IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 20, 2019, accepted September 2, 2019, date of publication September 5, 2019,
date of current version September 19, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2939566

ATOS: Adaptive Program Tracing With Online Control Flow Graph Support

HE SUN[1,2], CHAO ZHANG[2,3], HE LI[2,4], ZHENHUA WU[2,4], LIFA WU[5], AND YUN LI[2]

[1]Institute of Command and Control Engineering, Army Engineering University of PLA, Nanjing 210007, China
[2]Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China
[3]Beijing National Research Center for Information Science and Technology, Beijing 100084, China
[4]Information Engineering University of PLA Strategic Support Force, Zhengzhou 450002, China
[5]School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210003, China

Corresponding authors: Chao Zhang (chaoz@tsinghua.edu.cn) and Lifa Wu (wulifa@njupt.edu.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2017YFB0802900, in part by the NUPTSF under Grant NY219004, in part by the National Natural Science Foundation of China under Grant 61772308 and Grant U1736209, and in part by the BNRist Network and Software Security Research Program under Grant BNR2019TD01004 and Grant BNR2019RC01009.

ABSTRACT Program tracing solutions (i.e., tracers) can faithfully record runtime information about a program's execution and enable flexible and powerful offline analysis. Therefore, they have become fundamental techniques extensively utilized in software analysis applications. However, few tracers have paid attention to the size of traces and corresponding overheads introduced to offline analysis, as well as the Control Flow Graph (CFG) support. This paper presents ATOS, an efficient tracing solution, to address these issues. It adaptively adjusts the granularity of tracing while conservatively preserving the essential execution information. We implement a prototype of ATOS and evaluate it on several benchmarks. The results show that ATOS can greatly reduce the size of a trace and accelerate offline analysis, while preserving the execution states and supporting existing applications seamlessly. For example, using ATOS, the trace produced by the application CryptoHunt is reduced by 46 times, while the analysis time is reduced by 34 times.

INDEX TERMS Program tracing, control flow graph, loop optimization, adaptive granularity, check point.

I. INTRODUCTION

Program tracing involves recording the runtime information about a program's execution. The recorded trace information is useful for software analysis, especially for security analysis, such as malware analysis and vulnerability discovery. Therefore, program tracing becomes a fundamental building block of diversified applications [1]–[4], especially for offline mode approaches. Figure 1 shows a general workflow of applications that utilize tracing.

Despite its wide applications, tracing has two ***limitations*** that are not well discussed. First, in general, the traces recorded are very large, making offline trace analysis time-consuming and inefficient. For example, the average size of the trace log files produced by CryptoHunt [4] is over 2 GB in our test; the smallest one is about 750 MB, while the biggest one is over 9 GB. Moreover, CryptoHunt is reported to spend 43.3 minutes parsing and analyzing a single trace of an RSA-based application [4].

The associate editor coordinating the review of this manuscript and approving it for publication was Sedat Akleylek.

Second, existing tracing solutions do not provide CFG (Control Flow Graph) information during runtime logging and leave CFG recovery to the offline trace analysis phase, which in general inefficiently parses all instructions and potential paths along the trace to recover the CFG. As reported by X-Force [2], it needs about 8h on average for dynamically recovering the CFGs.

Note that, a CFG is essential for manual understanding of a program (e.g., reverse engineering) and automated program analysis (e.g., function identification). Missing CFG information in the trace, therefore, causes performance penalty for offline analysis. On the other hand, it also stumbles certain online applications.

However, few solutions have addressed these problems. A good tracing solution should (1) produce reasonably sized traces while retaining all relevant information and (2) efficiently build and maintain a CFG during runtime tracing.

Our Solution. To solve the above problems, we hereby present ATOS , an adaptive tracing solution with online CFG support. It selectively skips tracing instructions that can be efficiently recovered offline, rather than recording

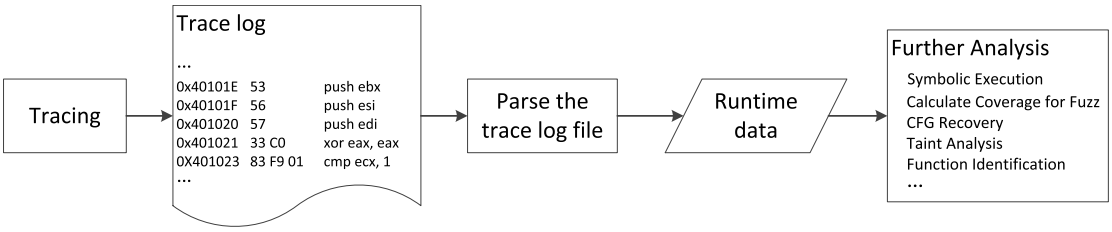


FIGURE 1. The general workflow of tracing-based applications.

instructions one by one, to reduce the size of a trace. Further, ATOS statically analyzes the target program, and dynamically tracks the code generated at runtime, thus building a CFG while tracing.

Tracing with Adaptively Adjusted Granularity. In many cases, it can be deduced whether some instructions will be executed and we can skip recording them one by one without losing information. For example, once the instruction list of a basic block is resolved, we can trace whether this block is executed (*block-level tracing*) rather than tracing each instruction within it (*instruction-level tracing*).[1] Similarly, there are two other tracing levels: *function-level* and *loop-level*. Adjusting different tracing granularity for different code snippets is a straightforward approach for reducing the size of a trace. Therefore, ATOS adopts a novel solution to adjust the tracing granularity adaptively by examining the tracing state and choosing granularity accordingly.

Loop Identification and Optimization. A loop, in general, is a repeated sequence of instructions. Loops are one of the root causes of the oversized traces. To reduce the size of a trace, ATOS performs *loop-level tracing* once the instruction list of a loop iteration has been determined. The challenge here is identifying loops during runtime tracing. ATOS first statically analyzes the target program to identify candidate loops (including nested loops) and then monitors runtime code modification and generation to recognize new loops. Moreover, we have refined the loop-level tracing process to handle complicated loops, that is, ones that have a very high cyclomatic complexity.

Online CFG Support. CFG support is essential for program analysis. ATOS provides online CFG support, not only for offline analysis but also for the online tracing process. For example, both the aforementioned adaptive tracing and loop identification rely on CFG. To enable such an analysis, ATOS tracks not only the control flow information, but also the type of each instruction (e.g., *call* instruction and indirect jump instruction), the tracing status of all basic blocks and functions, and loop structure information. This information is recorded in an augmented CFG, called the *shadow CFG*. Furthermore, applications that utilize just-in-time compilation (e.g., script engines) or those that tend to hide their inner logic (e.g., malware) often update their code and CFG at runtime. To handle this, ATOS also monitors dynamic code generation or modification during tracing and updates the CFG dynamically.

Results. We implement a prototype of ATOS based on the debugger shipped with IDA Pro[2] and evaluate ATOS on a wide range of applications. The results show that ATOS is able to efficiently trace these programs with sufficient information, and reduce the time and space overheads by order of magnitude, compared to state-of-the-art tracing approaches. For example, after replacing the default tracer with ATOS , the application CryptoHunt [4] yields a trace that is 46x shorter and reduces the analysis time by 34x.

In summary, we make the following contributions.

- We propose a novel tracing solution that can adaptively adjust the tracing granularity, able to selectively record instructions to reduce trace size while preserving sufficient runtime information.
- We present a practical and efficient loop identification method that is suitable for online tracing.
- We design a shadow CFG mechanism and implement an efficient CFG maintenance scheme that can support both online analysis and offline analysis.
- We implement a prototype of ATOS , which is able to trace programs efficiently and support existing applications seamlessly.

Organization. The rest of this paper is organized as follows. We present the related work in Section II and illustrate the overview design of ATOS in Section III. More details of ATOS , including tracing granularity, CFG management, and the adaptive tracing strategy are discussed in Sections IV, V, and VI, respectively. The evaluation results are presented in Section VII, followed by the discussion in Section VIII and conclusion in Section IX.

II. RELATED WORK

In this section, we will briefly review existing tracing methods and discuss different aspects of tracing, including the underlying tracing mechanisms, the tracing granularity provided by existing methods and post-tracing analysis methods.

A. TRACING MECHANISMS

There are five main types of tracing mechanisms: hardware-assisted tracing, static-instrumentation-based tracing, dynamic-instrumentation-based tracing, full-system-emulation-based tracing, and debugger-based tracing.

[1]This rule does not apply if the block has indirect call instructions or instructions that dynamically update code. Otherwise, certain information could be missed in the trace.

[2]https://www.hex-rays.com/products/ida/

Hardware-assisted tracing mechanisms record simplified control flow and timing information in an encoded data stream with hardware facilities, such as Intel Processor Trace (PT) [5] and ARM Embedded Trace Macrocell (ETM). They only introduce a very low overhead (e.g., less than 5% for kAFL [6]) in the tracing process, and tracing can be performed on any operating system as long as the hardware supports the mechanism. However, the tracing results should be decoded by static analysis before performing further analysis [5], and online mode analysis is difficult for hardware assisted approaches [7].

Binary instrumentation inserts additional code into an executable to observe or modify its behavior. There are two kinds of approaches for instrumentation: *static instrumentation* and *dynamic instrumentation*. Static-instrumentation-based-tracing mechanisms, such as Pebil [8] and MIL [9], rewrite the target program to insert tracing code. However, in dynamic-instrumentation-based-tracing mechanisms, such as Intel Pin [10], Dyninst [11], and Valgrind [12], the code for tracing is inserted at runtime. Static instrumentation is faster because the code is statically written in the binary, but it also suffers from the known drawbacks of static analysis [10], [13], for example, it cannot handle dynamically generated code, indirect branches or opaque predicates very well. Thus dynamic instrumentation is used more widely.

Full-system-emulation-based tracing is built upon a full-system emulator, such as QEMU[3]. It can access the whole state of the guest machine, and therefore can be used for full-system analysis. Various approaches have been proposed based on it, including online mode analysis platforms such as S2E [14] and the record-and-reply approach of Panda [15].

Debugger-based tracing [16] introduces the highest overhead compared to the other tracing mechanisms. However, debugger-based tracers can be easily ported to platforms that are not commonly used since debuggers are supported by most environments, and their design are always similar.

In summary, hardware-assisted and full-system-emulation-based approaches can perform system-wide tracing, while others can only trace target applications. Full-system-emulation-based tracing has a greater capability but is slower than hardware-assisted tracing. Static instrumentation is faster than dynamic instrumentation and debugger, but its capability is the weakest. The capabilities of dynamic instrumentation and debuggers are almost equal, while dynamic instrumentation is a little faster.

B. TRACING GRANULARITY

Generally, there are three types of traces with different levels of granularity: instruction-level tracing, basic-block-level tracing, and API-level (system-call-level, or function-level) tracing.

Instruction-level tracing records all executed instructions, as well as the execution state of each instruction. Most dynamic taint analysis solutions [17], [18] use instruction-level tracers.

Basic-block-level tracing records the information for each basic block executed. Different approaches have different ways of identifying basic blocks. For example, Intel Pin [10] dynamically identifies the borderboarder of each basic block at runtime, while IDA Debugger [19] identifies all basic blocks statically before executing them. Basic-block-level tracing is often used in fuzzing solutions to evaluate code coverage [20], [21].

API-level tracing records all function calls (API calls) during execution. It is one of the most important techniques for kernel analysis [22] and malware analysis [23].

Among these, instruction-level tracing is fine-grained, but requires the maximum overhead. The other two are coarse-grained methods, which produce smaller traces, but additional operations are required to recover the full trace. Therefore, an adaptive solution is required to adjust the granularity according to different cases. The similar insight has been used in approaches [24]–[26] and achieves excellent performance.

C. TRACE ANALYSIS

There are two main ways to analyze traces. Online analysis approaches perform the analysis during the tracing process, while offline analysis approaches perform it after the tracing process.

A classical application of tracing is dynamic symbolic execution (DSE) [1], which has both online and offline analysis modes. For example, S2E [14] is a typical online analysis DSE solution. It performs a selective symbolic execution that automatically makes decisions about whether to execute the program symbolically or concretely. Unlike S2E, SAGE [27] uses an offline analysis that analyzes the trace after the tracing finishes. Mayhem [28] proposes a hybrid execution that combines online and offline analyses. Either online or offline analysis is selected at each branch according to the system resources (memory usage).

As shown in Figure 1, offline mode analysis requires an extra overhead when saving and parsing the trace log file. However, offline analysis is more friendly to the tracers. It is obvious that a lighter tracer will make the tracing application more stable. Besides, for time-consuming approaches, such as taint analysis [29], an online mode analysis may interfere with the normal execution of the target programs, for example, by causing a network connection timeout.

On the other hand, an online mode analysis does not have the overheads of saving the trace log file and parsing traces. Furthermore, some tracing applications, such as selective symbolic execution [14], [30], have to manipulate the control flow [14], [30]. Online mode analysis is more suitable for these applications. In summary, for different applications, we must balance the overheads of tracing and analysis, as well as the application scenario.

[3]https://www.qemu.org

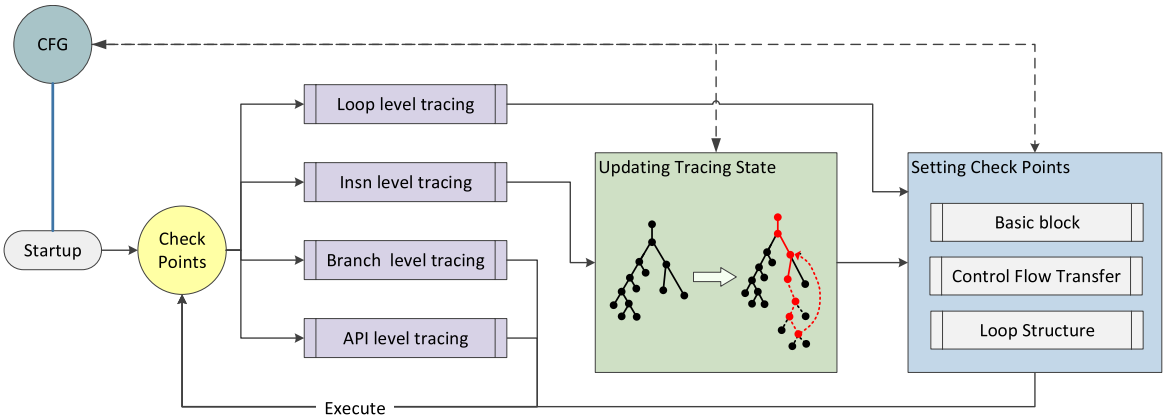


FIGURE 2. Illustration of the adaptive tracer ATOS 's workflow.

III. OVERVIEW

Figure 2 shows the workflow of our tracer ATOS . At its core, it adaptively selects the tracing granularity, allowing it to skip the recording of repeated and deterministic instructions, which, thus, reduces the size of the trace log without losing information.

To support adaptive granularity selection, ATOS instruments target programs with *check points* (e.g., an augmented form of breakpoints for debugger-based tracers[4]) to control the tracing granularity of code snippets, and dynamically updates these check points according to the tracing state.

The tracing state is recorded in the augmented control flow graph (denoted as *shadow CFG*), which is dynamically updated by ATOS during tracing, and used for making decisions about the tracing granularity.

Note that, ATOS outputs a structured trace instead of the plain text used in existing approaches [4], [31]. With CFG information the provided, we could efficiently recover the full trace for further analysis.

A. INSTRUMENTATION WITH CHECK POINTS

ATOS instruments target programs with *check points* to control how the tracing is performed, for example, where to trace and what level of granularity to use.

A check point is an augmented form of a breakpoint. It can be informally represented by a tuple of (*addr*, *code*, *attr*), where *addr* is the address of the instrumentation, *code* represents the tracing logic to perform when the check point is hit, and *attr* represents attributes associated with this check point, e.g., the type of the check points and static analysis results (e.g., loop details) that can be used for tracing.

ATOS introduces seven types of check points, as listed in Table 1: three types, three types of them support instruction-level, branch-level (discussed later) and API-level (or function-level) tracing; another three types, support loop identification and optimization (for loop-level tracing), and the final one monitors dynamic code modification or generation (for dynamic CFG updates).

[4]Check points are of different forms for different types of tracers. For example, for dynamic binary instrumentation based tracers, a check point is an augmented form of instrumentation callback function.

TABLE 1. Description of check points introduced by ATOS.

checkpoint	description
Insn-CP	Check point for instruction level tracing.
Branch-CP	Check point for branch level tracing.
Func-CP	Check point for API level tracing.
WE-CP	Check point for monitoring write on executable memory.
Counter-CP	Check point for counting the loop iterations.
OutLoop-CP	Check point for detecting exits of loops.
BreakLoop-CP	Check point for detecting breaks of loops.

Given a check point *CP (addr, code, attr)*, the program being traced will suspend at *addr* during execution, and execute the instrumented *code* to conduct the tracing depending on the attributes recorded in *attr*. After a check point is hit and executed, the target program will resume its execution.

B. BOOTSTRAP

To bootstrap the tracing, initial check points should be set and essential support data should be retrieved during startup.

First, ATOS applies classic static analysis [19] to obtain the control flow graph of the target program. It also initializes the instruction type, tracing status and loop information respectively, and stores them together in the shadow CFG.

Then, ATOS identifies the first function that will be instrumented with check points. In general, the function at the entry point of the program will be used as the first function. However, a program usually has some bootstrap code, which is not of interest for offline analysis, before its `main` function. To further reduce the size of the trace, ATOS tries to identify the `main` function, for example, via solutions such as F.L.I.R.T [32], and uses it as the first function to instrument.

Note that each instruction has to be traced when it is executed for the first time. In other words, instruction-level tracing has to be applied to instructions at first. Therefore,

ATOS instruments each instruction in the first function with the *Insn-CP* check point during startup.

Note that programs could dynamically modify or generate code, which may change existing instructions that have already been traced , as well as the control flow graph. To preserve the execution information, ATOS instruments recognized code regions with *WE-CP* check points. Once the executable memory region is being written to, the *WE-CP* check point will be hit and the tracer will perform instruction-level tracing on the overwritten (or generated) instructions again and update the CFG accordingly.

C. TRACING GRANULARITY

To reduce the trace size, the tracer must selectively record the executed instructions. ATOS provides four levels of tracing granularity.

All instructions, including those dynamically modified or generated code, are required to be traced when they are executed for the first time. Therefore, ATOS provides the basic *instruction-level* tracing.

With the CFG support, a tracer can skip recording instructions in a basic block, if all its instructions have been resolved and recorded before, and record only whether the block is executed. This is called *block-level* tracing, and is often used in tracers. Furthermore, note that if a basic block has only one successor, then the successor will always be executed if this block is executed. Thus, such successors can be merged into their predecessor basic blocks. Then, for a basic block with multiple successors, the tracer will record the basic block together with its branch conditions (which can indicate the successor). This is the block level tracing solution provided by ATOS , which is also called *branch-level* tracing.

In certain cases, the details of some functions (e.g., library functions) are not of interest for further analysis. Therefore, ATOS provides an *API-level* tracing, which skips API details during recording.

Loops can also result in oversized traces. ATOS uses loop-level tracing, which folds the execution of a loop structure by skipping the repeatedly executed code. tracing, which folds the execution of the loop structure by skipping the repeatedly executed code.

D. CFG MAINTENANCE

CFG is fundamental for most program analysis applications. However, due to the open challenges of static analysis [13], [33], statically-computed CFGs are often incomplete (refer to the left-hand CFG in Figure 2). Thus, it is necessary to update the CFG with runtime information.) is incomplete.

At runtime, the target program may self-modify its code and generate a new edge (e.g., the red dotted line in the right-hand CFG), which is unknown to static analysis. Furthermore, the new code may introduce a backward edge pointing to the existing code, forming a loop which is also unknown to static analysis neither.

Therefore, ATOS provides runtime CFG maintenance support, which updates the CFG and loop structure information according to the runtime tracing states.

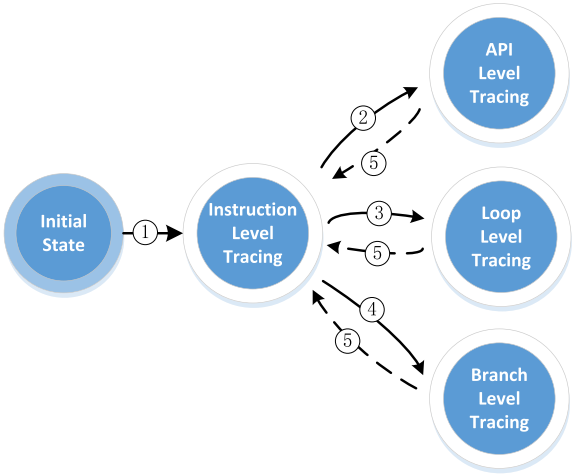


FIGURE 3. The tracing granularity transition state machine.

E. ADAPTIVE TRACING STRATEGY

Code snippets (e.g., instructions, basic blocks, and loops) of different tracing states should have different tracing granularity. ATOS adopts a novel solution to adaptively adjust the tracing granularity of code snippets. The granularity transition state machine is illustrated in Figure 3.

By default, all instructions will be recorded one-by-one when they are executed for the first time; that is, *Insn-CP* check points will be hit (line ① in Figure3). If the conditions for skipping certain code snippets are met, that is, *Branch-CP*, *Func-CP* or loop-based check points are hit (line ② ③ ④ respectively), ATOS will perform the coarse-grained tracing accordingly. If a *WE-CP* check point is hit (line ⑤), that is, some code is dynamically generated or modified, ATOS will switch back to instruction-level tracing to record necessary execution information.

In the following sections, we will present the detailed design and implementation of tracing granularity, CFG maintenance and the tracing strategy in ATOS .

IV. TRACING GRANULARITY

In this section, we will describe the four levels of tracing granularity supported by ATOS , including two main aspects: (1) what information is recorded; and (2) how to recover the trace with a succinct record.

Instruction-level tracing is the basic tracing method, which records information about each instruction executed. When an *Insn-CP* is hit, ATOS records the current stack, registers, referenced memory values, etc. Afterward, ATOS compares the current code and control flow to those recorded in the shadow CFG, and updates the shadow CFG if they are inconsistent. Furthermore, ATOS sets new check points on the code according to the tracing state to enable coarse-grained tracing. There are more details of the CFG maintenance and checkpoints given in Sections V and VI.

Branch-level tracing is used for coarse-grained tracing. We propose two rules to filter the basic blocks that should

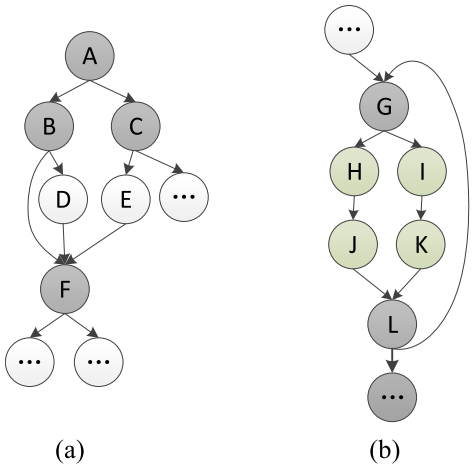


FIGURE 4. Branch level tracing.

be recorded for tracing, and introduce the way to recover the full trace with limited information. First, *Branch-CP* check points are set only on the traced basic blocks that have at least two successors. Figure 4 shows an example of branch-level tracing. Initially, all basic blocks are not traced and an *Insn-CP* is set on each instruction. When all instructions in the basic block have been traced, ATOS removes the *Insn-CP* from them. Then *Branch-CP* are set on the gray nodes *A, B, C, F* (Figure 4 (a)), since they have at least two successors.

Second, all basic blocks that are within the loop structures should be instrumented with *Branch-CP*. Otherwise, we would not be able to distinguish between two different iterations of the same loop. For example, in Figure 4 (b), if no check points are set on blocks *H, I, J, K*, then ATOS cannot recover the details of any iteration of a loop from *G* to *L*.

When triggering a *Branch-CP*, more information is recorded in addition to the current execution state, to help facilitate later recovery of the entire trace. The candidate paths between two blocks with *Branch-CP* are recorded in the successor block's tracing state. For example, paths *(B, F)*, *(B, D, F)*, and *(C, E, F)* are recorded in `F`'s tracing state. In this way, ATOS can recover the sub-trace between two blocks, by simply requesting the starting block's branch condition and the ending block's tracing state.

Moreover, the path information is recorded for basic blocks that do not have any successor. These basic blocks end with an indirect jump, *ret* instruction or call instruction to function *exit*. ATOS does not remove the *Insn-CPs* from them, and records the last instruction during branch-level tracing, so that the trace can be recovered with the path information.

API-level tracing is used for tracing calls to library functions. ATOS considers only functions statically linked or dynamically exported by system-provided libraries as API functions, since they are not of interest for most applications. ATOS sets a *Func-CP* on each call target. When a *Func-CP* is hit, ATOS checks whether the execution has reached the first instruction of an API function. If it has, ATOS performs API-level tracing to trace this function; that is, only the function name, arguments, and return value are recorded.

Loop-level tracing is used to fold the loop bodies in the trace. In the static analysis phase, the information about loop structures is recorded (Section V-C). The loop structures are identified at runtime, and the execution of a loop body is folded dynamically. The number of basic blocks executed, as well as the whole loop body, is recorded. With this information, ATOS is able to recover the entire trace, even if the loop structure has been folded.

To identify loops and optimize loop tracing, ATOS provides multiple types of check point. More details are given introduced in Section V-C1 and Section VI-B.

Dynamic code. ATOS also provides a mechanism to detect the use of the dynamic code [34], which generates new code or modifies existing code at runtime.

The new code can be discovered and added to the shadow CFG (Section IV). However, it works only if the code is instrumented with instruction-level tracing. To enable tracing the dynamic code in coarse-grained tracing, ATOS provides a new type of check point, *WE-CP*, to monitor the newly generated code.

A *WE-CP* is set on all identified code regions. When an executable code region is being written to, the *WE-CP* check point will be hit. ATOS then obtains the address that is written by parsing the instruction that is just executed. Then, instruction-level tracing is enabled so that the newly generated code can be added into the shadow CFG.

V. CFG MAINTENANCE

The shadow CFG may be inconsistent with the runtime program state due to the incompleteness of the static analysis. Thus, ATOS proposes an efficient CFG maintenance scheme, by applying which ATOS can identify the code and control flow transfers, which are not recorded in the shadow CFG. Edges in the shadow CFG are updated and obfuscated or smashed CFGs are merged.

Moreover, to set the check points for coarse-grained tracing, ATOS identifies the tracing state based on the CFG and the execution information. Four types of tracing states are identified. They indicate which specific code snippets have been executed, including basic blocks, loop structures, and control flow transfers, etc. In later phases, ATOS sets check points according to the tracing state, and enables the adaptive tracing strategy (Section VI).

A. CFG MAINTENANCE SCHEME

ATOS maintains the CFG by locating the current instruction in the shadow CFG. The main workflow is as follows.

A pointer is moving accordingly to the execution. If the current control flow transfer is identified by static analysis, ATOS can obtain potential instructions that will be executed by querying the successors of the previous instruction (which the pointer is pointing to) in the shadow CFG. This operation can be performed in O(1) time, since information about the successors is already stored in the shadow CFG.

The address and byte code are both considered to check whether the recorded instruction is the same as the code executed. The static obtained control flow is correct in most cases, so most control flow transfers can be identified by querying the pointer.

For the self-modified code [3], junk code [34], and indirect control flow transfers [35], static analysis results are usually not correct enough. The code and control flow transfers that are not recorded in the CFG are called *new code* and *new edge* or *new control flow transfer*. To locate new code or a new edge, ATOS also stores each instruction as a tuple (address, byte code, pointer). Given an unrecorded instruction, ATOS first indexes it by successively querying the address and the byte code. If a record is found, the pointer is updated to the recorded one, and the new edge is recorded as well; otherwise, it indicates that new code has been executed, static analysis is performed to obtain the CFG for it.

1) IDENTIFYING NEW CFG EDGES

New CFG edges are identified as follows.

If the current instruction is inside a basic block, the next instruction can be indexed in the basic block. If the executed instruction is not the one in the basic block right after the current instruction, a new CFG edge should be added and treated as a jmp or cJmp instruction. There are several cases for jmp and cJmp instructions.

- If the instruction jumps to the head of a library function, it is processed as a call instruction.
- If it jumps to an existing block in the current function, ATOS first checks whether this control flow transfer is already recorded in the shadow CFG. If not, a new edge will be added.
- If it jumps inside a function, ATOS will add an edge and merge the CFG.
- If it jumps inside a function but the boundary is recognized incorrectly, i.e., the target is on the chunk, then a new function will be created and static analysis will be performed for it. And the jmp/cJmp process is repeated.
- If the jump target is not even in the function area and does not belong to any other function, ATOS creates a function at the address and checks whether the jump target is a standard function header. If so, the instruction is processed as a call; otherwise, the CFG of the new function is merged into the original one.

A call instruction is treated as a jump if the called address is not a normal function or within a known function. If the call target is the header of a known function, but the static analysis fails to recognize the call, ATOS will add the CFG edge. A call to an unknown area will lead to static analysis at the target address, and the call process routine will be re-executed.

For ret instructions, if the return address is inconsistent with the statically recognized return address, ATOS will treat the instruction as *jump* since no representation of the current call stack can be found.

In addition, an exception handler can also lead to control flow transfers. The CFG of an exception handler is also merged.

Merging CFGs. Suppose there is a control flow transfer from basic block a to b, which belong to CFGs A and B, respectively. The sub-CFG of B rooted by b is marked as C. ATOS merges C into A by calculating the address intersection of A and C. If the intersection is a null set, ATOS will directly add C into A; otherwise, a new function will be created by merging C and the sub-CFG of A that does not overlap C.

B. IDENTIFYING NEW CODE

If the byte code in the given address is not the same as that recorded in the shadow CFG, or there is no code available at the target address in the shadow CFG, this indicates that new code has been generated. ATOS statically obtains the sub-CFG of the new code and then adds the edge. The sub-CFG is added into the shadow CFG, so that the following newly generated code will not be identified as new code.

C. IDENTIFYING TRACING STATES

After maintaining the CFG, ATOS identifies the current tracing state

Table 2 shows the defined tracing states. Initially, all code is marked as *NormalTracing*. During the tracing, when a specific code snippet is traced, the tracing state saved in the check point is updated correspondingly. After further operations (e.g., setting check points; See Section VI), the tracing state will be recorded in the shadow CFG and shift back to *NormalTracing*.

When the previous instruction is the last one of a basic block, the tracing state is updated to *BBTraced*. If a control flow transfer is executed (such as jump, call, or exception handler), the tracing state is set to *CFTTraced*. Section V-A introduced different cases of control flow transfers, and modification of the shadow CFG is recorded as a sub-tracing state of *CFTTraced*. The details are shown in Table 2.

1) UPDATING TRACING STATE OF LOOP STRUCTURE

LoopTraced is set when all basic blocks in the loop body have been traced. To achieve this, ATOS should be able to identify the loop structure, and be aware that the loop body has been executed during the tracing, which indicates that the loop body may execute repeatedly soon.

Here, we clarify the ***definition of a loop*** used in this paper. ATOS inherits the same definition as CryptoHunt [4]. A *loop* is a sequence of instructions that meets one of the following requirements. 1) The sequence of instructions repeats at least one time. 2) The instruction sequence ends with a conditional or unconditional jump instruction to the beginning of the instruction sequence. CryptoHunt comes up with an offline method to identify loop structure and nested loop. Unlike CryptoHunt, we focus on two forms of loop structure that can be identified by static analysis: simple cycles and the loop of strongly connected components (SCC loop). A simple cycle does not visit any vertex more than once [36], [37]

TABLE 2. Description of tracing states.

tracing state		description
NormalTracing		No other operations should be performed after maintaining the tracing state.
BBTraced		The basic block has been just traced.
LoopTraced		Loop body has been traced at least once.
CFTTraced	NoCFGChange	The current control flow transfer has recorded in the shadow CFG, so the shadow CFG is not changed.
	IntraCFGChange	The memory of the function is not re-written, but the current CFG is not the same as the shadow CFG. This is often caused by static analysis errors.
	IntraCFGAddEdge	The control flow transfer is not recorded in the shadow CFG, but both nodes of the control flow transfer are recorded in the shadow CFG. This is often caused by indirect jump whose target is within the current function.
	InterCFGAdd	The CFG has been merged by sub-CFG of another function.
	APICalled	The control flow transfer target is the first one of a library function.
	FunctionCalled	The control flow transfer target is the first one of function. The function is not a library function and has not been traced before.
	FunctionReCalled	The control flow transfer target is the first one of function. The function is not a library function and has been traced before.

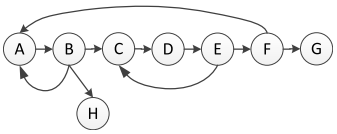


FIGURE 5. Loop struture.

and an SCC loop of a directed graph is a maximal strongly connected subgraph in which strongly connected means that there is a path between all pairs of vertices of the directed graph [38]. As shown in Figure 5, *A, B, C, D, E, F* is an SCC loop; while *A, B* and *C, D, E* are two simple cycles.

Second, ***loop structures are identified*** in initialization and when ATOS finds new code or a control flow transfer. Simple cycles (loop structures) and SCC loops are identified by Johnson's algorithm [37] and Kosaraju's algorithm [38].

Suppose b and s are basic blocks in a CFG, C is the basic block sequence of the cycle or SCC loop, and $succ(b)$ are the successors of basic block b in the CFG. If b satisfies (1), which means that at least one successor of b is not in the cycle (or SCC loop), b will be recorded for later use.

$$b \in C \wedge \exists s \, (s \in succ \, (b) \wedge s \notin C) \tag{1}$$

Once the loop structures are identified, OutLoop check points are set on any basic blocks that may jump out of the cycle. In Figure 5, B is a basic block that may jump out of cycle *A, B*, an E is a basic block that may jump out of cycle *C, D, E*.

Third, in the following tracing, if an OutLoop check point is hit, ATOS checks whether all the code in the corresponding simple cycle has been traced successively (ignoring the function calls in them) before the check point. If so, this indicates that the loop structure has been traced; ***LoopTraced*** **state is set and loop level tracing is enabled.**

VI. TRACING STRATEGY

So far, we have discussed tracing granularity and the CFG maintenance scheme. In this section, we will introduce how they are associated for adaptive tracing. Generally, check points are set according to the tracing state, and when check points are hit, tracing is performed for the corresponding granularity.

When the tracing state is *NormalTracing*, no check points are set and the execution resumes.

When the tracing state is *BBTraced*, it indicates that the basic block has just been traced. ATOS removes the *Insn-CP* in the basic block, and sets *Branch-CP* on the last instruction of a basic block with more than one successors. Note that since ATOS must confirm all the control flow has been recorded in the shadow CFG, an *Insn-CP* is always set for control flow transfer instructions, such as *call*, *ind-jmp* and *ret* instructions.

A. TRACING STRATEGY FOR CONTROL FLOW TRANSFER

When the tracing state is *CFTTraced*, ATOS deals with all possible cases mentioned in Section V-A.1. Generally, the check points can be classified into two categories, inter-procedure and intra-procedure.

NoCFGChange, IntraCFGChange, IntraCFGAddEdge, and InterCFGAdd are dealt with in the intra-procedure phase. If the tracing state is *NoCFGChange*, no check point will be set. If the tracing state is *IntraCFGChange, InterCFGAdd* or *IntraCFGAddEdge*, static analysis will be performed, and the loop structures are identified. And *OutLoop-CPs* will be set. If the tracing state is *IntraCFGChange* or *IntraCFGAddEdge*, an *Insn-CP* will be set on each code in the function. If the tracing state is *InterCFGAdd* and there is no check point set on the merged sub-CFG, an *Insn-CP* will be set on each code in the sub-CFG.

APICalled, FunctionCalled, and FunctionReCalled are dealt with in the inter-procedure phase. If the tracing state

is *APICalled*, an *Insn-CP* will be set on the return address of the current function. If the tracing state is *FunctionCalled*, static analysis will be performed and *OutLoop-CPs* will be set. Moreover *Insn-CPs* will be set on all the code in the function. If the tracing state is *FunctionReCalled*, no new check points will be set since the check points have been set on the function.

B. TRACING STRATEGY FOR LOOP STRUCTURE

When the tracing state is *LoopTraced*, loop level tracing can be enabled. ATOS sets check points as follows.

First, *BreakLoop-CPs* are set on the basic blocks that satisfy (1) in Section V-C1. For example, in Figure 5, *C, H* are selected for the simple cycle *A, B*, and *F* is selected for the simple cycle *C, D, E*.

Then, ATOS removes all branch check points between the start and end instructions in each identified loop structure, which is recorded. Besides, a *LoopCount-CP* is set on an instruction in the loop structure. The instruction is selected if it belongs to only one loop structure.

Afterward, ATOS resumes tracing. If a *BreakLoop-CP* is hit, ATOS again adds the removed check points and records the number of the loop body executed from the *Counter-CP*. If other check points are hit, then more code has been executed in the called function in the loop body. ATOS picks the current basic block, and obtains its predecessors. A predecessor without *Branch-CP* and *Insn-CP* is recorded, which is used for recovering the trace in the later phase.

Dealing with functions with large cyclomatic complexity. When the cyclomatic complexity is large (40 based on our observations), there will be a large number of simple circles in the CFG. Moreover, the overhead for identifying loops will be unacceptable since there are too many simple cycles to be matched. We propose a simple workaround in which is a loosen solution. ATOS randomly drops some of the statically found simple cycles. As the cyclomatic complexity increases, more of the simple cycles are dropped randomly. Branch level tracing is used for the dropped simple cycles.

Dealing with SCCs. If the instruction (marked as *ins*) with a BreakLoop-CP and the traced simple cycle are in the same SCC loop, ATOS searches the instructions that satisfy (1) of the SCC loop and sets *BreakLoop-CP* on them. Then ATOS has two modes for dealing with this. The first one is to continue tracing the program. The code in the SCC loop is traced at branch level. Note that since the *OutLoop-CPs* are still set on the code, the loop structures within the SCC loop will be continually identified. The trace obtained can be recovered in this mode.

The second mode is to simply skip the details of the loop structures. ATOS removes the check points of all the code in the SCC that is not with a *Insn-CP* on it. Thus, only the untraced code can be hit, until the execution leaves the SCC loop (e.g., a *BreakLoop-CP* is hit). At this time, the trace can not be recovered any more. However, it is sufficient for applications that do not need a full trace, such as CFG recovery, fuzzing, and most malware analysis approaches.

Recover the trace. The recovery of a branch level trace is as discussed in Section IV.

ATOS has recorded enough information during loop-level tracing, including the execution number of the loop structures, the instruction with the *BreakLoop-CP* that was hit, the instructions with *OutLoop-CP*, and the details of the called functions that execute different paths. With the maintained CFG and the trace of the first execution of the loop structure, the skipped code from the simple cycles can be recovered by counting the number of loops and filling them with the loop body obtained by static analysis.

Take the loop *A, B* in the CFG in Figure 5 as an example. There are only two basic blocks *A, B* in the trace for the loop. Suppose the execution reaches the basic block *C*, and the number of the repeated executions of *A, B* recorded in *Counter-CP* is 3. Then the full trace is *A, B, A, B, A, B, C*.

If ATOS runs in the second mode to deal with the SCC loop, the skipped code in the SCC loop cannot be recovered. ATOS provides a configuration for the cases where complex loop structures are important.

VII. EVALUATION

We implemented the prototype of ATOS based on IDA built-in dissembler, debugger and IDAPython, which is the IDA Pro Python language programming interface [19]. ATOS supports all architectures that IDA supports. We use IDA Pro v6.95 because of copyright issues. VirtualBox[5] with Windows 7 OS is used for executing the samples, with 8 GB of memory and two cores. The project has 4000 lines of code in total.

In the following of this section, we will first introduce the evaluation setup, and then discuss the details and results of our evaluation. The following questions are answered aiming to borne out the completeness and effectiveness of ATOS .

- Is the trace complete, especially for code that is traced under branch level and loop level?
- Is ATOS efficient enough for tracing purpose?
- Can ATOS help to reduce the whole analysis time compared to the existing approaches?
- Is the online maintained CFG helpful for real applications?

A. EVALUATION SETUP

We chose various programs and open source or accessible approaches as the baseline approaches.

Dataset. There are various kinds of samples in the dataset, including OpenSSL test programs, SPEC CPU2000 benchmarks, known programs packed by various off-the-shelf packing tools, programs of different algorithms written in C programming language, and real APT (Advanced Persistent Threat) samples from open malware collections. We randomly pick several samples for each evaluation. Relevant data are fed to the programs that needed inputs. For example, we feed a small but valid piece of C programming

[5]https://www.virtualbox.org/

TABLE 3. Trace length of different approaches. Columns 2-5 are the trace length of different approaches. The last column is whether ATOS 's recovered trace is the same as the baseline result, in which 'Y' means yes and 'N' means no.

	ATOS	ATOS recovery	baseline	complete
BubbleSort	2871	9445	9445	Y
HeapSort	1036	4540	4540	Y
MergeSort	219	4529	4529	Y
QuickSort	625	2583	2583	Y
BinarySearch	34	1700	1700	Y
linearSearch	8	204	204	Y

language source code with 143 LoC for *cc1* from SPEC CPU2000 benchmarks, and an array of 800 numbers for *bubble sort* and *binary search*. The OpenSSL library functions are statically linked in the programs.

Baseline Approaches. We compare ATOS with several approaches. CrytpoHunt [4] is a state-of-art Pin-based approach for detecting cryptographic functions. It has very clear boundaries for tracing, parsing trace log file, and performing further analysis. Besides, the pin tracer is also used in VMHunt [31]. The IDA tracer is published as a default module of IDA Pro.[6] Moreover, we compare ATOS with DPI [3] to evaluate the efficiency of packer analysis.

B. COMPLETENESS

Q1: Is the trace result of ATOS complete?

Since ATOS outputs a trace with different levels of granularity, we need to verify the completeness of the trace. The intuition behind this evaluation is that if the traces output by ATOS and a baseline approach are the same, then ATOS gives a complete result.

Approach. The baseline tracer was that of CryptoHunt. To minimize possible randomness in the programs, we choose six programs that implement several algorithms written in C language. Only the code in the *main* function[7] is evaluated.

The first step is tracing and recovering the trace of ATOS . The details of how to recover the trace under branch level and loop level are introduced in Sections IV and Sections VI-B, respectively. Then, we use the tracer of CryptoHunt to trace the programs. At last, we compare the traces to verify the completeness of ATOS .

Results. Table 3 shows the results of this evaluation. The baseline trace length obtained by the tracer of CryptoHunt is shown in the second column. The results of ATOS are in the third column. And the forth column shows the recovered trace length of ATOS . Note that as both CryptoHunt and ATOS records the trace of the full program, we manually pick out the corresponding trace for the code of the *main* function. The last column shows that the results for ATOS outputs are the same results as those for the CryptoHunt tracer.

[6]This evaluation is also tested on IDA Pro v6.95. We can see from the update logs that higher versions do not have much promotion in the default tracer, so the evaluation is acceptable.

[7]The code in *main* function performs the algorithms. Besides, since they are basic algorithms, we are sure that there is no randomness in the code.

Due to the API-level tracing, ATOS 's results do not include the execution path of system provided library functions. But the loss is reasonable, because the details of the known library functions are not useful in most cases. For example, since user-defined code causes most crashes, most fuzzing approaches (such as *PTFuzz* [39], *redqueen* [40], *CollAFL* [41], etc.) do not pay any attention to library functions. Malware analysis approaches (such as X-Force [2] and BE-PUM [42]) also ignore the details of the library function either. They pay more attention on which library functions are called and what their arguments are, which are already recorded.

Conclusion. In summary, the traces under coarse-grained tracing are recoverable and the recovered traces are complete.

C. PERFORMANCE

Q2: How about the performance of ATOS ? How much space is spent for the trace log file? How much time is spent in tracing and parsing the trace log file?

We evaluate how ATOS performs compared to existing solutions, including the time and space overheads. Generally, the time overhead includes the overheads for tracing and parsing the trace log file. The size of the trace log file is used as the measure of the space used. For ATOS , the time overhead includes the time spent of static analysis, tracing, and parsing trace log file. For the other approaches evaluated, the time overhead includes the time spent of tracing and parsing trace log file.

Approach. We evaluate ATOS , IDA tracer, and CryptoHunt. The parser of CryptoHunt is used as the baseline in comparing the time spent on parsing different trace log files. IDA tracer can generate trace files, but no default parser is provided. We implement one just like CryptoHunt. I

We separately recorded the time and storage spent by each approach. The time spent was normalized with the native execution time for the sample.

Space overhead results. The y-axis of Figure 6 shows sizes of the trace log files for the three approaches using a logarithmic scale for clarity. The results vary considerably: *srptest* has the largest trace log file at over 9 GB. The trace produced by IDA tracer for *mcf* is the smallest at only 24 kB.

In most cases, ATOS uses the least amount of space , though it contains both trace information and CFG information. CryptoHunt outputs very large trace log files, because it records all the instructions, even those before the entry point of the program and those after the program has exited. Specifically, the average size of the trace log files produced by ATOS is 46 times smaller than the average for CryptoHunt

Note that IDA tracer skips all the library functions, so the space overhead is smaller than that for CrytoHunt. For the sample *mcf*, because we feed it with infeasible input, the program exits soon after its *main* function starts. Therefore, the trace result for IDA tracer is very short.

For the packed samples, the space used by IDA tracer is a little smaller than that used by of ATOS . This is because

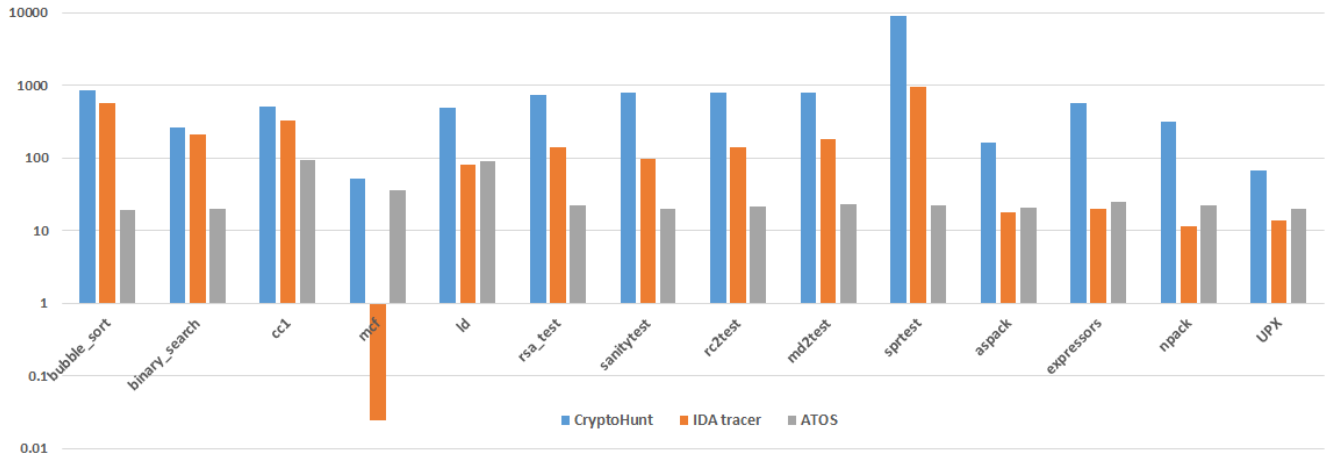


FIGURE 6. Space overhead of different approaches. To show the results more clearly, y axis is normalized using the base 10 logarithm.

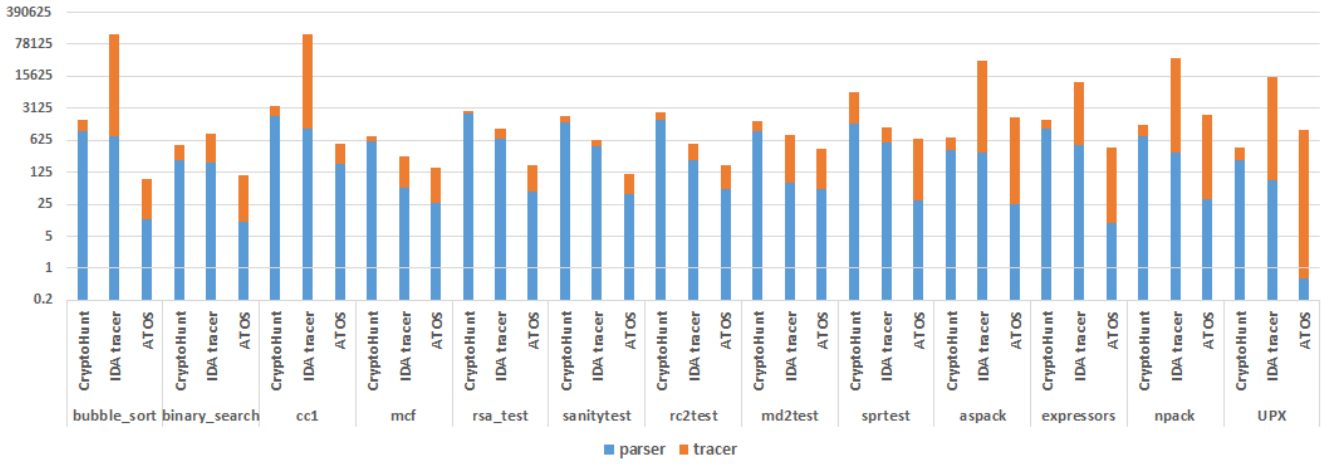


FIGURE 7. Time overhead of different approaches. To show the results more clearly, y axis is normalized using the base 5 logarithm.

IDA tracer cannot record the details of dynamically generated code. Only the addresses of the instructions that generated at runtime are recorded, which reduces the amount of space required.

Time overhead results. Figure 7 shows the time overhead compared to native execution. The y-axis has a logarithmic scale to base 5. Note that we have added the static analysis overhead into the tracer's overhead and all the data is normalized to the time spent for the native execution.

The time spent by IDA tracer is really high, which is in line with our expectations. Although ATOS is built upon IDA Pro, it performs better, especially when considering the overall time, which includes both tracing and parsing. Because the trace produced by ATOS is much shorter than those produced by the other approaches, ATOS performs best for all samples, except for aspack and UPX packed samples. This is because ATOS performs static analysis during the execution when the dynamically generated code is executed. Figure 7 clearly shows that ATOS performs better than the other approaches in most cases.

Figure 8 shows the tracer overhead for CryptoHunt and ATOS , and Figure 9 shows the parser overhead for parsing the trace log files obtained by CryptoHunt and ATOS . Although ATOS spends more time in tracing packed samples, overall ATOS spends much less time tracing than CryptoHunt tracer. The parsing time for ATOS is much less than that for CryptoHunt.

Complexity. Additionally, we evaluate the time computational complexity of the proposed algorithms. There are three main parts in ATOS : static analysis, tracing, and parsing trace log file. Suppose there are `v` vertices, `e` edges, and `c` cycles in the CFG.

In the static analysis phase, a deep first search algorithm is applied to generate the CFG and record the essential information for branch-level tracing. The time complexity is `O(v+e)` and the space complexity is `O(e)` for performing DSF and `O(v)` for recording branch level tracing information. Simple cycles are statically identified by applying Johnson's algorithm, with `O((v+e)(c+1))` time complexity and `O(n+e)` space complexity. SCC loops are statically

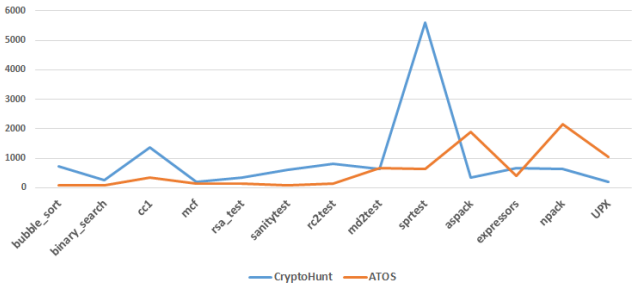


FIGURE 8. Tracer overhead of ATOS and CryptoHunt.

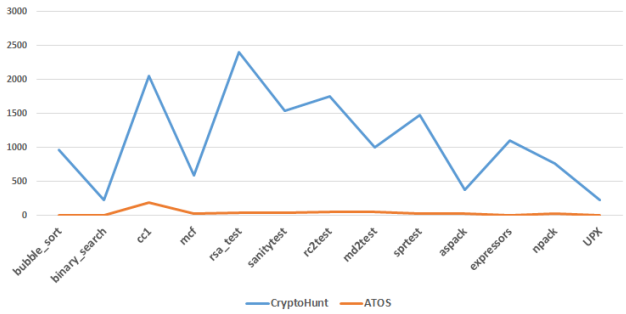


FIGURE 9. Parser overhead of ATOS and CryptoHunt.

identified by utilizing Kosaraju's algorithm. They run in `O(v+e)` time and need `O(v)` space.

In the dynamic analysis phase, ATOS runs `O(1)` time to locate control flow in the shadow CFG. In addition, it needs `O(1)` time to record the current execution state and `O(n)` time to locate the loop body where `n` is the number of loops.

Besides, it needs `O(n)` time to recover the simplified trace, where `n` denotes the number of the basic blocks that have additional information of the folded trace.

Conclusion. ATOS does well in this evaluation. Much shorter traces than other approaches are output, although it obtains both trace and CFG information of the program. Using the disassembly results, we can recover the instruction trace. ATOS also performs best in most cases in the time overhead evaluation, which proves that that the length of the trace is significant. This evaluation indicates that the longer the trace is, the more time is needed to parse it, which is fully in line with our intuitive perception.

D. TIME REDUCED FOR EXISTING APPROACHES

Q3: Can ATOS help in reducing the whole analysis time compared to existing approaches?

In the above evaluation, we have discussed the overhead of tracing and parsing trace log file. We evaluate whether ATOS can help in reducing the time overhead of applications that utilize tracing as a built-in component. The state-of-art cryptographic function detection approach (CryptoHunt [4]), the code simplification approach for virtualized binary (VMHunt [31]) and packer & malware analysis approach (DPI [3]) is used as the benchmark. Note that since DPI provides a web service without open-source code, we do not evaluate the different phases of these applications. The entire analysis time spent is recorded and OpenSSL, packed, and APT are used in this evaluation.

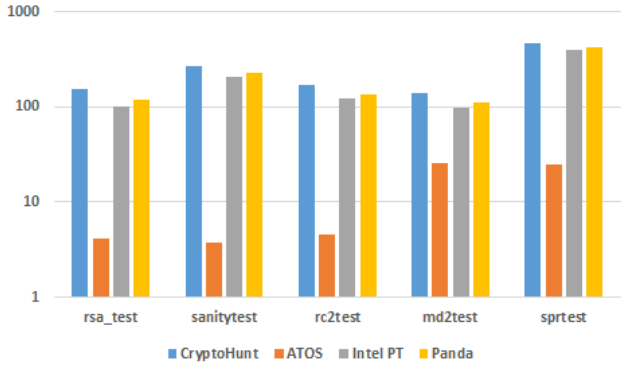


FIGURE 10. Overall performance among ATOS , Intel PT, Panda, and CryptoHunt.

CryptoHunt evaluation approach. The OpenSSL samples are used for CryptoHunt. Since CryptoHunt is open-source, we replace the default tracer of CryptoHunt with ATOS , Intel PT, and Panda respectively and compare the overall time spent with the time spent by the original CryptoHunt.

CryptoHunt evaluation results. Figure 10 shows the time spent by CryptoHunt with different tracers. CryptoHunt. All of them can identify the cryptographic functions correctly. Owing to the shorter trace generated by ATOS , CryptoHunt with ATOS performs much better than the others, although the tracing process of Intel PT is really fast. The result corroborates the fact that the length of the trace significantly affects the time overhead of the post-analysis. Specifically, ATOS helps to reduce the analysis time of original CryptoHunt by about 34 times on average.

VMHunt evaluation approach. The samples packed by VMProctect[8] are used to evaluate the VMHunt. VMHunt is open-sourced just like CryptoHunt. We replaced the default tracer of VMHunt with ATOS and recorded the overall time spent. Then, the time spent of the original VMHunt is compared with VMHunt with ATOS .

VMHunt evaluation results. Figure 11 depicts the time spent by the VMHunt with ATOS and the original VMHunt. Both the original VMHunt and the VMHunt with ATOS as its tracer can simplify the virtualized code. ATOS generated a good shorter trace, and VMHunt with ATOS greatly outperforms the original VMHunt. In particular, ATOS helps to reduce the analysis time for about three times on average. Since the innumerable loop structures are difficult to be fully identified in virtualized samples and ATOS randomly records some of them statically, the loop bodies of the original code are only partially recorded and optimized. Therefore, the improvement made is not as significant as that of CryptoHunt with ATOS . Moreover, ATOS is helpful.

[8]https://vmpsoft.com/

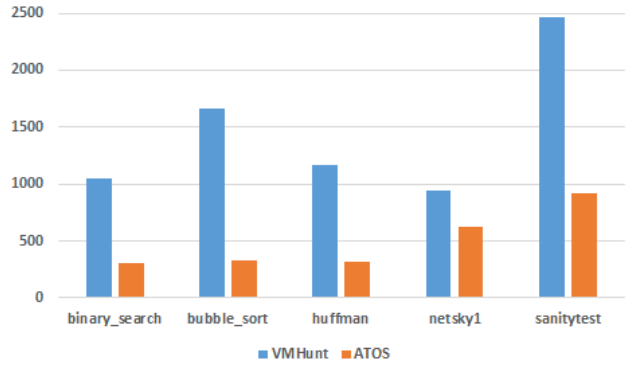


FIGURE 11. Overall performance between ATOS and VMHunt.

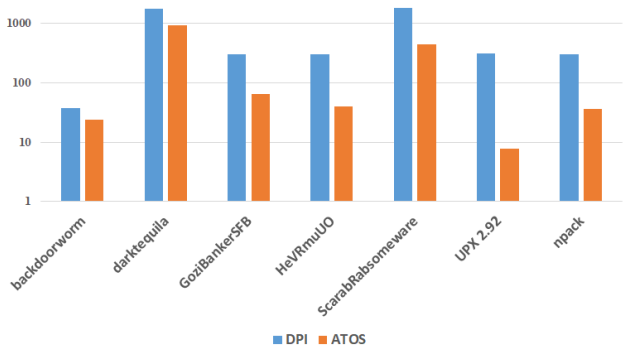


FIGURE 12. Results of DPI and ATOS.

DPI evaluation approach. The packer samples and real APT samples are utilized for DPI. We extend the ATOS to report equivalent information such as DPI. ATOS can easily report the unpacking routine of packers. Whenever the executed bytes are not the same as those recorded in the shadow CFG, it demonstrates that a new layer is built; static analysis is performed to make the CFG consistent with the execution state. Since ATOS cannot observe the repacked code which is not executed, the only limitation is that it cannot identify the packers of Type VI [3].[9] However, this is unimportant because it does not affect anything but the type of the packer defined by DPI. Finally, we compare the analysis time of ATOS and DPI.

DPI evaluation results. Figure 12 illustrates the analysis time for ATOS and DPI. It can be observed that the analysis time for ATOS is much faster than that of DPI, on the tested samples. There is a manual verification in the web service of DPI, and the web service seems to be slightly unstable. Therefore, we did not perform a large-scale evaluation of ATOS and DPI. In addition, we believe that DPI can also analyze more packers and malware. In Section VII-E, we present more results of ATOS regarding packers and real-world malware.

Conclusion. ATOS is able to help to reduce the analysis time of existing approaches.

[9]Insn-CP will be set on the modified code because of the hit of WE-CP, however, the Insn-CP will not be triggered, so ATOS cannot identify packer behaviour of Type VI.

TABLE 4. Evaluate on packed samples.

	ATOS length	ATOS time(s)	Find OEP	BBDSE length	IDA tracer time (s)
UPX	645	7.814	Y	62091	286.020
ASPack	4597	30.458	Y	377349	535.291
WinUPack	3968	27.252	Y	657473	1594.379
FSG	870	9.252	Y	68978	139.824
MEW	893	13.593	Y	59320	121.476
NPack	5900	36.499	Y	138231	348.009
Expressor	1573	17.799	Y	635356	1417.650
PECompact	645	7.814	Y	62091	176.992
ZProtect	11649	105.718	Y	-	1289.439

E. APPLICATIONS

Q4:Whether ATOS can be helpful to more real applications?

In the former evaluation, we have shown that ATOS can help to improve the analysis performance of existing approaches. In this section, we introduce several applications that can be built on ATOS .

Approach. First we evaluate ATOS on more packers. A simple binary that writes a file is packed by different packers. Since ATOS is only a trace solution, we do not use any additional option of the packers that may disable the tracing process in this evaluation. Therefore, the original code can be executed successfully in the evaluation. In addition, since we cannot access the programs evaluated by BB-DSE, the data of BB-DSE result are only used as reference data on the length of the trace. Besides, IDA tracer is chosen as the benchmark for evaluating the time spent, because (1) BB-DSE does not report the analysis time; (2) ATOS is developed on IDA tracer.

Then, in case the malware detects the analysis engine at runtime, we also build a path exploration approach based on ATOS . We execute ATOS according to the path exploration strategy to improve the coverage of dynamic analysis. The indirect control flow transfers, exceptions, and self-modification layers are recorded during the exploration. Real APT samples are used in this evaluation.

Afterward, we show the benefit of online maintained CFG. The obfuscated code may split a single function into different pieces and store them in discontinuous memory, which makes it difficult to recognize the function boundary for a state-of-art disassembler. Moreover, when some of the code in the loop structure is generated at runtime or *ret* instruction is involved in the loop structure, it is difficult to identify the loop structure at runtime.

Finally, we discuss the methods used to extend ATOS for the fuzzing test and API-based approaches.

Packer analysis result. Table 4 show results of ATOS as well as the length of the trace and the time spent on tracing.

The second and third columns indicate the ATOS results. The fourth column shows that ATOS is able to identify the original entry points of these packers, since the `main` function can be marked by F.L.I.R.T [32]. [10] The fifth column is the length recorded by BB-DSE. We can obtain the CFG

[10]ATOS make it possible by applying static analysis method at runtime. The F.L.I.R.T is used as long as new code is generated.

TABLE 5. Path exploration and malware analysis results. *exec* means the number of exceptions, *ind-ret* means the number of *ret* instructions that do not return to the caller of the function, *ind-call* means the number of indirect call instructions, *layers* means the number of code layers that are generated at runtime by self-modification.

	exec	ind-ret	ind-call	layers
vCfjTmdR	1	0	76	3
ScarabRansomware	2	1	68	4
GoziBankerISFB	1	0	20	2
HeVRmuUO	3	0	50	1
KRKeMaIts	4	1	65	2
lsPEcswsco	0	0	48	3
arsstealersafeloader	1	1	72	6
DarkTequila	6	0	43	3
BackdoorWormSMB	107	0	44	1

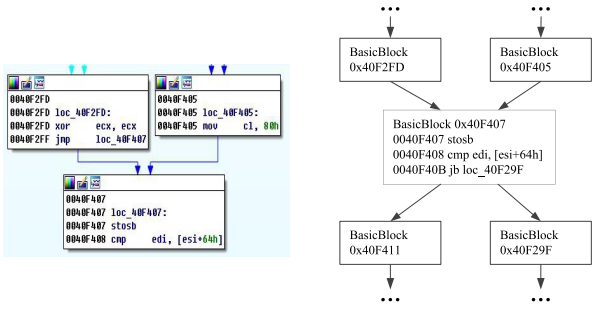
of the packer within a trace of less than 1% of that recorded by BB-DSE. *ZProtect* is not reported by BB-DSE, so there is no result for it in the fifth column. The last column is the tracing time of IDA tracer. We can see that ATOS is much more efficient than the original IDA tracer.

Path exploration and malware analysis results. Table 5 shows the analysis results of the newly reported APT malware. The second column shows the number of deliberately set exceptions which means only the exception is triggered, the further payload can be executed. The third column shows the number of abnormal return instructions. The normal *ret* instruction will lead the control flow back to the caller of the current function. However, packers and malware may break this rule and the *ret* instruction is used for leading the control flow to a certain address. This technique is called stack tempering in BB-DSE, which can be identified by ATOS is. The fourth column is the number of newly found indirect call sites. Only when a certain path is executed, do we get to know the real call site of the indirect call instruction. The last column is the layers contained in the samples. A *Layer* means a piece of code that is generated at runtime which is first introduced by DPI. As discussed above, ATOS can also identify the number of layers.

The result indicates that ATOS can be used as the front end of a malware analysis engine.

Note that *BackdoorWormSMB* requires to open some services installed in the operating system. However, the services are invalid by default in our environment. Thus, the sample continually triggers exceptions and a total of 107 exceptions are recorded. We extends ATOS to bypass certain exceptions according to the path exploration strategy, so ATOS is able to trace the sample as well.

CFG construction results. ATOS constructs the entire CFG during the execution and the loop can be easily recognized. Take the packer *WinUpack* as an example. Figure 13 (a) shows the results for IDA Pro. The later code from 0x40f08 is dynamically generated so that IDA Pro cannot obtain the whole CFG. Even if the execution has come to the address, IDA cannot generate the CFG by default either. Figure 13 (b)



(a) CFG generated by IDA (b) CFG generated by ATOS

FIGURE 13. Part of CFG generated by ATOS.

shows the reconstruction result of ATOS . Since the entire CFG is very large, the key part is shown in the figure. It is obvious that ATOS can help to understand the program better, both for human analyzers and automatic approaches.

Other applications. Besides the above applications, ATOS can be used in nearly all applications that need tracing methods. For example, state-of-art fuzzing approaches usually do not care about the execution details. Instead, they often have a requirement to identify whether the code has been executed (as known as code coverage). At this time, we can add a new type of check points in ATOS , which are only set on the code that has not been executed. If the execution with a seed leads the control flow to an un-executed area, the check point on the code will be triggered. In this way, we can determinewhether the seed has hit an un-executed area. This idea has been used in UnTracer [43].

As ATOS is able to adjust the tracing granularity according to the tracing status, we can also extract the API information during the tracing process. Therefore, ATOS can be used in API-based approaches [23], [44].

Conclusion. ATOS is useful for various tracing-based applications.

VIII. DISCUSSION

ATOS is not intended to be a new tracing mechanism, but rather a novel tracing method that improves the performance of tracing based offline mode analysis. The evaluation shows that ATOS is able to reduce the overhead of existing approaches.

In this section, we first discuss the difference between ATOS and existing approaches from the concept aspect. Since there is a large body of related work in the areas of tracing, to limit our scope of discussion, we concentrate on several widely used approaches in the discussion. Then, we discuss the future work that can make ATOS more powerful.

Compared to Intel Pin. Pin uses an incremental manner to build the PinTool. Callback functions will be called every time according to the tracing granularity, even if the user does not want to trace a piece of code under a certain tracing level. Lots of filters should be placed in the callback functions to avoid tracing the specific code.

Besides, the statically linked library functions are difficult to be identified by Pin approaches. CryptoHunt has shown

that to identify the function by analyzing the trace is time-consuming.

In fact, other dynamic instrumentation approaches also have the above problems. Though one can also implement a tool such as ATOS with Pin, as far as we know, no such approach proposed.

Compared to Intel PT. We do not implement ATOS upon PT, based on the following concerns. First, PT based approaches cannot output the instructions trace directly. Only part of the control flow transfer instructions addresses are recorded, and the record is stored in compressed form. Therefore, static analysis is needed to decode the trace [5]. Obviously, it is difficult to obtain the details of the dynamically generated code which is widely used in malware and JIT programs.

Second, we have trouble in building a virtual environment or a sandbox with PT. In fact, as far as we know, no such existing work is available currently, though a recoverable controlled virtual environment is essential for malware analysis. However, we regard implementing ATOS with PT as future work, since debugger is not the only best choice. Besides, the way to record the information by PT is just the same as the proposed *branch level tracing* in this paper. The key problem is decoding the dynamically generated code.

Compared to Panda. We do not implement ATOS upon Panda, different phase of ATOS . although the methods of ATOS can be used in the reply phase to record less but sufficient information for further analysis. However, it may not be a good choice to perform offline analysis with Panda. The same thing happens in other QEMU-based approaches. These approaches are more suitable for online mode analysis. Besides, it is difficult to perform path exploration of malware with Panda, since many malware are environment sensitive and logic bombs are common in malware.

Future work. First, malware can disable the analyzer by attacking the debugger or detecting the runtime environment [45]. This problem can be alleviated by path exploration approaches and we are trying to propose one with ATOS .

Moreover, ATOS is weak at tracing multi-thread and multi-process programs. In fact, this is a general problem of all the local perspective approaches. There is no overall solution to this problem, because some of the approaches are too complex to be solved by program analysis methods. In the current implementation, we try to hook the entry of new threads or processes to monitor the details, which can alleviate the problem.

ATOS performs well as shown in the evaluation, even though it has some limitations. Additionally, ATOS is still under developing to improve its capability.

IX. CONCLUSION

This paper proposes ATOS , an efficient tracing method with adaptively adjusted granularity. It dynamically adjusts the tracing granularity to reduce the size of the trace log file without missing essential information. A generic loop structure folding method is used to reduce the tracing of repeatedly executed code, and an efficient CFG maintenance scheme is introduced to support both online and offline analyses. The prototype of ATOS is implemented. The evaluation results validate the efficiency of ATOS .

REFERENCES

[1] S. Bardin, R. David, and J.-Y. Marion, "Backward-bounded DSE: Targeting infeasibility questions on obfuscated codes," in *Proc. IEEE Symp. Security Privacy (SP)*, May 2017, pp. 633–651.

[2] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: Force-executing binary programs for security applications," in *Proc. 23rd USENIX Secur. Symp.*, 2014, pp. 829–844.

[3] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2015, pp. 659–673.

[4] D. Xu, J. Ming, and D. Wu, "Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2017, pp. 921–937.

[5] Intel Corporation. *Intel Processor Trace*. Accessed: Mar. 14, 2019. [Online]. Available: https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing

[6] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "KAFL: Hardware-assisted feedback fuzzing for OS kernels," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 167–182.

[7] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, "PT-CFI: Transparent backward-edge control flow violation detection using intel processor trace," in *Proc. 7th ACM Conf. Data Appl. Secur. Privacy*, Mar. 2017, pp. 173–184.

[8] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "PEBIL: Efficient static binary instrumentation for Linux," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2010, pp. 175–183.

[9] A. S. Charif-Rubial, D. Barthou, C. Valensi, S. Shende, A. Malony, and W. Jalby, "MIL: A language to build program analysis tools through static binary instrumentation," in *Proc. 20th Annu. Int. Conf. High Perform. Comput.*, Dec. 2013, pp. 206–215.

[10] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 190–200, Jun. 2005.

[11] Open Source. (2016). *Dyninst: An Application Program Interface (API) for Runtime Code Generation*. Accessed: Sep. 10, 2019. [Online]. Available: http://www.dyninst.org

[12] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 89–100, Jun. 2007.

[13] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proc. 23rd Annu. Comput. Secur. Appl. Conf. (ACSAC)*, Dec. 2007, pp. 421–430.

[14] V. Chipounov, V. Kuznetsov, and G. Candea, "S^2E: A platform for *in-vivo* multi-path analysis of software systems," in *Proc. 16th Conf. Architectural Support Program. Lang. Oper. Syst. (ASPLOS)*, Mar. 2011, pp. 265–278.

[15] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee, "Tappan zee (north) bridge: Mining memory accesses for introspection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2013, pp. 839–850.

[16] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "PolyUnpack: Automating the hidden-code extraction of unpack-executing malware," in *Proc. 22nd Annu. Comput. Secur. Appl. Conf. (ACSAC)*, Dec. 2006, pp. 289–300.

[17] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proc. IEEE Symp. Secur. Privacy*, May 2010, pp. 317–331.

[18] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2015, pp. 674–691.

[19] C. Eagle, *The IDA Pro Book*. San Francisco, CA, USA: No Starch Press, 2011.

[20] M. Zalewski. *American Fuzzy Lop*. Accessed: Sep. 10, 2019. [Online]. Available: http://lcamtuf.coredump.cx/afl/

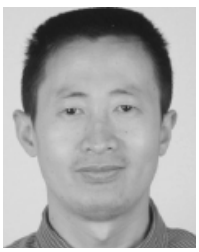
[21] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 1032–1043.
[22] S. Lu, M. Zhang, Z. Li, H. Li, X. Kuang, and G. Zhao, "Dynamic binary translation and instrumentation based function call tracing," *J. Comput. Res. Develop.*, vol. 56, no. 2, pp. 421–430, Feb. 2019.
[23] R. Veeramani and N. Rai, "Windows API based malware detection and framework analysis," *Int. J. Sci. Eng. Res.*, vol. 25, no. 3, pp. 1–6, Mar. 2012.
[24] Z. Cheng, X. Chang, L. Zhu, R. C. Kanjirathinkal, and M. Kankanhalli, "MMALFM: Explainable recommendation by leveraging reviews and images," *ACM Trans. Inf. Syst.*, vol. 37, no. 2, Mar. 2019, Art. no. 16.
[25] M. Luo, F. Nie, X. Chang, Y. Yang, A. G. Hauptmann, and Q. Zheng, "Adaptive unsupervised feature selection with structure regularization," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 4, pp. 944–956, Apr. 2018.
[26] M. Luo, X. Chang, L. Nie, Y. Yang, A. G. Hauptmann, and Q. Zheng, "An adaptive semisupervised feature analysis for video semantic recognition," *IEEE Trans. Cybern.*, vol. 48, no. 2, pp. 648–660, Feb. 2018.
[27] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox fuzzing for security testing," *ACM Queue*, vol. 55, no. 3, pp. 40–44, Mar. 2012.
[28] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 380–394.
[29] B. Yadegari and S. Debray, "Bit-level taint analysis," in *Proc. IEEE 14th Int. Working Conf. Source Code Anal. Manipulation*, Sep. 2014, pp. 255–264.
[30] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea, "Selective symbolic execution," in *Proc. 5th Workshop Hot Topics Syst. Dependability (HotDep)*, 2009, pp. 1–6.
[31] D. Xu, J. Ming, Y. Fu, and D. Wu, "Vmhunt: A verifiable approach to partially-virtualized binary code simplification," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 442–458.
[32] Hex-Rays. *Fast Library Identification and Recognition Technology (F.L.I.R.T.)*. Accessed: Mar. 14, 2019. [Online]. Available: https://www.hex-rays.com/products/ida/tech/flirt/index.shtml
[33] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proc. 10th ACM Conf. Comput. Commun. Secur.*, Oct. 2003, pp. 290–299.
[34] G. Bonfante, J. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry, "CoDisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 745–756.
[35] B. Yadegari and S. Debray, "Symbolic execution of obfuscated code," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 732–744.
[36] P.-L. Giscard, P. Rochet, and R. Wilson, "Evaluating balance on social networks from their simple cycles," Jun. 2016, *arXiv:1606.03347*. [Online]. Available: https://arxiv.org/abs/1606.03347
[37] D. B. Johnson, "Finding all the elementary circuits of a directed graph," *SIAM J. Comput.*, vol. 4, no. 1, pp. 77–84, 1975.
[38] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, MA, USA: MIT Press, 2009.
[39] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "PTfuzz: Guided fuzzing with processor trace feedback," *IEEE Access*, vol. 6, pp. 37302–37313, 2018.
[40] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence," in *Proc. NDSS*, 2019, pp. 1–15.
[41] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path sensitive fuzzing," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 679–696.
[42] N. M. Hai, M. Ogawa, and Q. T. Tho, "Obfuscation code localization based on CFG generation of malware," in *Proc. Int. Symp. Found. Pract. Secur.* Cham, Switzerland: Springer, 2015, pp. 229–247.
[43] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," in *Proc. IEEE Symp. Security Privacy (SP)*, 2019, pp. 1122–1137.
[44] M. Alazab, R. Layton, S. Venkataraman, and P. Watters, "Malware detection based on structural and behavioural features of api calls," in *Proc. Int. Cyber Resilience Conf.* Joondalup, WA, Australia: Edith Cowan Univ., 2010, pp. 1–11.
[45] D. Kirat and G. Vigna, "Malgene: Automatic extraction of malware analysis evasion signature," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 769–780.

HE SUN was born in Qiqihar, Heilongjiang, China, in 1990. He received the B.E. and M.S. degrees from the PLA University of Science and Technology, in 2013 and 2016, respectively, where he is currently pursuing the Ph.D. degree. He is also a Visiting Student with Tsinghua University. His research interests include malware analysis and network security.

CHAO ZHANG was born in Hubei, China, in 1986. He received the B.E. and Ph.D. degrees from Peking University, in 2008 and 2013, respectively. He is currently an Associate Professor with Tsinghua University. His research interests include software analysis and machine learning.

HE LI was born in Datong, Shanxi, China, in 1989. He received the B.E. degree from PLA Information Engineering University, in 2011, where he is currently pursuing the M.S. degree. He is also a Visiting Student with Tsinghua University. His research interest includes system security.

ZHENHUA WU was born in Zaozhuang, Shandong, China, in 1989. He received the B.E. degree from PLA Information Engineering University, in 2012, where he is currently pursuing the M.S. degree. He is also a Visiting Student with Tsinghua University. His research interests include malware analysis and software supply chain security.



LIFA WU was born in Hubei, China, in 1968. He received the Ph.D. degree from Nanjing University, in 1998. He is currently a Professor with the Nanjing University of Posts and Telecommunications. His research interests include network security and protocol reverse engineering.

YUN LI was born in Zhoukou, Henan, China, in 1997. She received the B.E. degree from the Beijing University of Posts and Telecommunications, in 2019. She is currently pursuing the Ph.D. degree with Tsinghua University. Her research interests include program analysis, system security, and blockchain security.