ORIGEN: Automatic Extraction of Offset-Revealing Instructions for Cross-Version Memory Analysis

Qian Feng[†] Aravind Prakash[◊] Minghua Wang[†$] Curtis Carmony[†] Heng Yin[†]

[†]Department of EECS, Syracuse University, USA
[◊]Computer Science Department, Binghamton University
[$]Baidu Security Lab, Baidu Inc.
[†]{qifeng,ccarmony,heyin}@syr.edu [◊]aravind@cs.binghamton.edu [$]wangminghua01@baidu.com

ABSTRACT

Semantic gap is a prominent problem in raw memory analysis, especially in Virtual Machine Introspection (VMI) and memory forensics. For COTS software, common memory forensics and VMI tools rely on the so-called "data structure profiles" – a mapping between the semantic variables and their relative offsets within the structure in the binary. Construction of such profiles requires the expert knowledge about the internal working of a specified software version. At most time, it requires considerable manual efforts, which often turns out to be a cumbersome process. In this paper, we propose a notion named "cross-version memory analysis", wherein our goal is to alleviate the process of profile construction for new versions of a software by transferring the knowledge from the model that has already been trained on its old version. To this end, we first identify such *Offset Revealing Instructions* (ORI) in a given software and then leverage the code search techniques to label ORIs in an unknown version of the same software. With labeled ORIs, we can localize the profile for the new version. We provide a proof-of-concept implementation called ORIGEN. The efficacy and efficiency of ORIGEN have been empirically verified by a number of softwares. The experimental results show that by conducting the ORI search within Windows XP SP0 and Linux 3.5.0, we can successfully recover data structure profiles for Windows XP SP2, Vista, Win 7, and Linux 2.6.32, 3.8.0, 3.13.0, respectively. The systematical evaluation on 40 versions of OpenSSH demonstrates ORIGEN can achieve a precision of more than 90%. As a case study, we integrate ORIGEN into a VMI tool to automatically extract semantic information required for VMI. We develop two plugins to the Volatility memory forensic framework, one for OpenSSH session key extraction, the other for encrypted filesystem key extraction. Both of them can achieve the cross-version analysis by ORIGEN.

ASIA CCS '16, May 30-June 03, 2016, Xi'an, China

DOI: http://dx.doi.org/10.1145/2897845.2897850

1. INTRODUCTION

Memory analysis aims at extracting security-critical information from a memory snapshot of a running system or a program. It has many security applications, such as virtual machine introspection [16], malware detection and analysis [21], game hacking [3], digital forensics [12,38], etc. Most of these applications require retrieving desired information from a memory snapshot of a running software or system, so we refer to them as memory analysis tools in general.

For all these memory analysis applications, we need to have the precise knowledge about data structures that are relevant to the specific analysis purpose. Most of existing memory analysis tools usually build a data structure profile, i.e. a mapping between data structures to their offsets in the target binary, to derive analysis decisions. The data structure profile is constructed to incorporate precise knowledge about data structures. For instance, we may build a precise data structure profile about the offset values of important fields, such as the process name, process ID, and the pointer to the next `EPROCESS` structure, in the `EPROCESS` data structure in order to retrieve running processes from a memory snapshot for Windows OS.

The creation and maintenance of the data structure profile is a nontrivial problem, especially for COTS binaries. It requires the expert knowledge about the internal working of the target software. Existing work, such as Volatility [38], VMST [13] and Virtuoso [9], have made a big progress on automatic introspection code generation. Their techniques work well when the target software is open-source [9,38], or when the well-defined code pieces are provided, which can be reused for introspection [13].

For COTS software, however, existing memory analysis tools still rely on cumbersome reverse engineering techniques to build the profile. In most cases, the profile generation still depends on the manual effort. Unfortunately, the daunting profile creation task is not a one-time effort. It is tightly coupled to the specific version of the software being analyzed, and needs to be constantly rebuilt for new versions of the software. As a result, the effort spent on building the analysis profile for one particular version of a program could not be applicable to its future versions. For example, a memory analysis tool, such as Volatility [38], has to create a profile for every version of a COTS software to be analyzed. Once the version is changed, the profile has to be manually updated for the exact same software so that the analysis can proceed correctly.

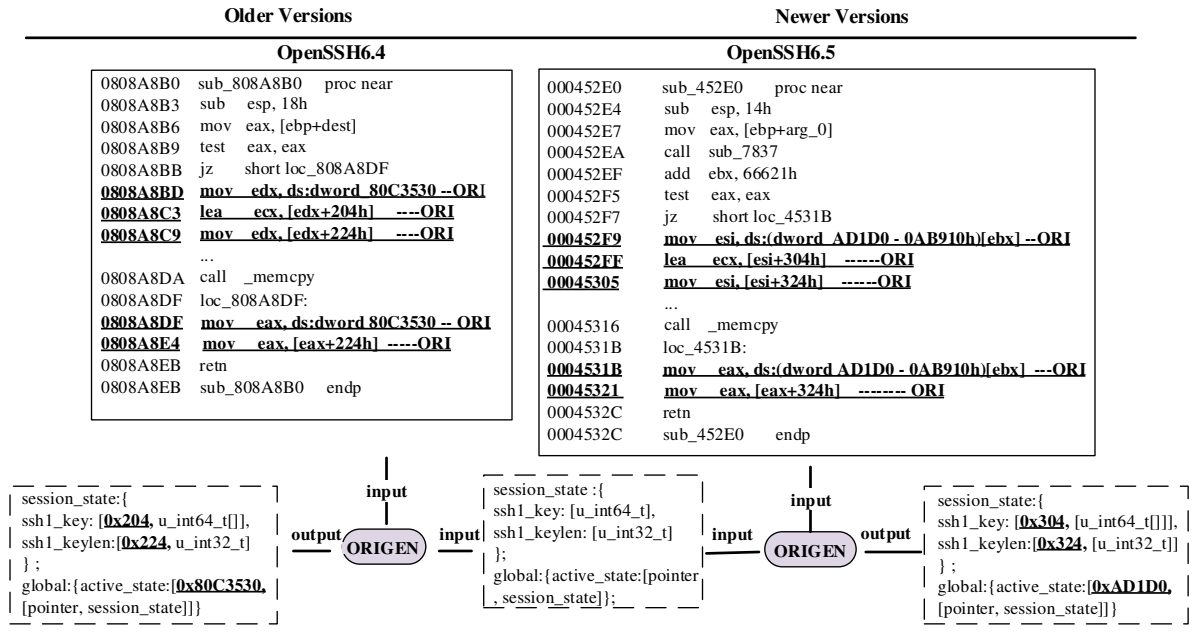


Figure 1: The OpenSSH example. It shows code snippets to retrieve the session key for openssh in two versions. Offset-Revealing Instructions (ORIs) are highlighted in both versions. Given the abstract profile, the profile localization determines the offsets from the identified ORIs and produces a localized profile for each version.

In this paper, we propose a novel notion of "cross-version memory analysis". That is, the data structure profile used in one version can be adapted to other versions of the same software without manual efforts. With the *cross-version memory analysis* property, we can automatically build profiles for new versions of a software by transferring the knowledge from the profile that has already been trained for its old version. Our intuition is that adjacent versions of the same software tend to be similar. The experimental results in Section 6.2 substantiate this claim. Based on this idea, we can transfer the relevant knowledge from an already trained profile to build the profile for an unseen new version. The less different a new version is from the previous version, the more accurately the profile can proceed the analysis .

To achieve the cross-version memory analysis, we combine program analysis and code searching techniques to automatically transfer the data structure profile across different versions of a software. We observed that some instructions, at the binary level, reveal the actual offsets (as constant values) for the specified data structure fields and global variables, as these offsets have been statically determined at compile time. We name these instructions "offset-revealing instructions" (in short, ORI). Given a trained profile on one version, we label ORIs in the binary of this version by program analysis techniques. With the knowledge of learned ORIs in this version, we can identify semantically-equivalent ORIs in its new versions by the code searching technique, and localize the introspection profile by updating offset values for correspondent data structure fields based on identified ORIs.

We have developed a prototype system called ORIGEN and evaluated its capability on a number of software families including Windows OS kernel, Linux OS kernel, and `OpenSSH`. Particularly, we systematically evaluate it on 40 versions of `OpenSSH`, released between 2002 and 2015. The experimental results show that ORIGEN can achieve a precision of about 90% by transferring relevant knowledge in the profile of a different version automatically. The results suggest that ORIGEN advances the existing memory analysis methods by reducing the manual efforts while maintaining the reasonable accuracy. We further have developed two plugins to the Volatility memory forensic framework [38] and integrated them in ORIGEN, one for OpenSSH session key extraction, and the other for encrypted filesystem key extraction. We show that each of the two plugins can construct a localized profile and then can perform specified memory forensic tasks on the same memory dump, without the need of manual effort in creating the corresponding profile.

Certainly, we admit that ORIGEN may not work when our assumption does not hold, i.e. when a software version is significantly different from the base version on which the ORI signatures are generated. For these cases, we can generate a new profile to cover its ORI signatures and apply to many other similar versions. Nevertheless, ORIGEN introduces a promising solution for cross-version memory analysis and demonstrates an empirically validated approach to greatly reducing the manual effort for profile creation. The research along this direction is important because it could streamline the memory analysis process, with minimal manual intervention required.

In summary, the contribution of this paper is threefold:

- we propose a novel notion of cross-version memory analysis. We made the first attempt to conduct the

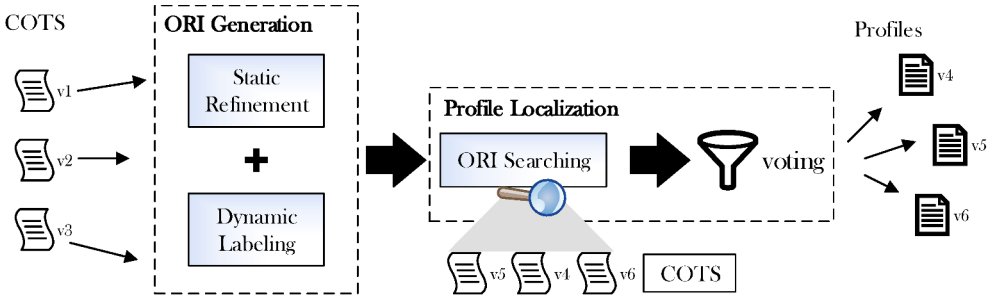


Figure 2: The overview of ORIGEN

memory analysis across different versions of the software. Our study demonstrates that the across-version memory analysis can be achieved with a minimal or reduced human intervention.

- we developed a prototype system ORIGEN, which combines the program analysis and code search technique to address the new problem domain.
- we systematically evaluated the accuracy of ORIGEN under 40 versions of the `OpenSSH` family, and the evaluation results show that ORIGEN can achieve a precision of more than 90%. The case studies also demonstrate ORIGEN can successfully recover the offsets for key semantic fields across different versions of `OpenSSH`, Windows, Linux, a loadable kernel module for Linux.

2. OVERVIEW

We utilize a running example in Figure 1 to demonstrate our problem. Although we target at the memory analysis for the COTS software, for clarity, we utilize the open-source software `OpenSSH` to demonstrate our basic idea. Figure 1 shows code snippets for two versions of OpenSSH (6.4 and 6.5), where several highlighted instructions are used to access `ssh1_key` and `ssh1_keylen` fields in the structure of `session_state`, and a global variable `active_state`, which points to the structure `session_state`. The constant values carried by these instructions indicate the exact offsets of these fields inside the data structure. Therefore, these highlighted instructions are ORIs.

In this case, there are three symbols shared by OpenSSH (6.4 and 6.5). We utilize the abstract profile to denote these common symbols. Given this abstract profile, we develop an SSH key extraction tool that can locate encryption keys for active SSH sessions in a memory snapshot in the cross-version manner. ORIGEN will automatically identify ORIs in `OpenSSH6.4`, and transfer the profile for `OpenSSH6.4` to a localized profile for `OpenSSH6.5` based on identified ORIs in the older version. Using this localized profile, the SSH key extraction can immediately work for `OpenSSH6.5`, without any code modification. This demonstrates the nature of cross-version memory analysis for ORIGEN.

Problem Statement.

In this paper, we aim to achieve the cross-version memory analysis. That is, we can automatically generate profiles for new versions of a software by transferring the knowledge from the model that has already been trained on its old version. Given an abstract profile that a memory analysis tool relies on and a base version of target software, ORIGEN locates ORIs in the base version and searches these ORIs in the target version. With newly identified ORIs in the target version, we can localize the profile for the new version.

More specifically, when provided a different version of the same software, we aim to achieve the following goals: 1) identify instructions that are semantically equivalent to the ORIs identified from the base version; 2) extract the offsets from these instructions; 3) generate a localized profile for the new software version. In summary the challenge is to find ORIs in the target program of a given base version.

If we have the source code for the program to be analyzed, a straightforward way would be to use the compiler toolchain to output such information directly while the compiler generates the binary code. In many cases, the source code is often not available (e.g., VMI for Microsoft Windows). Therefore, we need to develop a binary analysis technique to extract this information from binary code.

System Overview.

Figure 2 illustrates an overview of our solution. It involves the ORI labeling and the profile localization.

In general, ORI labeling takes a base binary as the input, and performs dynamic and static analysis to finally output all labeled ORIs in the base binary. Profile localization searches a target binary for the instructions that are semantically equivalent to labeled ORIs in the base binary, and localize the profile for the target binary. The details will be discussed in latter sections.

3. ORI SIGNATURE GENERATION

3.1 ORI Signature Definition

An ORI is an instruction that has a constant field that reveals the offset of a field in the data structure definition, or the location of a global variable within the data section. The definition is as follows:

Definition 1. *Offset Revealing Instruction (ORI) is a tuple of* (p, c, t, f)*, where* p *is the program counter,* c *is the constant field within the instruction,* t *indicates the data structure type, and* f *denotes the field name within the data*

0x80037324: mov eax, [edx+0Ch]	**R**	offset 0x170 **base:** 0x80090a08 **type:** session_state
0x800370a4: mov dword ptr [eax+8], 0	**W**	offset 0x15c **base:** 0x80090a08 **type:** session_state
0x80046659: mov edx, [eax+21Ch]	**R**	offset 0x21c **base:** 0x80090a08 **type:** session_state
0x80037324: mov eax, [edx+0Ch]	**R**	offset 0x160 **base:** 0x80090a08 **type:** session_state
0x80045624: mov ecx, [esi+214h]	**R**	offset 0x214 **base:** 0x80090a08 **type:** session_state

Figure 3: The demo of the session_state object tracing log.

structure definition. For a global variable, t is "data section", and f is the name of the global variable.

3.2 ORI Labeling

In this section, we describe how we label ORIs in a binary and generate signatures for the labeled ORIs. It can be considered as a learning stage. At this stage, we attempt to learn ORI signatures which will be used for latter version-independent memory analysis.

ORIs for Global Variables.

It is straightforward to identify ORIs for global variables. Once the exact location is determined for a global variable in the base version, we can simply scan the binary code to identify all the instructions that refer to this location. The location for a global variable often has a distinct value, because it is located in the data section of the binary module. For the running example, we can see that `active_state` is a global variable and we can find its address 0x80C3530 from the debug symbol. Through scanning in the binary, we can label the 0x808ABD as an ORI directly.

For the rest of this section, we focus on ORI identification. The offsets of data structure fields are often very small, and small constant numbers are pervasive in binary code. Therefore, we use a different solution. We first dynamically trace the binary program and identify a set of instructions that access the specified data structure fields (which is described in "Dynamic Labeling"). We call these instructions "ORI candidates". Based on ORI candidates, we perform static analysis to filter out false ORIs and discover more ORIs, which is described in " Static ORI Discovery".

Dynamic ORI Labeling.

The goal of dynamic labeling is to collect a set of instructions that either read or write the given data structure field defined in the abstract profile. To do so, we need to know not only when an instance of the data structure is created and later destroyed, but the lifetime of data structure instance during the program execution. With the aid of the information about live data structure instances, we can pinpoint the instructions that access their specific fields during tracing the program execution.

To this end, we should have certain knowledge about data structures in the base version of the software. There are three types of information we need to know about the data structures in the base version: 1) The functions which create and delete the data structure instances of interest; 2) Data structure definitions that are relevant to the analysis task; 3) Actual offset for each data structure field of interest.

We hook functions which create and delete the data structure instances of interest during the binary execution to label the live data structure instances in the memory. We can further identify all instructions which have `write` or `read` operation on these live data structure instances by monitoring all the memory read and write operations during the execution. The data structure definition and its field offset information can help to extract ORIs in these identified instructions. For the programs with source code, such knowledge can be easily obtained. Even for the many binary programs (e.g., Windows), we can still obtain the knowledge from documentation of APIs. For the binary programs with limited documentation, we have to rely on reverse engineering to retrieve the needed knowledge. This is a reasonable assumption, because without this knowledge, memory analysis is not even possible in the first place.

As for our running example shown in Figure 1, we have to know the definition of `session_state` and a global variable `active_state` pointing to this structure in `OpenSSH6.4`. Moreover, for the data structure fields of interest, we need to know their actual offsets within the data structure `session_state`. Furthermore, we hook the `alloc_session_state()` function to keep track of the creation of `session_state`. As OpenSSH sever never frees the `session_state` instance, we do not hook any other functions.

When tracing the program execution, we may face several situations: (1) if an instruction does not access the field of interest at all, we simply drop it; (2) if an instruction accesses multiple data structure fields at different times, we also drop it due to its ambiguity; (3) if an instruction is observed to only access a single field of interest and the constant value carried in it matches with the field's actual offset, we treat this instruction to be an ORI; and (4) if an instruction is observed to only access a single field of interest but it does not carry a constant or the constant value does not match with the field's actual offset, we keep it as an ORI source. Although this instruction is not a real ORI by definition, it may lead us to find a real ORI through the following static analysis.

Static ORI Discovery.

Based on the ORIs and ORI sources labeled through dynamic analysis, we further perform static analysis to discover more ORIs which are missed by dynamic analysis.

Starting from an identified memory access instruction (either ORI or ORI source), we perform the backward data-flow analysis to know how the memory operand is computed. More specifically, we perform backward data-flow analysis on the memory operand in that instruction, and look for a variable that holds the base address and a constant value that holds the offset. For example, in Figure 4, the memory-access instruction at 0x402, which is the source for the ORI at 0x3fe, is first identified via dynamic analysis. ORI, by definition is an instruction of the form 'base + offset' where offset is equal to the offset within the object that the access corresponds to. We first perform backward data-flow analysis from the ORI-source to reach the ORI, then, we ex-

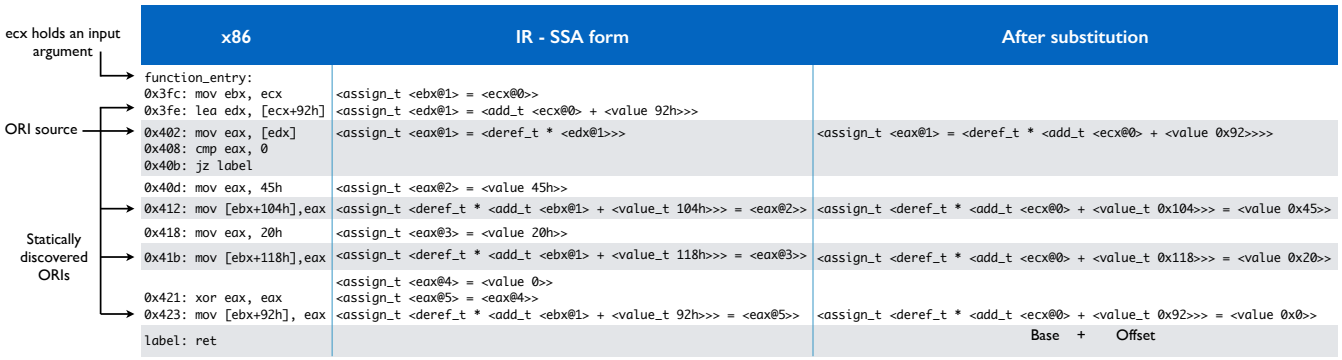


Figure 4: Static discovery of ORIs.

tend the analysis to identify the source of the base register. With the base register identified, flow-insensitive forward-data-flow analysis on the base register reveals all the ORIs present in the function. That is, in Figure 4, an ORI source at 0x402 is first identified via dynamic analysis. Then, the corresponding ORI is identified at 0x3fe. The register containing the base address is identified as `ecx@0`.

From the variable that holds the base address, we perform forward data-flow analysis within the same function to discover more ORIs. If we observe a constant value being added to the base, and that value matches with one of our data structure fields in the profile, whichever instruction carries this constant is a new ORI. In Figure 4, we start from `ecx@0` and perform forward data-flow analysis and discover ORIs at 0x412, 0x41b and 0x423.

To accomplish the said data-flow analysis, the x86 code is first converted into an IR-SSA form (column 2 in Figure 4) and the *use-def* and *def-use* chains are directly derived from them [28]. Then, the definitions are recursively propagated by substituting them into the uses until each of the statements is composed of only the entry point definitions (e.g., ecx@0 in Figure 4). Column 3 in Figure 4 presents the IR statements after substitution. In the end, we identify a statement to be an ORI if and only if (1) The expression contains a 'base + offset'[1] form where base is equal to the previously identified source of the base register (e.g., `ecx@0` in the example) and (2) The offset equals to a valid offset value within the profile.

4. PROFILE LOCALIZATION

For each symbol defined in the abstract profile, we have one or (often) multiple ORIs for the base version of a binary. To localize the profile for a new version of the binary, we try to find instructions in the new binary that match with these ORI signatures and update the profile based on the abstract profile and identified ORIs in the new binary.

4.1 ORI Identification

We consider matching ORI signatures in a new binary as a code search problem, and leverage the existing code search technique to conduct the profile localization.

To precisely label ORIs in a new binary, we need to conduct a CFG-based code search approach. The assumption is that two versions of a binary share the similar control flow graphs. This has been substantiated by existing literatures [10, 27, 33], and many other works also apply this assumption into many applications [23]. The CFG-based code search considers a instruction with the similar position in the control flow structure as a match. In this way, even if the ORI in the new binary has a different representation, the CFG-based code search can still find it, as long as two versions of the binary share similar control flow graphs.

The CFG-based search includes the control flow graph extraction and graph matching. We leverage the existing tool BinDiff [10] to achieve the CFG-based code search. It has two advantages. Firstly, its control flow graph matching and instruction alignment perfectly suit our usage scenario. Secondly, it is a mature tool with good runtime performance. Therefore, we utilize Bindiff to match the base version of a binary with the new version.

4.2 Profile Generation

The output of Bindiff is a one-to-one mapping between instructions of two binaries. We can generate the profile for the new binary, according to the abstract profile and the mapping. The profile generation is to walk through each symbol in the abstract profile and update the field offset information based its correspondent ORIs.

To this end, ORIGEN locates ORIs in the new binary based on the mapping, identifies all ORIs for each symbol, and updates offset information based on these identified ORIs. ORIGEN can locate the semantically equivalent ORI instructions in the new binary by looking up the instruction mapping. It considers instructions mapped by ORIs of the base version as qualified ORIs. ORIGEN clusters all identified ORIs by their symbol names, and update the offset information for each symbol based on its ORI cluster.

By the ORI definition in Section 3.1, we know each symbol involves the object type and field name. Each ORI cluster have one or more ORIs. If there is only one ORI in the cluster, we can directly extract its offset information from the ORI and assign it to the symbol. In most cases, the ORI cluster contains multiple ORIs. We adopt the voting mechanism to update the offset of a symbol. This is because that the CFG-based code search could introduce the erroneousness, and this could wrongly consider some instructions as ORIs. Without false ORIs, the ORIs for the same symbol share the same offset value. False ORIs will break this consistency and generate different offset values to confuse ORIGEN. The voting mechanism is designed to automat-

[1]Offset of 0 is a special case where the memory access appears like a regular dereference.

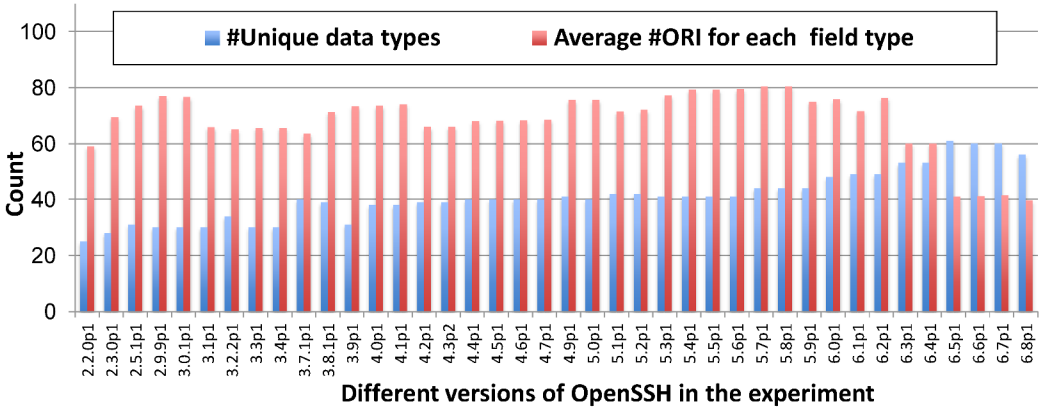


Figure 5: The statistics of the data types and the average number of ORIs to the field type in the OpenSSH dataset.

ically filter offset values from false ORIs and improve the accuracy of the profile generation.

Considering each offset value as a vote from its ORI, the voting mechanism will rank all offset values by the number of votes, and select the offset with the largest number of votes as the true offset for the symbol. Repeat this process, the profile generation will assign each symbol with an offset value and generate the profile for the new binary.

4.3 Error Correction

It is possible that ORIGEN fails to update the offset value for a symbol in the new binary, if all of ORIs of some symbol in the abstract profile are misidentified in the new binary. We adopt two strategies to resolve this problem.

The first strategy is the conservative strategy. We can filter out symbols with the high possibility to be wrongly labeled in the generated profile. Each symbol has a cluster. We use the variance from the set of offset values in the cluster to determine its false possibility. A threshold is set to determine whether the symbol is filtered or not. If the variance of the symbol value is above the threshold, we consider this symbol as a false and filter it out.

The second strategy is that we do not discard any symbol in the profile. Instead, we apply the profile to conduct the memory analysis. During the memory scanning, we collect the values from these symbols, and screen false ones by heuristics. Once we found some abnormal values, we filter the symbol from the profile.

We also can combine two strategies together to conduce the error correction. In all, the error correction can greatly reduce the false positive rate for the generated profile. This is substantiated by the experiment in Section 8.

5. IMPLEMENTATION

We have implemented the prototype of ORIGEN in C and Python. More specifically, we write the dynamic labeling plugin for DECAF [18] in C. As a whole-system dynamic analysis platform, we use DECAF to trace a user-level program, an entire OS kernel, or a specific kernel module. Besides, we write an IDA Pro plugin for static binary analysis, based on IDA-decompiler [4]. We leverage BinDiff for the ORI search. The entire ORIGEN has around 300 lines of C code and 2K lines of Python code.

6. EXPERIMENTS

This section empirically evaluates ORIGEN. First, we represent the experiment setup in Section 6.1, and then we systematically evaluate the accuracy of ORIGEN in the cross-version setting in Section 6.2 and Section 6.3. In Section 6.5, we apply ORIGEN into two use cases: memory forensics and VMI. Finally, we evaluate the runtime performance of ORIGEN in Section 6.6.

6.1 Experiment Setup

All experiments are conducted on a machine with Intel(R) Core i5 @ 2.9GHz and 16 GB DDR3-RAM running 64-bit Ubuntu 14.04. We evaluate ORIGEN on four sets of software families: including `Windows`, `Linux`, `OpenSSH` and `dm_crypt`, as shown in Table 1. To verify the accuracy of the proposed method, we systematically evaluate ORIGEN on `OpenSSH` family. For the rest of the software families, we conduct case study analysis on some representative versions.

The experimental set is representative for the following reasons: 1) the set is a sufficient sampling of real-word softwares. The versions in our experiments cover a span of 13 years of `OpenSSH`, from 2.2.0p1 in 2002 to 6.8p1 in 2015; 2) the data types and the structs in `OpenSSH` are rich and representative. For example, there are 1,904 structs and 22,618 fields in total for 40 versions of `OpenSSH`. Figure 5 illustrates the number of unique data types in each version. The size and diversity of the data should provide a systematic and objective evaluation for the proposed approach; 3) the source code of `OpenSSH` provide a gold standard for evaluating the performance of ORIGEN.

Evaluation Metrics: We employ `precision` to evaluate the performance of ORIGEN. Given a source version s, our task is to predict the offsets of correspondent data types in the target version t. The offset precision for the target version is calculated from:

$$\text{precision} = \frac{|\delta|}{|s \cap t|}, \tag{1}$$

where $|s \cap t|$ represents the total number of shared data field names in the two versions, and $|\delta|$ represents the number of correctly predicted offsets. The ground truth of the data field names can be directly obtained from the source code of `OpenSSH`. Note, the source code is not used in prediction.

Program	# of Ver	Start Ver		End Ver	
		Ver	Date	Ver	Date
Windows	3	XP3	2001	Wind7	2009
Linux	9	2.6.32	2010	3.13.0	2014
OpenSSH	40	2.2.0	2002	6.8.0	2015
dm_crypt	8	3.5	2012	3.13.0	2015

Table 1: Datasets of released versions

6.2 Overall True/False Positive Analysis

We also evaluate the accuracy of ORIGEN using the OpenSSH family. We use its 40 versions which covers a span of 13 years. Each version gets the true profile from its source code. We conduct the pair-wise profile generation on the 40 samples by ORIGEN, and calculate the offset prediction precision. For each version, ORIGEN utilizes it as a base version to localize profiles for all 40 versions. Each localized profile calculates precision by diffing itself with the true profile from the source code of that version.

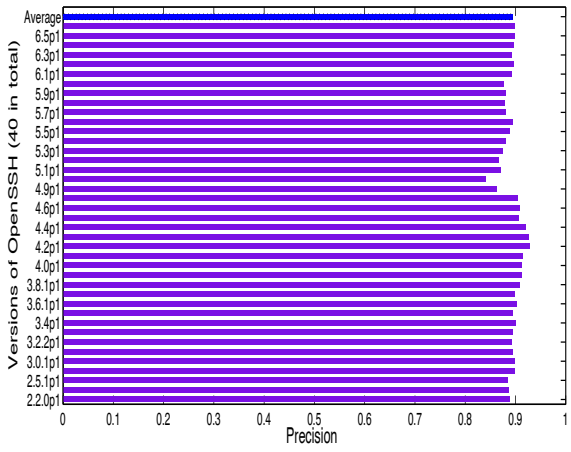


Figure 6: The average precision of our method on 40 versions of OpenSSH. The dashed bar on top shows the average.

Figure 6 shows the overall precision of ORIGEN on each test OpenSSH version, where the x-axis represents the offset prediction precision and the y-axis lists the versions. The dashed bar labeled as "average" on top represents the average precision across all 40 versions. As we see, on average, ORIGEN obtains a reasonable precision of 89.33%. The variance of the precision across versions is only 0.003, with the highest precision of 92.88% and the lowest of 83.98%. The small variance suggests that the proposed method is robust. The results shown in Figure 6 substantiate the efficacy of ORIGEN and suggests that ORIGEN points to a feasible solution for cross-version memory analysis.

We inspect the results and hypothesize that the accurate result derives from two main reasons: 1) the most of field types are referenced by multiple ORIs. A single or a few ORI searching failures can be corrected through the voting mechanism; 2) the code search based cross-version inference is resistant to some data structure reorganizations. We calculate the statistics on ORIs for each field to explain the first reason. As shown in Figure 5, we can see that each data type has more than 50 ORIs for its fields on average. Any single or a few ORI searching failures can be recorrected by rest of correct ORIs. We also manually investigate 40 reconstructed data profiles from Figure 6 and find that ORIGEN still correctly infers `connection_in` and other fields in `session_state` in `OpenSSH2.2`, even if `session_state` data structure first appears in `OpenSSH5.3`. The reason is that `OpenSSH5.3` creates `session_state` as a wrapper to wrap these fields in previous versions. The code accessing these fields are relatively stable. ORIGEN can still identify ORIs from these codes and update the type information.

We further inspect the false positives in our method and find most of false positives are caused by the inaccuracy of the code search technique used by ORIGEN. For example, Bindiff cannot yield good alignment results if source code are compiled from different compiler or different optimization level. We can further improve the accuracy of the binary alignment by leveraging more advanced techniques [6, 11, 15]. In this paper, we will discuss how to address the false positive issue in Section 6.4.

6.3 In-depth True/False Positive Analysis

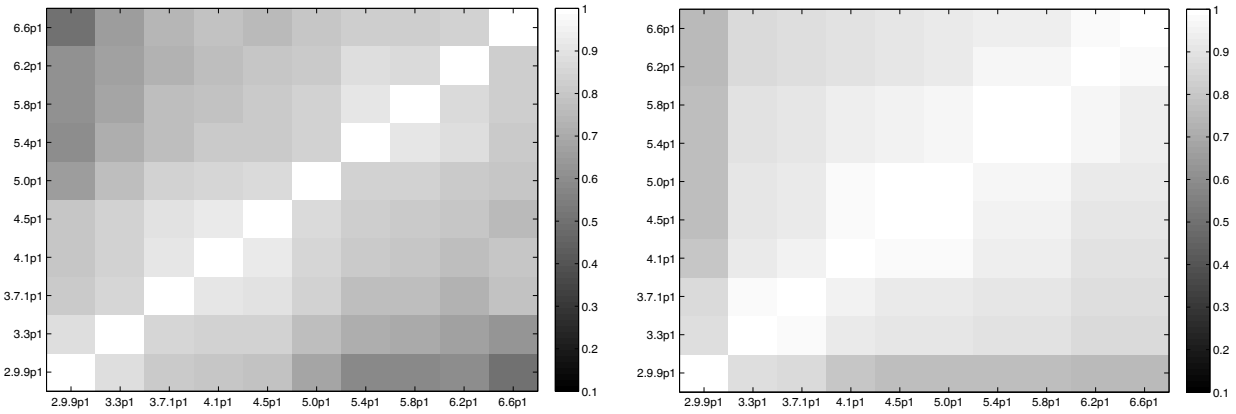
We also conduct an in-depth analysis to evaluate the accuracy of ORIGEN. Figure 7 presents detailed comparison results in the heat map. For the convenience of illustration, we only include 10 representative versions from 2.9.9p1 to 6.6p1, where each block indicates a pair-wise prediction experiment on the two versions. The brightness of the block in Figure 7(a) shows the offset prediction precision for 100 pair-wise profile generations; in Figure 7(b), the brightness indicates the true profile similarity for the 100 pairs.

We can see that ORIGEN exhibits better performance for adjacent versions, or in other words, it has the better performance when the time interval of two versions is smaller. For example, two adjacent versions of OpenSSH 3.3p1 and 4.5p1 have a very high offset prediction precision. This is reasonable, because two adjacent versions tend to have less differences in their binaries. In most cases, these differences in adjacent versions are from minor code changes such as security patches, so these two binaries still share most of similar codes. When the time interval of two versions is large enough, ORIGEN may not generate the profile with the good quality. In this case, we can either use the method in Section 6.4 or create a new base model on the more recent version. The new model creation is much less frequent than the version change. In fact, we only need to create 2 models for the 40 versions of OpenSSH.

The true data structure similarity in Figure 7(b) shows a good explanation about the performance of ORIGEN. Each true data structure similarity in this matrix is calculated by diffing true data structures of two versions. We can see that most of adjacent versions can reach 100% similarity. When the time interval increases, the drop of data structure similarities is marginal. This also demonstrates that adjacent versions have few design changes and look similar. This results substantiate our intuition that software of different versions tend to be similar.

6.4 Handling False Positives

The accuracy of ORIGEN has been verified in Section 6.2. The average precision is about 90%, but there are still 10% false positives, which might not be desirable in some mission critical applications. To this end, we incorporate a thresholding method to reduce the number of false positives. The idea is that we can adopt the number of accesses to quantify the searching robustness of the data field type, and only con-



(a) The offset prediction precision

(b) The true data structure similarity

Figure 7: The illustration of pair-wise experiments on 10 representative versions of `OpenSSH`.

sider the data field type above the threshold as the searching candidate. We admit it will sacrifice coverage for accuracy, but it is necessary for the practical integration in some cases.

The result as shown in Figure 8 illustrates the precision under different thresholds, where the x-axis lists the threshold, and the y-axis represents the precision. For each threshold, the 95% confidence interval of 40 versions is also plotted. As we see, the precision increases along with the threshold, and a bigger threshold leads to a more accurate result, e.g. the precision is 98.53% under the threshold 32. As the threshold determines the searching robustness of the data type, a method with a bigger threshold behaves more prudently, and makes less yet more accurate predictions. For example, when the threshold is 2, our method yields 116,446 predictions; but when the threshold is 16, it yields only 42,324 confident predictions. The experimental results substantiate the claim that our method can be tailored to produce very few false positives.

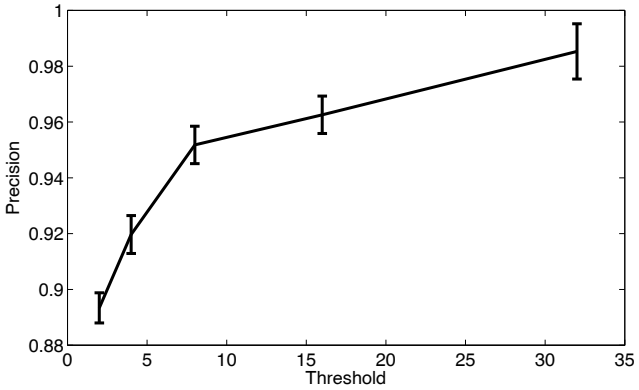


Figure 8: Precision of our method under different thresholds.

6.5 Case Studies

In this section, we conduct a qualitative analysis to evaluate the practice of ORIGEN. We select several key data fields in all of the software samples listed in 1 and conduct case studies in two application scenarios: virtual machine introspection and memory forensics.

For virtual machine introspection (VMI), we choose to enhance DECAF [18], the dynamic analysis platform. DECAF relies on VMI to retrieve the running processes and loaded modules inside a virtual machine to analyze the behaviors of specified processes or kernel modules, for automatic malware detection and analysis. However, it only supports a limited number of guest OS versions (including Windows and Linux), due to the hardcoded profiles. To support a new guest OS version, a user must compile and load a kernel module inside the virtual machine to generate the corresponding profile. We aim to demonstrate that with help of ORIGEN, we can eliminate this manual task by automatically generating the profile from a given virtual machine image within just a few minutes. This case study can demonstrate how ORIGEN greatly improves the usability of VMI for the cloud provider.

For memory forensics, we show two forensic analysis tasks: OpenSSH session key extraction, and dm_crypt[2] encryption key extraction. We develop two plugins on Volatility memory forensics framework [38] to accomplish these two tasks, respectively. We aim to demonstrate that with help of ORIGEN, we can perform these analysis tasks in a cross-version manner. It means that without knowing the version information of the application in a memory dump, we can automatically create a localized profile and then immediately perform the forensic analysis on the memory dump.

We select key data fields as a demo for each analysis. The second column in Table 2 lists key data fields of interest. To be more specific, for Windows VMI, we need the global variable `PsActiveProcessHead` as the starting point to traverse the linked list of `EPROCESS`, and then within each `EPROCESS` object, we obtain the process ID in `UniqueProcessID`, the name in `ProcessName`, and so on. We visit the next `EPROCESS` object through `ActiveProcessLinks`. Similarly for Linux VMI, we need to start from `init_task` to traverse the `task_struct` linked list and locate the process ID in `pid`, and the process name in `comm`, and so on.

In memory forensics scenario, for dm_crypt, we create a signature using the five fields in the structure `crypt_config` to scan the memory and find the actual encryption key in `crypt_config.key`.

We select three base versions for each software, as shown in Table 2. In order to evaluate the strength of ORIGEN,

[2]dm_crypt is a disk encryption tool in Linux.

Name	Field Name	ORI Statistic on Windows XPSP0			WinXPSP2	WinVista	Win7
		DL	SL	Total	D (TP/FP)	D (TP/FP)	D (TP/FP)
Windows	EPROCESS.UniqueProcessId	5	7	12	√(12/0)	√(9/3)	√(9/3)
	EPROCESS.EitTime	0	2	2	√(2/0)	√(2/0)	√(2/0)
	EPROCESS.ActiveProcessLinks	1	3	4	√(4/0)	√(4/0)	√(4/0)
	EPROCESS.ProcessName	0	4	4	√(4/0)	√(3/1)	√(3/1)
	EPROCESS.PEB	5	2	7	√(7/0)	√(4/3)	√(4/3)
	EPROCESS.DirectoryTableBase	2	1	3	√(3/0)	√(3/0)	√(3/0)
	.data : PsActiveProcessHead	0	3	3	√(3/0)	√(3/0)	√(3/0)
Linux		ORI Statistic on Linux3.5.0			Linux2.6.32	Linux3.8.0	Linux3.13.0
		DL	SL	Total	D (TP/FP)	D (TP/FP)	D (TP/FP)
	.data: init_task	0	10	10	√(8/2)	√(8/2)	√(8/2)
	task_struct.tgid	9	1	10	√(10/0)	√(8/2)	√(8/2)
	task_struct.pid	8	2	10	√(5/5)	√(8/2)	√(7/3)
	task_struct.comm	1	4	5	√(4/1)	√(5/0)	√(5/0)
	task_struct.tasks	1	2	3	√(3/0)	√(3/0)	√(3/0)
	task_struct.mm	42	5	47	√(29/18)	√(37/10)	√(37/10)
	mm_struct.pgd	12	4	16	√(12/4)	√(11/5)	√(11/5)
OpenSSH		ORI Statistic on OpenSSH5.9			OpenSSH5.3	OpenSSH6.0	OpenSSH6.5
		DL	SL	Total	D (TP/FP)	D (TP/FP)	D (TP/FP)
	.bss: active_state	1	5	6	√(6/0)	√(6/0)	√(6/0)
	session_state.ssh1_key	0	2	2	√(2/0)	√(2/0)	√(2/0)
	session_state.ssh1_key_length	0	4	4	√(4/0)	√(4/0)	√(4/0)
dm_crypt		ORI Signature Statistic on Linux3.8.0			Linux3.5.0	Linux3.11.0	Linux3.13.0
		DL	SL	Total	D (TP/FP)	D (TP/FP)	D (TP/FP)
	crypt_config.cpher	1	3	3	√(3/0)	√(3/0)	√(3/0)
	crypt_config.cipher_string	1	4	4	√(4/0)	√(4/0)	√(4/0)
	crypt_config.iv_size	1	8	9	√(9/0)	√(9/0)	√(9/0)
	crypt_config.key_size	1	4	5	√(5/0)	√(5/0)	√(5/0)
	crypt_config.key	1	3	4	√(4/0)	√(4/0)	√(4/0)

Table 2: The efficacy of ORIGEN on different applications. DL denotes the dynamic labeling; SL for static labeling. D for "Detected". TP for correctly matched ORIs in the new version and FP for wrongly matched ORIs for the new version

these test versions span several major revisions, ranging from `Windows XP`, `Linux 2.6.32`, and `OpenSSH 5.3`, to `Windows 7`, `Linux 3.13.0`, and `OpenSSH 6.5`.

ORIGEN can accurately generate a profile for each of the four analysis tasks, and the results are shown in Table 2. Table 2 lists the software family names to be tested, their base version and three test versions. For each software family, the ORI labeling and matched results are listed respectively. For ORI label, it shows the number of ORIs via the dynamic labeling (DL) and the static labeling (SL) respectively. The column of "Total" shows a sum of ORIs generated via two phases. For each test version, we also list the number of correctly labeled ORIs and missed ORIs respectively.

The results in Table 2 demonstrate three points. First, ORIGEN can precisely label ORIs in the base version for the data fields in each profile. We can see that each data field has more than one ORI in the base version. Second, the static labeling can improve the ORI coverage. By comparing the ORI number in DL column and total column, we see that the static ORI labeling can help find more ORIs. Finally, the error correction can help to reduce the false positive rate. We found that the profile localization for the four software families cannot find all semantically-equivalent ORIs for their test versions, but the error correction still helps to infer the accurate offset for each data type field in the generated profile. For example, there are 47 ORIs in total for the field `task_struct.mm` in the base version of Linux, `Linux 3.5.0`. However, ORIGEN only correctly finds 37 ORIs in `Linux 3.8.0`. By adopting the strategy one in discussed in Section 4.3, the correct offsets can still be found by filtering the false offset values from the false 10 ORIs.

The Demo of ORIGEN .

To the end, we show the dm_crypt key extraction result to demonstrate the effectiveness of ORIGEN shown in Figure 9. ORIGEN has not information about the version information for the test dm_crypt in the memory dump. It extracts the binary from the memory dump and automatically generates the concrete profile for fields in Table 2. Then it utilizes the concrete profile and successfully extracts the dm_crypt key.

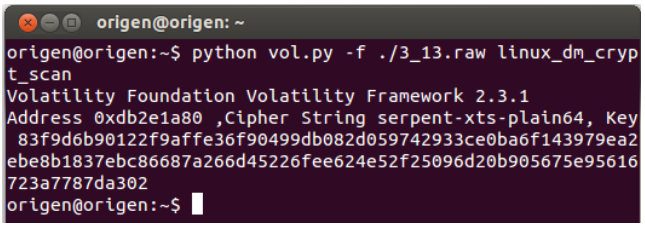


Figure 9: The demo result of dm_crypt version-independent memory analysis.

6.6 Runtime Performance

In this section, we verify the runtime performance of ORIGEN. Table 3 demonstrates the average running time of ORIGEN in Table 2. It includes the ORI labeling and the profile localization time.

We can see that it takes few seconds on average to finish the labeling for one ORI. Among steps of the ORI labeling, code disassembly takes up to 30 seconds for complex binary code like Linux kernel. The rest of steps such as the intra-procedural data-flow analysis only cause negligible runtime overhead. The profile localization takes several minutes to generate a profile. Most time is spent on the binary code alignment by BinDiff. It is reasonable, because conducting the alignment on the large scale binary is time consuming.

For VMI, ORIGEN takes around two minute to generate a profile for an unknown virtual machine image and then can immediately perform security monitoring from the hypervisor layer. This generation time could be greatly improved by conducting more efficient code search technique. Our goal is not to completely resolve this problem but provide a promising solution for cross-version memory analysis.

Family Name	Total Time	
	ORI Labeling	Profile Localization
Windows	59 sec	1.1 min
Linux	1.3 min	3.2 min
OpenSSH	39.3 sec	18.4 sec
dm_crypt	24 sec	10 sec

Table 3: The total time for each application on average.

7. DISCUSSION

In this section, we mainly discuss about the limitation and potential challenges of this work.

Code Syntactic Changes.

We leverage the code search techniques to conduct the binary alignment for the profile localization. It is possible that some syntactic changes modify the control flow graph for the new version of a binary, such as inline functions or code optimizations. This can reduce the code search accuracy of ORIGEN. We summarize possible syntactic changes and list the robustness of the code search technique used by ORIGEN to these changes in Table 4.

Fortunately, many related works have already focused on this issue and proposed more accurate search results [7]. The goal of the paper is to explore the feasibility of ORIGEN. In the future, we will work on how to improve the accuracy of the generated profile by ORIGEN.

Code Semantic Changes.

ORIGEN by design can only infer the offset value for data fields which have been trained in the older version. If the data type is newly added, ORIGEN cannot infer the offset value for it. During the software development, it is common to add the security patches or redesign the code in the new version. These patches or code reorganization could change the semantics of the older version. For example, the new version could add extra data types or remove some data fields. In these cases, ORIGEN will fail to generate the profile for these new coming data types. One possible way to sidestep this limitation is to train the additional model for the new version, and apply the new model to generate profiles for its similar versions.

Code Change	Strength
Register Assignment	Yes
Control Flow Flattening	Yes
Instruction Scheduling	Yes
Opcode Selection	Yes
Function Parameters	Yes
Function Inlining	Maybe
Calling Convention	Partial

Table 4: Robustness Analysis

8. RELATED WORK

Code Search in Binary and Its application.

The code search technique recently has attracted much attentions. Most previous work put their efforts on the performance improvement for searching semantic equivalent codes in code database [6, 10, 11, 15, 22–24, 27, 29, 33, 34, 36, 36]. Many researchers also applied these promising code search algorithms into different applications [5,19]. Bug search utilizes the search techniques to quickly identify the program bugs [33, 34]. Patch generation applies the code similarity techniques to the semantic code discovery. Program lineage exercises the code similarity methods to infer the evolutionary relationship among a collection of software. Software plagiarism and repackage discovery also adopts the code search techniques [20], and so on. This paper is the first attempt at the cross-version memory analysis by leveraging the code search techniques. The experiments also shows it is promising to apply the code search techniques for the across-version memory analysis.

Memory Forensics.

Several memory analysis tools [1,14,26,32,35,38] etc. have been proposed to aid the automatic memory forensics. They aim at analyzing and retrieve sensitive information from a memory dump. A key aspect of memory forensics is to encode the semantic related information into the data structure profile and follow the profile to conduct the specific analysis. The profile is predefined to the specific version of the image being analyzed, and update the profile according to versions of the target software.

State-of-the-art techniques rely on reverse engineering to reconstruct the profile of semantic information. The reverse engineering most often requires the manual effort or use non-trivial scripts [2] that operate on the source code. In this paper, we propose the idea of cross-version memory analysis. Instead of reverse engineering version by version, it transfers the knowledge from the trained model for the older version to generate the profile for the new version.

Virtual Machine Introspection (VMI).

VMI extracts semantic knowledge from a running virtual machine to monitor and inspect semantic behaviors of the guest machine. Due to the nature of isolation, VMI has been applied for many security applications. For example, many

intrusion detection applications utilize the VMI technique to conduct more accurate detections [16, 30, 31]. Some malware analysis approaches also relies on the VMI to capture the detail malware behaviors which cannot be captured by previous work [8, 21]. Furthermore, VMI techniques are also well used in memory forensics and process monitoring [17].

The main challenge in the VMI technique is to bridge the semantic gap between the guest OS and outside analysis tools. Many existing works have already made a great step on this problem [9, 13]. A recent tool, DECAF [18] performs VMI to retrieve key semantic information from a guest OS. In each of the above efforts, similar to memory forensics, non-trivial effort is required to construct a profile) of key semantic values and their concrete interpretations within the guest OS. Although VMST can reuse the OS code pieces of the introspection property to achieve the automatic VMI. However, the approach used in VMST could not be general enough to support the automatic introspection for some internal and close-sourced data structures.

Data Structure Reverse Engineering.

Reverse engineering data structures from binary executables is very valuable for many security problems. Particularly, Howard [37] and REWARDS [25] makes use of dynamic binary analysis to recover the types and data structure definitions from the execution of a binary program. For each instruction during the execution, they infer and propagate the types of the instruction operands. Certain memory access patterns also need to be recognized to discover specific data structures like arrays, linked lists, and embedded data structures. For most COTS binaries without well defined documentation about their function prototypes, Howard [37] and REWARDS [25] can only infer the primitive data types such as integer, string or pointers. The manual efforts are still required for higher semantic data type inference. In our paper, ORIGEN is proposed to alleviate the manual efforts. Instead of inferencing the data types for new version of a binary from the scratch, ORIGEN can utilize the knowledge from data types in the older version which has been analyzed to assist the profile generation for the new version.

9. CONCLUSION

In this paper, we presented the notion of "cross-version memory analysis". We detailed a solution and implemented a prototype called ORIGEN that is able to search the code in one binary, and locate the ORIs in another version of the code. The experimental results verified the efficacy of the proposed method. Specifically, our method successfully recovers the offsets for key semantic fields across different versions of OpenSSH, Windows, Linux, a loadable kernel module for Linux. In addition, it achieved a precision of 90% on 40 versions of OpenSSH. The experiments also demonstrated the efficiency of our method, where it took half a minute to identify all the chosen semantic fields on Windows and Linux respectively. Finally, we integrate ORIGEN into DECAF to demonstrate its effectiveness in VMI.

Acknowledgment

We would like to thank anonymous reviewers for their feedback. This research was supported in part by National Science Foundation Grant #1054605, Air Force Research Lab Grant #FA8750-15-2-0106, and DARPA CGC Grant #FA8750-14-C-0118. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

10. REFERENCES

[1] Insight-VMI, A semantic bridge for virtual machine introspection and forensic applications. https://code.google.com/p/insight-vmi/wiki/LinuxDebugSymbols.

[2] Linux memory forensics using Volatility – Prerequisites. https://code.google.com/p/volatility/wiki/LinuxMemoryForensics.

[3] E. Bursztein, M. Hamburg, J. Lagarenne, and D. Boneh. Openconflict: Preventing real time map hacks in online games. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy(Oakland'11)*, pages 506–520, 2011.

[4] F. Chagnon. IDA Decompiler. https://github.com/EiNSTeiN-/ida-decompiler.

[5] P. Comparetti, G. Salvaneschi, C. Kolbitsch, C. Kruegel, E. Kirda, and S. Zanero. Identifying dormant functionality in malware programs. In *Proceedings of 2010 IEEE Symposium on Security and Privacy(Oakland'10)*, pages 61–76. IEEE, 2010.

[6] Y. David and E. Yahav. Tracelet-based code search in executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI'14)*, volume 49, pages 349–360. ACM, 2014.

[7] Y. David and E. Yahav. Tracelet-based code search in executables. In *ACM SIGPLAN Notices*, volume 49, pages 349–360. ACM, 2014.

[8] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security(CCS'08)*, pages 51–62. ACM, 2008.

[9] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the IEEE Symposium on Security and Privacy(Oakland'11)*, pages 297–312. IEEE, 2011.

[10] T. Dullien and R. Rolles. Graph-based comparison of executable objects (english version). *SSTIC*, 5:1–3, 2005.

[11] M. Egele, M. Woo, P. Chapman, and D. Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 303–317. USENIX Association, Aug. 2014.

[12] 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(ACSAC'14)*, pages 196–205. ACM, 2014.

[13] Y. Fu and Z. Lin. Space traveling across vm: Automatically bridging the semantic-gap in virtual machine introspection via online kernel data redirection. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy(Oakland'12)*, pages 586–600. IEEE, May 2012.

[14] Y. Fu, Z. Lin, and D. 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)*, 2015.

[15] D. Gao, M. K. Reiter, and D. Song. Binhunt: Automatically finding semantic differences in binary programs. In *Information and Communications Security*, pages 238–255. Springer, 2008.

[16] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of Network and Distributed Systems Security Symposium(NDSS'03)*, volume 3, pages 191–206, 2003.

[17] B. Hay and K. Nance. Forensics examination of volatile system data using virtual introspection. *ACM SIGOPS Operating Systems Review*, 42(3):74–82, 2008.

[18] A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin. Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform. In *International Symposium on Software Testing and Analysis(ISSTA'14)*, pages 248–258. ACM, July 2014.

[19] X. Hu, T.-c. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security(CCS'09)*, pages 611–620. ACM, 2009.

[20] J. Jang, M. Woo, and D. Brumley. Towards automatic software lineage inference. In *Proceedings of the 22nd USENIX conference on Security(USENIX'13)*, pages 81–96. USENIX Association, 2013.

[21] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and Communications Security(CCS'07)*, pages 128–138. ACM, October 2007.

[22] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A search engine for binary code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 329–338. IEEE, 2013.

[23] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Recent Advances in Intrusion Detection*, pages 207–226. Springer, 2006.

[24] A. Lakhotia, M. D. Preda, and R. Giacobazzi. Fast location of similar code fragments using semantic juice. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, page 5. ACM, 2013.

[25] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium(NDSS'10)*, February 2010.

[26] MANDIANT Memoryze. http://www.mandiant.com/resources/download/memoryze.

[27] J. Ming, M. Pan, and D. Gao. ibinhunt: Binary hunting with interprocedural control flow. In *In International Conference on Information Security and Cryptology*, pages 92–109. Springer, 2013.

[28] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[29] B. H. Ng and A. Prakash. Expose: Discovering potential binary code re-use. In *Computer Software and Applications Conference (COMPSAC'13), 2013 IEEE 37th Annual*, pages 492–501, July 2013.

[30] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy(Oakland'08)*, pages 233–247. IEEE, 2008.

[31] B. D. Payne, M. De Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the Twenty-Third Annual Computer Security Applications Conference(ACSAC'07)*, pages 385–397. IEEE, 2007.

[32] N. L. Petroni, Jr., A. Walters, T. Fraser, and W. A. Arbaugh. FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation*, 3(4):197 – 210, 2006.

[33] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. In *IEEE Symposium on Security and Privacy (Oakland'15)*. IEEE, 2015.

[34] J. Pewny, F. S. C. Rossow, and T. Holz. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference(ACSAC'14)*, pages 406–415. ACM, 2014.

[35] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu. Vcr: App-agnostic recovery of photographic evidence from android device memory images. In *Proceedings of the 22th ACM SIGSAC Conference on Computer and Communications Security(CCS'15)*, pages 146–157. ACM, 2015.

[36] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages and applications*, volume 48, pages 391–406. ACM, 2013.

[37] A. Slowinska, T. Stancescu, and H. Bos. Howard: a dynamic excavator for reverse engineering data structures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, 2011.

[38] Volatility: Memory Forencis System. https://www.volatilesystems.com/default/volatility/.