavani Thuraisingham, Zhiqiang Lin

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
UNIVERSITY OF TEXAS AT DALLAS
800 W CAMPBELL RD
RICHARDSON, TX 75080 US

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AF Office of Scientific Research
875 N. Randolph St. Room 3112
Arlington, VA 22203

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/AFOSR RTA2

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
AFRL-AFOSR-VA-TR-2020-0092

12. DISTRIBUTION/AVAILABILITY STATEMENT
A DISTRIBUTION UNLIMITED: PB Public Release

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This project seeks to develop a set of fundamental and binary-centric techniques for kernel malware defense. Defeating kernel malware is challenging because kernel malware runs as the same privilege level as the OS kernels, and they can easily disable and fight against the security software at this layer. The unique difference compared to all the existing work is that we focus on the semantic and syntactic analysis of OS kernel binary code to discover the invariants between kernel code and data, from which to detect kernel intrusions, investigate damages, repair attacks, and enforce the preventions from hypervisor layer. During the past five years supporting period, a number of fundamental techniques have been developed from this project, and these include address-agnostic cross-kernel pointer integrity checks (FPCK), robust kernel object semantic inference (Argos), kernel tap points discovery (AutoTap), and superset disassembly (MultiVerse) and so on. These binary-centric techniques have enabled kernel invariant understanding, extraction, and enforcement (e.g., rewriting with the tap points) from virtual machine layer (a layer that cannot be disabled by kernel malware inside the virtual machines). In total, 25 peer-reviewed academic papers supported or partially supported by this project have been published, many of which appeared in top venues such as IEEE S&P, CCS, USENIX Security, NDSS, FSE, ICSE, ACSAC, and RAID.

15. SUBJECT TERMS
KERNAL MALEWARE, KERNAL DATA INVARIANT

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			NGUYEN, TRISTAN

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Unclassified	Unclassified	Unclassified	UU	**PAGES**	**19b. TELEPHONE NUMBER** *(Include area code)* 703-696-7796

Final Report
AFOSR Grant No. FA9550-14-1-0119

Towards Fundamental and Binary-Centric Techniques for Kernel Malware Defense

Nov 20th, 2019

1 Principal Investigator

PI: Dr. Bhavani Thuraisingham
Computer Science Department, EC31
The University of Texas at Dallas
Richardson, TX 75080-3021
Voice: (972) 883-4738
Email: bhavani.thuraisingham@utdallas.edu
Web: https://personal.utdallas.edu/~bhavani.thuraisingham/

Subcontract PI: Dr. Zhiqiang Lin
Computer Science and Engineering Department
The Ohio State University
Columbus, OH 43210-1277
Voice: (614) 292-0055
Email: zlin@cse.ohio-state.edu
Web: https://web.cse.ohio-state.edu/~lin.3021

2 Project Motivation and Summary

This project seeks to develop a set of fundamental and binary-centric techniques for *kernel malware defense*. Defeating kernel malware is challenging because kernel malware runs as the same privilege level as the OS kernels, and they can easily disable and fight against the security software at this layer. The unique difference compared to all the existing work is that we focus on the semantic and syntactic analysis of OS kernel binary code to discover the invariants between kernel code and data, from which to detect kernel intrusions, investigate damages, repair attacks, and enforce the preventions.

In the past five years, we have published over 25 academic papers (13 are directly related and supported by this project, and 12 are partially supported), and obtained very promising results. For instance, we have developed techniques to understand the various constraints of the kernel invariants (from robustness and trustworthiness perspective) in year one [2, 7, 16, 19, 21], extracted many of them (e.g., point-to relations among data structures) in year two [9], and built defense

frameworks by using the tap points [20], hardware features in year three [11, 14]. Finally, we have significantly improved the binary analysis in year four and five, and made them more practical for security applications [3, 13].

3 Major Achievement of the Project

3.1 Building a Dynamic Binary Analysis Platform to Analyze OS Kernel

Developing an API-Rich, Cross-OS Dynamic Binary Instrumentation Framework. Since the key goal of this project is to develop dynamic kernel binary analysis to extract kernel invariants and use them for kernel malware defense, we need to have a platform that can perform our desired analysis. Today, there are many dynamic binary instrumentation (DBI) platforms such as PIN, VALGRIND, DYNAMORIO, QEMU, and BOCHS. Each platform is built atop its own virtual machine (VM), and has its own pros and cons. The first step in this project is to build a new out-of-VM DBI with an emphasis on supporting security applications. We have successfully built such a platform, and published the following paper to describe the details on how we design, implement and evaluate it.

VEE	J. Zeng, Y. Fu and **Z. Lin**, "PEMU: A Pin Highly Compatible Out-of-VM Dynamic Binary Instrumentation Framework", In *Proceedings of the 11th International Conference on Virtualization Execution Environments*, Istanbul, Turkey, March 2015

Source code availability. Also, note that the code of this platform has also been made public available at https://github.com/utds3lab/pemu

3.2 Understanding the State-of-the-art in Hypervisor Layer Kernel Malware Defense, and its Robustness and Trustworthiness

An important question we need to answer in this project is how to fundamentally secure the OS kernel. Historically, we have learned that using a layer below monitoring is a fundamental mechanism for maintaining systems security. Intrusion detection, access control, sandboxing, inlined reference monitors, firewalls, and anti-viruses all involve security monitoring. An ideal monitoring system should have both a complete view of the monitored target (the OS kernel in our case) and the ability to (stealthily) protect the monitoring system itself. While there are many ways to do so, it is not a simple task. Over the past few decades, a large amount of research has been carried out to search for better and more secure ways to develop such monitors, particularly from the virtual machine (VM) layer.

Survey. By tracing the evolution of out-of-VM security monitoring, we have examined and classified different approaches that have been proposed to overcome the semantic gap, the fundamental

challenge in hypervisor-based monitoring, and how they have been used to develop various security applications. In particular, we reviewed how the past approaches address different constraints such as practicality, flexibility, coverage, and automation while bridging the semantic gap; how they have developed different monitoring systems; and how the monitoring systems have been applied and deployed. In addition to systematizing all of the proposed techniques, we also discussed the remaining research problems and shed light on the future directions of hypervisor based monitoring. The details about our result are presented in the following paper.

CSUR	E. Bauman, G. Ayoade and **Z. Lin**, "A Survey on Hypervisor Based Monitoring: Approaches, Applications, and Evolutions", *ACM Computing Surveys*, August 2015.

Robustness . Kernel memory analysis is crucial to identify the kernel level attacks, especially for a live system. It is increasingly valuable, especially in cloud computing. However, kernel memory analysis on commodity operating systems (such as Microsoft Windows) faces the following key challenges: (1) a partial knowledge of kernel data structures; (2) difficulty in handling ambiguous pointers; and (3) lack of robustness by relying on soft constraints that can be easily violated by kernel attacks.

To address these challenges, we present AUTOTAP, a memory analysis system that can extract a more complete view of the kernel data structures for closed-source operating systems and significantly improve the robustness by only leveraging pointer constraints (which are hard to manipulate) and evaluating these constraints globally (to even tolerate certain amount of pointer attacks). We have evaluated AUTOTAP on 100 memory images for Windows XP SP3 and Windows 7 SP0. Overall, AUTOTAP can construct a kernel object graph from a memory image in just a few minutes, and achieves over 95% recall and over 96% precision. Our experiments on real-world rootkit samples and synthetic attacks further demonstrate that AUTOTAP outperforms other external memory analysis tools with respect to wider coverage and better robustness. We have presented the details about these findings as well how we design experiments in the following paper.

ACSAC	Q. Feng, A. Prakash, H. Yin, and **Z. Lin**, "MACE: High-Coverage and Robust Memory Analysis For Commodity Operating Systems", In Proceedings of the *30th Annual Computer Security Applications Conference*, New Orleans, Louisiana, December 2014.

Trustworthiness . Memory analysis serves as a foundation for many security applications such as memory forensics, virtual machine introspection and malware investigation. However, malware, or more specifically a kernel rootkit, can often tamper with kernel memory data, putting the trustworthiness of memory analysis under question. With the rapid deployment of cloud computing and increase of cyber attacks, there is a pressing need to systematically study and understand the problem of memory analysis. In particular, without ground truth, the quality of the memory analysis tools widely used for analyzing closed-source operating systems (like Windows) has not been

thoroughly studied. Moreover, while it is widely accepted that value manipulation attacks pose a threat to memory analysis, its severity has not been explored and well understood. To answer these questions, we have devised a number of novel analysis techniques including (1) binary level ground-truth collection, and (2) value equivalence set directed field mutation. Our experimental results demonstrate not only that the existing tools are inaccurate even under a non-malicious context, but also that value manipulation attacks are practical and severe. Finally, we show that exploiting information redundancy can be a viable direction to mitigate value manipulation attacks, but checking information equivalence alone is not an ultimate solution. The details about these research results are published in the following paper.

TDSC	A. Prakash, E. Venkataramani, H. Yin, and **Z. Lin**, "On the Trustworthiness of Memory Analysis—An Empirical Study from the Perspective of Binary Execution", *IEEE Transactions on Dependable and Secure Computing*, October 2014.

3.3 Automatically Deriving Pointer Reference Expressions and Derandomizing Addresses from Binary Code for Integrity Check and Repair

Kernel Pointer Integrity Check and Repair with Reference Expressions. A pointer, whose value is a memory address, is ubiquitous in a large body of software especially those written in C/C++. Recognizing and locating pointers in a memory (crash) dump is valuable in many applications. In program debugging, pointers are the root cause of segmentation fault. Given a crash dump, if we can locate where the crashed pointer is, it will significantly help the bug reporting. In security, pointers especially the ones pointing to program code (i.e., function pointers), are often the direct targets for control flow hijacks. For instance, over 96% of kernel rootkits hijack kernel function pointers to subvert normal control flow of the OS kernel. Given a running program or an OS kernel, if we can locate its function pointers, we would have been able to check their integrity and detect the control flow violations.

To advance the state-of-the-art, in this work we introduce a new concept called *pointer reference expression* (ptr-rexp for short) and we show that such an expression is an invariant that can be extracted from binary code and used to locate pointers in memory. More specifically, ptr-rexp encodes how a pointer is accessed through the combination of a base address (usually a global variable) with certain offset and further pointer dereferences. With ptr-rexp, we can then traverse a memory dump by following from the root of the pointer (e.g,. a global variable that is static) to reach the target locations. To derive ptr-rexp, we present a new dynamic binary analysis that tracks the dependences of how a memory address is computed. This analysis starts from a pointer data-use point (e.g., an indirect function call), and backward resolves the dependences until reaching the root of the pointer, namely a global variable address. Such a resolution process can directly produce a run-time address-independent ptr-rexp since global address is usually static, which can be used for cross checking.

As an application of our techniques, we demonstrate how to use this invariant for kernel memory

dump analysis, especially for the checking of kernel function pointer integrity. To this end, we have to solve another challenge: how to determine whether a pointer is hijacked. We propose a *pointer integrity checking* technique. We base our technique on the observation that after a program is compiled, the instructions (i.e., the *code*) are usually static, and the difference between the same program on two machines is the program *data*. As such, with our address-independent ptr-rexp, we can simultaneously traverse two memory snapshots: one is from a trusted kernel and the other is from the untrusted one. While the identified function pointers could be located in the dynamically allocated program addresses, which can differ simply due to the behavior of the program heap allocators, fortunately our ptr-rexp is exactly designed to enable appropriate pointer integrity comparison between an untrusted kernel and the trusted one, and we can compare either their values or their targeted code to determine it has been hijacked. The details of our paper are described in the following FSE paper.

FSE	Y. Fu, Z. Lin, and D. Brumley, "Automatically Deriving Pointer Reference Expressions From Executions For Memory Dump Analysis", In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, BERGAMO, ITALY, September 2015

Derandomizing Kernel ASLR Address space layout randomization (ASLR) has become a prominent defense against the attacks that use a hard-coded address to compromise vulnerable systems. Examples of such attacks include Internet worms that use the same virtual address to compromise the control flow of the same vulnerable program, or some kernel rootkits that overwrite the same virtual address to hide or redirect the kernel control flow. At a high level, ASLR randomizes the base address of program code and data including both heap and stack. Consequently, traditional memory exploits through return-into-libc or return oriented programming (ROP) can be mitigated because of the memory address diversity enabled by ASLR. ASLR has also been pushed to the kernel space due to the existence of exploitable vulnerabilities in OS kernels as well as the threats from kernel rootkits. Modern OS kernels such as Windows, Linux, and Mac OS all have adopted ASLR to randomize both the kernel code and the kernel data including those in kernel global, heap and stack regions. As such, the address of kernel code and data (e.g., system call dispatcher table) will be relocated to different memory locations in different runs.

The implication of kernel ASLR has twofold: on one hand it significantly decreases the success rate of kernel memory exploits as well as some kernel rootkit attacks; on the other hand it also hinders the application of online kernel introspection and offline kernel memory forensics, both of which need to interpret (or reconstruct) kernel events outside of the (guest) OS. Specifically, for an introspection and forensic tool to be effective, it often requires a pre-knowledge of the OS kernel such as where kernel code and important kernel data structure is located. For instance, to interpret a system call event, it requires to know the address of the system call tables; to intercept the kernel object allocation and deallocation, it requires to know the addresses of the functions that manages the kernel heaps; to traverse certain dynamically allocated kernel objects, it needs to know their rooted global addresses. Unfortunately, kernel ASLR will randomize these addresses, and we must

derandomize them for introspection and forensics.

Therefore, we have conducted the first systematic study to search for the optimal solutions for introspection and forensics to derandomize the kernel ASLR. In particular, since the key challenge lies in deriving the strong and robust invariants inside kernel memory, we systematically examine both kernel read-only code and data that can be used to derandomize the ASLR. For read-only kernel data, we examine the strings and entries of code pointers (e.g., jump tables) and we propose to use the entries of the code pointers as the invariants. For kernel code, we examine how kernel code is updated, from which to derive the invariants. We also perform a comparison study among these approaches by using robustness and efficiency metrics. The details of this study is published in the following CODASPY'16 paper.

CODASPY Y. Gu, Z. Lin, "Derandomizing Kernel Address Space Layout for Introspection and Forensics", In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy*, New Orleans, LA, March 2016.

3.4 Automatic Inference of Kernel Object Semantics and Uncovering of from Binary Code

Uncovering the semantics (i.e., the meanings) of kernel objects is important to a wide range of security applications, such as virtual machine introspection (VMI), memory forensics, and kernel internal function inference. For example, knowing the meaning of the `task_struct` kernel object in the Linux kernel can allow VMI tools to detect hidden processes by tracking the creation and deletion of this data structure. In addition, knowing the semantics of `task_struct` enables security analysts to understand the set of functions (e.g., the functions that are responsible for the creation, deletion, and traversal of `task_struct`) that operate on this particular data structure.

However, uncovering the semantics of kernel objects is challenging for a couple of reasons. First, an OS kernel tends to have a large number of objects (up to tens of thousands of dynamically created ones with hundreds of different semantic types). It is difficult to associate the meanings to each kernel object when given such a large number. Second, semantics are often related to meaning, which is very vague even to human beings. It is consequently difficult to precisely define semantics that can be reasoned by a machine. In light of these challenges, current practice is to merely rely on human beings to *manually* inspect kernel source code, kernel symbols, or kernel APIs to derive and annotate the semantics of the kernel objects.

To advance the state-of-the-art, we present ARGOS (published in RAID 2015), the first system for *A*utomatic *R*everse en*G*ineering of kernel *O*bject *S*emantics. To have a wider applicability and practicality, ARGOS works directly on the kernel binary code without looking at any kernel source code or debugging symbols. Similar to many other data structure (or network protocol) reverse engineering systems, it is based on the principle of *data uses tell data types*. Specifically, it uses a dynamic binary code analysis approach with the kernel binary code and the test suites as input, and outputs the semantics for each observed kernel object based on how the object is used.

RAID	J. Zeng, and Z. Lin, "Towards Automatic Inference of Kernel Object Semantics from Binary Code", In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses*, Kyoto, Japan. November 2015

A tap point is an execution point where active monitoring can be performed. Uncovering tap points inside an OS kernel is important to many security applications such as virtual machine introspection (VMI), kernel malware detection, and kernel rootkit profiling. For example, by tapping the internal execution of the creation and deletion of process descriptors, it can enable a VMI tool to track the active running processes. Prior systems mainly hook the execution of the public exported APIs (e.g., system calls such as `fork` in Linux) to track the kernel object creation (e.g., `task_struct`). However, attackers can actually use some of the internal functions to bypass the check and create the "hidden" objects. Therefore, it would be very useful if we can automatically identify these internal tap points and hook them for the detection.

To advance the state-of-the-art, we present AUTOTAP (published in RAID 2016), a system for AUTOmatic uncovering of TAP points directly from kernel binary. We focus on the tap points that are related to kernel objects since kernel malware often manipulates them. In particular, based on how an object is accessed, we classify the tap points into *creation, initialization, read, write, traversal,* and *destroy*. By observing which execution point is responsible for these object accesses, we derive the corresponding tap points.

Having the capability of uncovering the tap points, AUTOTAP will be valuable in many security applications. One use case is we can apply AUTOTAP to detect the hidden kernel objects by tapping the internal kernel object access functions. Meanwhile, we can also use AUTOTAP to reverse engineer the semantics of kernel functions. For instance, with AUTOTAP we can now pinpoint the function that creates, deletes, initializes, updates, and traverses kernel objects. In addition, we can also identify common functions that operate with many different type of objects, which will be particularly useful to uncover the meanings of kernel functions especially for closed source OS. The details about AUTOTAP are described in the following RAID paper.

RAID	J. Zeng, Y. Fu, and Z. Lin, "Automatic Uncovering of Tap Points From Kernel Executions", In *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions and Defenses*, Paris, France. September 2016

3.5 Using Hardware Features for Software Defense

Building New Control Flow Integrity with New Hardware Features. Control flow hijacking has been one of the most severe cyber threats for over 40 years. When given an exploitable vulnerability such as a buffer overflow in a program that consumes untrusted input, an attacker can directly compromise the execution of the program and perform whatever malicious actions of his or her wishes. Over the past few decades, we have witnessed numerous such attacks. Stack smashing, return-into-libc, return oriented programming (ROP) (and its variants such as BROP and JIT-ROP), jump-oriented programming (JOP), and even call-oriented programming (e.g., COOP) all belong to this category. It is likely that these attacks will continue to remain a major cyber threat for years to

come.

Correspondingly, numerous defenses have been proposed to defend against control flow hijacking. Notable examples include stack canary (which can defeat stack smashing), data execution prevention (DEP) (which can defeat code injection), address space layout randomization (ASLR) (which can make the hijack exploit code much harder to construct), and control flow integrity (CFI) (which aims to ensure the integrity of control flow transfer always following legal program path). Canary, DEP, and ASLR are all practical defenses and they all have been adopted by industry in mainstream computing devices including even in the mobile platform. To really defeat these advanced attacks, it appears that CFI is the most promising technique since in theory it can fundamentally solve the control flow hijacking problem because all these attacks including ROP violate the intended program control flow. However, in practice CFI has not been widely adopted yet, at least in the case of protecting commercial-off-the-shelf (COTS) binaries.

In this work, we propose PT-CFI, a practical backward-edge CFI that works for x86 COTS binaries by using a recent hardware feature, the Intel Processor Trace (PT). While Intel had offered prior hardware-based tracing features such as LBR and BTS, PT provides many compelling features. In particular, the path history recorded by LBR is limited to a few dozen instructions, and BTS has significant slowdown though it supports unlimited path history. Therefore, Intel recently introduced PT, which can log execution trace with extremely low performance impact (less than 5% performance overhead) and provide a complete control flow tracing with both cycle count and timestamp information. We have implemented PT-CFI and evaluated with both the SPEC2006 CPUINT benchmark suite and Nginx HTTP daemon. Experimental results show that PT-CFI only introduces very small overhead for the protected binaries. The detail about our PT-CFI is described in the following CODASPY paper.

CODASPY	Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, "PT-CFI: Transparent Backward-Edge Control Flow Violation Detection Using Intel Processor Trace", In *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy*, Scottsdale, Arizona. March 2017

Type Sensitive CFI. Programs aiming for low runtime overhead and high availability draw on several object-oriented features available in the C/C++ programming language, such as dynamic object dispatch. However, there is an alarmingly high number of object dispatch (i.e., forward-edge) corruption vulnerabilities, which undercut security in significant ways and are in need of a thorough solution. In this paper, we propose τCFI, an extended control flow integrity (CFI) model that uses both the types and numbers of function parameters to enforce forward- and backward-edge control flow transfers. At a high level, it improves the precision of existing forward-edge recognition approaches by considering the type information of function parameters, which are directly extracted from the application binaries. Therefore, τCFI can be used to harden legacy applications for which source code may not be available. We have evaluated τCFI on real world binaries including Nginx, NodeJS, Lighttpd, MySql and the SPEC CPU2006 benchmark and demonstrate that τCFI is able to effectively protect these applications from forward- and backward-edge corruptions

with low runtime overhead. In direct comparison with state-of-the-art tools, τCFI achieves higher forward-edge caller-callee matching precision

RAID	P. Muntean, M. Fischer, G. Tan, Z. Lin, J. Grossklags ,and C. Eckert, "τCFI: Type-Assisted Control Flow Integrity for x86-64 Binaries", In *Proceedings of the 21st International Symposium on Research in Attacks, Intrusions and Defenses*, September 2018

3.6 Improving Binary Code Analysis and Binary Rewriting

Type Inference with Binary Code. Being the final deliverable of software, executables (or binary code, as we use both terms interchangeably) are everywhere. They contain the final code that runs on a system and truly represent the program behavior. In many situations, such as when analyzing commercial-off-the-shelf (COTS) programs, malware, or legacy programs, we can only access program executables since the source code and debugging symbols are not available.

Analyzing executables is challenging because during compilation much program information is lost. One particularly critical piece of missing information is the *variables* that store the data, and their *type*, which constrains how the data is stored, manipulated, and interpreted. Given their importance, for the last 16 years a large amount of research has been carried out on *binary code type inference*, a challenging task that aims to infer typed variables from executables.

In this work we first systematize the area of binary code type inference according to its most important dimensions: the applications that motivate its importance, the proposed approaches, the types that those approaches infer, the implementation of those approaches, and how the inference results are evaluated. We also discuss limitations and point to underdeveloped problems and open challenges. This work is published in the following survey paper.

CSUR	J. Caballero, and Z. Lin, "Type Inference on Executables", In *ACM Computing Surveys*, Volume 48 Issue 4, May 2016

Superset Disassembly. Static binary rewriting is a core technology for many systems and security applications, including profiling, optimization, and software fault isolation. While many static binary rewriters have been developed over the past few decades, most make various assumptions about the binary, such as requiring correct disassembly, cooperation from compilers, or access to debugging symbols or relocation entries. This paper presents MULTIVERSE, a new binary rewriter that is able to rewrite Intel CISC binaries without these assumptions. Two fundamental techniques are developed to achieve this: (1) a superset disassembly that completely disassembles the binary code into a superset of instructions in which all legal instructions fall, and (2) an instruction rewriter that is able to relocate all instructions to any other location by mediating all indirect control flow transfers and redirecting them to the correct new addresses. A prototype implementation of MULTIVERSE and evaluation on SPECint 2006 benchmarks shows that MULTIVERSE is able to rewrite all of the testing binaries with a reasonable runtime overhead for the new rewritten binaries. Simple static instrumentation using MULTIVERSE and its comparison with dynamic

instrumentation shows that the approach achieves better average performance. Finally, the security applications of MULTIVERSE are exhibited by using it to implement a shadow stack. The paper that describes the details of our superset disassembly is published in the following NDSS paper.

NDSS	E. Bauman, Z. Lin, and K. Hamlen. "Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics", In *Proceedings of the 25th ISOC Network and Distributed System Security Symposium*, San Diego, CA, February 2018

Probabilistic Disassembly. Disassembling stripped binaries is a prominent challenge for binary analysis, due to the interleaving of code segments and data, and the difficulties of resolving control transfer targets of indirect calls and jumps. As a result, most existing disassemblers have both false positives (FP) and false negatives (FN). We observe that uncertainty is inevitable in disassembly due to the information loss during compilation and code generation. Therefore, we propose to model such uncertainty using probabilities and propose a novel disassembly technique, which computes a probability for each address in the code space, indicating its likelihood of being a true positive instruction. The probability is computed from a set of features that are reachable to an address, including control flow and data flow features. Our experiments with more than two thousands binaries show that our technique does not have any FN and has only 3.7% FP. In comparison, a state-of-the-art superset disassembly technique has 85% FP. A rewriter built on our disassembly can generate binaries that are only half of the size of those by superset disassembly and run 3% faster. While many widely-used disassemblers such as IDA and BAP suffer from missing function entries, our experiment also shows that even without any function entry information, our disassembler can still achieve 0 FN and 6.8% FP. The details of this paper are presented in the following ICSE paper.

ICSE	K. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin. "Probabilistic Disassembly", In *Proceedings of 41st ACM/IEEE International Conference on Software Engineering*, May 2019.

4 Other Achievements Partially Supported by this Project

In addition to the major achievements described in §3 (i.e., [2,3,5,7,9,10,11,13,14,16,19,20,21]), this project has also partially contributed to a few other projects such as binary code analysis with mobile platform (e.g., SmartGen [23], AuthScope [25], SkyWalker [1], LeakScope [24]), secure heap memory allocator (e.g., FreeGuard [17], Guarder [18]), IoTFuzzer [6]), privacy leakage in drivers [22], binary code decomposition [12], defending cryptographic key leakage in cloud VM [15], side channel defense in SGX [8], and understanding of password security policy [4]. All of these papers have explicitly acknowledged the AFOSR support of grant **FA9550-14-1-0119**.

References

[1] Omar Alrawi, Chaoshun Zuo, Ruian Duan, Ranjita Kasturi, Zhiqiang Lin, and Brendan Saltaformaggio. The betrayal at cloud city: An empirical analysis of cloud-based mobile backends. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019.

[2] Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. A survey on hypervisor based monitoring: Approaches, applications, and evolutions. *ACM Computing Surveys*, 48(1):10:1–10:33, August 2015.

[3] Erick Bauman, Zhiqiang Lin, and Kevin Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18)*, San Diego, CA, February 2018.

[4] Erick Bauman, Yafeng Lu, and Zhiqiang Lin. Half a century of practice: Who is still storing plaintext passwords? In *Proceedings of the 11th International Conference on Information Security Practice and Experience*, Beijing, China, May 2015.

[5] Juan Caballero and Zhiqiang Lin. Type inference on executables. *ACM Computing Surveys*, 48(4):65:1–65:35, May 2016.

[6] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18)*, San Diego, CA, February 2018.

[7] Qian Feng, Aravind Prakash, Heng Yin, and Zhiqiang Lin. Mace: High-coverage and robust memory analysis for commodity operating systems. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC'14)*, New Orleans, Louisiana, December 2014.

[8] Yangchun Fu, Erick Bauman, Raul Quinonez, and Zhiqiang Lin. Sgx-lapd: Thwarting controlled side channel attacks via enclave verifiable page faults. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'17)*, Atlanta, Georgia. USA, September 2017.

[9] Yangchun Fu, Zhiqiang Lin, and David Brumley. Automatically deriving pointer reference expressions from executions for memory dump analysis. In *Proceedings of the 2015 ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'15)*, Bergamo, Italy, September 2015.

[10] Yufei Gu and Zhiqiang Lin. Derandomizing kernel address space layout for introspection and forensics. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy*, New Orelans, LA, 2016. ACM.

[11] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace. In *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy*, Scottsdale, Arizona, USA, march 2017. ACM.

[12] Vishal Karande, Swarup Chandra, Zhiqiang Lin, Juan Caballero, Latifur Khan, and Kevin Hamlen. Bcd: Decomposing binary code into components using graph-based clustering. In *Proceedings of the 13th ACM Symposium on Information, Computer and Communications Security*, June 2018.

[13] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. Probabilistic disassembly. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE'19, pages 1187–1198, Montreal, Quebec, Canada, 2019.

[14] Paul Muntean, Matthias Fischer, Gang Tan, Zhiqiang Lin, Jens Grossklags, and Claudia Eckert. cfi: Type-assisted control flow integrity for x86-64 binaries. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 423–444, 2018.

[15] Erman Pattuk, Murat Kantarcioglu, Zhiqiang Lin, and Huseyin Ulusoy. Preventing cryptographic key leakage in cloud virtual machines. In *Proceedings of the 2014 USENIX Security Symposium*, San Diego, CA, August 2014.

[16] Aravind Prakash, Eknath Venkataramani, Heng Yin, and Zhiqiang Lin. On the trustworthiness of memory analysis—an empirical study from the perspective of binary execution. *IEEE Transactions on Dependable and Secure Computing*, 2014.

[17] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. Freeguard: A faster secure heap allocator. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS'17)*, Dallas, TX, November 2017.

[18] Sam Silvestro, Hongyu Liu, Tianyi Liu, Zhiqiang Lin, and Tongping Liu. Guarder: a tunable secure allocator. In *27th USENIX Security Symposium*, pages 117–133, 2018.

[19] Junyuan Zeng, Yangchun Fu, and Zhiqiang Lin. Pemu: A pin highly compatible out-of-vm dynamic binary instrumentation framework. In *Proceedings of the 11th Annual International Conference on Virtual Execution Environments*, Istanbul, Turkey, March 2015.

[20] Junyuan Zeng, Yangchun Fu, and Zhiqiang Lin. Automatic uncovering of tap points from kernel executions. In *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'16)*, Paris, France, September 2016.

[21] Junyuan Zeng and Zhiqiang Lin. Towards automatic inference of kernel object semantics from binary code. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'15)*, Kyoto, Japan, November 2015.

[22] Qingchuan Zhao, Chaoshun Zuo, Giancarlo Pellegrino, and Zhiqiang Lin. Geo-locating drivers: A study of sensitive data leakage in ride-hailing services. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS'18)*, San Diego, CA, February 2019.

[23] Chaoshun Zuo and Zhiqiang Lin. Smartgen: Exposing server urls of mobile apps with selective symbolic execution. In *Proceedings of the 26th World Wide Web Conference (WWW'17)*, Perth, Australia, April 2017.

[24] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2019.

[25] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. Authscope: Towards automatic discovery of vulnerable authorizations in online services. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS'17)*, Dallas, TX, November 2017.