PyPANDA: Taming the PANDAmonium of Whole System Dynamic Analysis

Luke Craig*, Andrew Fasano*†, Tiemoko Ballo*, Tim Leek* Brendan Dolan-Gavitt‡ William Robertson†
*MIT Lincoln Laboratory {Luke.Craig,Andrew.Fasano,Tleek,Tiemoko.Ballo}@ll.mit.edu
†Northeastern University fasano.a@northeastern.edu, w.robertson@northeastern.edu
‡New York University brendandg@nyu.edu

Abstract—When working with real world programs, dynamic analyses often must be run on a whole-system instead of just a single binary. Existing whole-system dynamic analysis platforms generally require analyses to be written in compiled languages, a suboptimal choice for many iterative analysis tasks. Furthermore, these platforms leave analysts with a split view between the behavior of the system under analysis and the analysis itself—in particular the system being analyzed must commonly be controlled manually while analysis scripts are run. To improve this process, we designed and implemented PyPANDA, a Python interface to the PANDA dynamic analysis platform. PyPANDA unifies the gap between guest virtual machines behavior and analysis tasks; enables painless integrations with other program analysis tools; and greatly lowers the barrier of entry to whole-system dynamic analysis. The capabilities of PyPANDA are demonstrated by using it to dynamically evaluate the accuracy of three binary analysis frameworks, track heap allocations across multiple processes, and synchronize state between PANDA and a binary analysis platform. Significant challenges were overcome to integrate a scripting language into PANDA with minimal performance impact.

I. Introduction

Although program analyses commonly focus on individual programs, real-world computing environments are generally complex, multi-process systems running with various privilege levels (i.e., kernel and userspace) where data flow between processes through porous interfaces. Program analysis frameworks capable of supporting such environments are able to conduct whole-system instrumentation [5], slicing [11], and multi-process taint tracking [15], all of which can be used in reverse engineering, vulnerability discovery, and root cause analysis. These whole-system analyses are largely conducted dynamically (through emulation), as statically reasoning about whole-system interactions requires understanding OS internals (e.g., scheduling, IPC) as well as reasoning about an exponentially growing set of feasible states.

While single-application static and dynamic analysis frameworks such as angr [47], Manticore [25], Triton [39], IDA Pro[1], and Ghidra[2] all support conducting analyses from scripting languages, such functionality is rarely present in whole-system dynamic analysis platforms leading to cumbersome workflows. For example, consider the task of conducting a whole-system dynamic taint analysis on data sent to a custom kernel module that ultimately flow into a user space application. An analyst must approach this task through two distinct, but complementary, processes. First, they must drive the guest system's behavior: boot the system, log in, obtain the relevant source code and toolchains, compile the code (or copy in a prebuilt binary), and load the kernel module. Then, once the system under test is properly configured, the actual analysis can begin by further driving the guest system's behavior to send data into the kernel module and, at the same time, asking the analysis platform to apply taint labels to the data in the guest's memory. After some indeterminate amount of time, the analyst would then need to query the analysis platform to identify where and how tainted data reached the userspace application.

This workflow highlights a number of significant challenges largely related to user experience, as well as an active research problem. The research challenge lies in bridging the semantic gap [18] to extract meaning from the emulator's view of guest memory (e.g., how the results from the taint analysis can be tied back to process names and non-randomized program counters). The user experience challenges are easier to tackle, but no less important from an end-user's perspective. These include copying files into the guest, driving guest behavior, and synchronizing guest behavior with analysis tasks.

To address these challenges, we designed and implemented PyPANDA: a Python 3 interface to the PANDA [10] whole system analysis platform. PyPANDA allows for driving a guest execution, running Python code at any PANDA callback (capable of reading or writing guest state), and interacting with PANDA plugins. Since Python has a large ecosystem of libraries, PyPANDA also enables novel combinations of existing libraries with a whole system dynamic analysis framework.

The remainder of this paper is structured as follows. §II

Workshop on Binary Analysis Research (BAR) 2021
21 February 2021
ISBN 1-891562-69-X
https://dx.doi.org/10.14722/bar.2021.23001
www.ndss-symposium.org

[1]https://hex-rays.com/products/ida
[2]https://ghidra-sre.org

covers necessary background information on the PANDA framework. The overall design goals for the project are presented in §III, and the implementation details are discussed in §IV. In §V, we detail PyPANDA's key features. §VI examines three program analysis tasks conducted with PyPANDA: comparing binary analysis frameworks on-the-fly, dynamically monitoring user and kernel-space heap allocations, and an entropy analysis of packed binaries linked with a web interface which transfers program state to Ghidra. We discuss our implementation of PyPANDA in §VII. Finally, we examine related work and conclude in §VIII and §IX.

II. PANDA

As PyPANDA builds atop PANDA, we provide background information on the PANDA dynamic analysis platform in this section. PANDA forks the QEMU emulator [1] to add callbacks before significant events in the emulation workflow, a record and replay system to enable offline analyses of guest behavior, and APIs for reading and writing guest state [10]. Together, these features enable dynamic analyses of "guest" systems (virtual machines) running under PANDA. PANDA's plugin system allows analysts to create tailored dynamic analyses by combining reusable, core plugins (e.g., a taint system) with custom, analysis-specific logic. PANDA has been used for reverse engineering, malware analysis, crash analysis, and trace collection [40, 42, 43, 27].

PANDA provides 48 callbacks at which plugins can register custom code to run. Each callback represents a low-level, architecture-agnostic operation such as before a basic block of code is executed, after a virtual memory address is read, or before the address space identifier (ASID) changes.

PANDA's record and replay system enables analysts to efficiently record guest behavior with little overhead and later replay its behavior under analysis. This allow for running slow analyses without the risk of analysis overhead affecting guest behavior.

PANDA has a small API for plugins to query and alter guest state but these interactions largely occur by plugins directly modifying emulator internal state.

PANDA provides a set of reusable plugins to enable common dynamic analysis tasks. Three notable plugins are PANDA's dynamic taint system taint2, a syscall tracking system syscalls2, and OS-introspection for both Linux and Windows OSI[3]. Plugins interact with one another through PANDA's plugin-to-plugin interface (PPP) which allows for plugins to provide custom callbacks which other plugins can request to be notified of. Plugins can also expose custom APIs which can be consumed by other plugins. The PPP interface allows for composition of plugin logic. For example, the file_taint plugin which applies taint labels to data read from a specified file, builds off syscalls2 and OSI to identify when data is read from the specified file and then uses taint2 to apply taint labels to those bytes.

[3]PANDA's OS-introspection requires using OSI profiles for the guest which describe sizes and offsets of key kernel data structures. These profiles can be generated from inside the guest or by analyzing a kernel's DWARF information when such information is available.

The workflow for a custom PANDA analysis typically begins by capturing a recording of an analyst interactively driving a guest system to exhibit some behavior of interest. With this recording in hand, the analyst begins the process of creating a new PANDA plugin in C or C++. That plugin can use the PPP interface to interact with other plugins and query the guest memory or registers at any point in time.

In our experience developing dozens of such plugins, we have found this task to be an iterative process. In their presentation of PANDA, Dolan-Gavitt et al. describe this clearly:

> "Plugins are typically written quickly and iteratively, running the replay over and over to construct more and better and deeper understanding of the important aspects of system execution, given the RE task." [10]

Although this task is quick and iterative, more akin to prototyping than production development, PANDA only supports plugin development in C and C++.

III. Design Goals

We begin our discussion of PyPANDA by identifying five design goals for the project.

1) High Performance Scripting: It is well known that Python programs typically execute slower than equivalent programs written in compiled languages such as C and C++ [13]. However, scripting languages such as Python are often better for rapid prototyping and gluing components together [29]. With this in mind, we determined that adding a new scripting interface to PANDA could greatly improve the project if the performance impact was acceptable.

2) Unified Analysis: The traditional PANDA workflow involves writing a plugin and either creating a recording manually or by using a script to control the serial console and PANDA monitor. This leads to scenarios in which key code is disparate: the process necessary to reproduce results may be scattered across different scripts or require manual effort.

We wish to replace this fragmented view of analysis and implementation and unify interaction outside the machine and analysis at the callback level. Beyond enabling a standardized reproducible method to generating recordings, this also allows for the entirety of the analysis to be done in a single script.

3) Facilitation of Integration: Integrations with publicly available libraries and frameworks enable developers to rapidly develop analyses without rebuilding common components. By expanding PANDA's interfaces to work

with a new language, we will allow PANDA developers to easily integrate with many additional libraries.

4) Ease of Use: Whole-system dynamic analysis can be difficult to learn due to the complexity of the systems being analyzed and unpolished user interfaces. We wish to lower the barrier of entry to this class of analyses by simplifying the PANDA interface without limiting experienced users.

5) Access to Core Emulator Data Structures: The API provided by PANDA enables analysts to complete common tasks such as reading or writing memory, but developing new dynamic analyses often requires interacting with a guest system in novel ways. To avoid limiting users to just features directly exposed by the PANDA API, users of our tool should be able to access and modify internal emulator data structures representing the guest system's state.

IV. System Design

To implement PyPANDA, we modified PANDA to build as a shared object and automatically generate headers parsable by CFFI. We then created a Python module to provide clean interfaces to PANDA's APIs, new features, unit tests, and examples. PyPANDA was implemented in 7,791 lines of Python code (after excluding autogenerated files) as measured by SLOCCount [48].

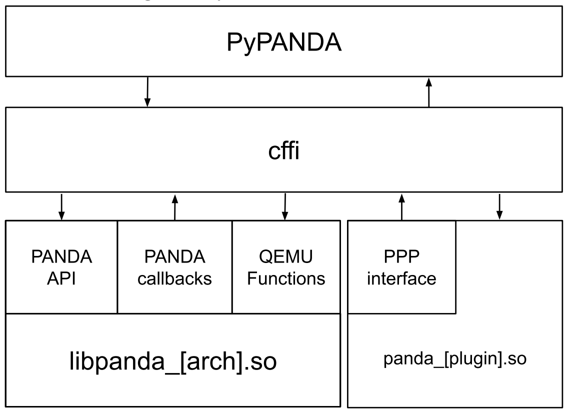
A. PANDA as a Library

PANDA is traditionally built as a series of binaries, one per supported architectures (panda-system-x86_64, panda-system-arm, etc.) and plugins are built for each architecture as shared objects. PANDA currently supports x86, AMD64, ARM, AArch64, PowerPC, and MIPS, but can be extended to support additional architectures. To enable PyPANDA we elected to build PANDA itself as a shared object. This allows us to intentionally invert the control the traditional PANDA model. In doing so we can provide unify the disparate components of whole-system dynamic analysis into a single script. Additionally, analyses will have full access to the emulator's internal data structures by means of shared library exported variables.

We chose not to simply embed a Python interpreter within the emulator, as done by PyREBox [7]. Although that approach would have simplified the development process, it cannot support our desired inversion of control model.

Constructing the libpanda objects required significant modifications to PANDA's Makefile and its codebase. In particular, PANDA's main function had to be split onto multiple stages such that control could transfer between PANDA and PyPANDA during initialization. Additionally, we expanded the PANDA API, a foreign-language agnostic interface for controlling PANDA and utilizing its functionality, in order to allow for library

Fig. 1. PyPANDA's architecture.



consumers to interact with the core of PANDA as well as its shared object plugins.

PyPANDA's overall architecture is shown in Figure 1. The libpanda objects and PANDA API are designed to allow clients to control PANDA (e.g., begin running a guest) as well as registering callbacks, interacting with C/C++ plugins, and altering guest state by exposing PANDA's APIs and internal objects. Together, these APIs and object allow for efficient control of and communication with PANDA from any language with a C foreign function interface[4].

B. Header Files

CFFI[5] enables interaction between C shared libraries and Python; it uses pycparser[6] to parse C header files that define structures along with function declarations and represents objects as if they were members of a traditional Python class. Unfortunately, pycparser, a pure Python parser for C, can only operate on preprocessed C headers—it cannot support preprocessor declarations such as ifdef and define. This presents a difficult problem when working with a project as large as PANDA. Pycparser documentation recommendations using the C Preprocessor to generate a single header file to avoid this problem. This is quite difficult in practice because pycparser is unable to parse C source which contains data format specifiers, such as attributes. An extra layer of complexity is added for the PANDA project because the output of the build process is not a single executable, but multiple executables targeted at various guest architectures. Structures are commonly defined using preprocessor macros to include different fields for different architectures. Attempting to generate preprocessed headers for PANDA with the C preprocessor produced over 50,000 lines of code and still caused parsing

[4]Although this work focuses on PyPANDA, we note that others have begun development of PANDA-rs, a Rust-interface to PANDA building off the architecture changes we introduced.

[5]https://cffi.readthedocs.io

[6]https://github.com/eliben/pycparser

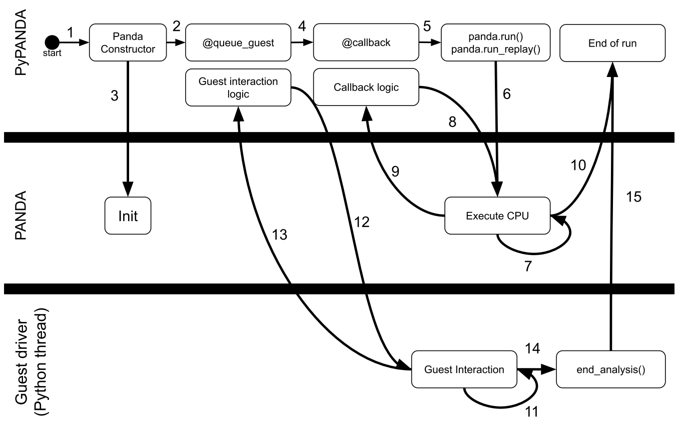
```
Algorithm 1: Simplified Header Generation Process
  Input: List of structure names, $requiredStructureList$ to
         recursively pull definitions for
  Output: Headers for all reachable types
  for $structure$ in $requiredStructureList$ do
      $potentialHeader \leftarrow pahole(structure)$
  while True do
      $errors \leftarrow cffiBuild(potentialHeader)$ ;
      if $errors = \emptyset$ then
          break ;
      else
          $missingStruct \leftarrow parseError(errors)$;
          if $missingStruct$ is nested struct then
              $potentialHeader \leftarrow pahole(missing)$ ;
          else
              $potentialHeader$ cast $missingStruct$ as $void*$;
  return $potentialHeader$;
```

errors for pycparser. Even if the errors could be rectified, this approach is inefficient.

Instead of parsing the source code of PANDA, we use debug information from our compiled shared objects. We use Poke-a-Hole (or pahole) [38], a utility to parse DWARF debug information, to gather information about each structure. Poke-a-Hole was selected because it allows interpretation of classes as structures, suppression of aligned attributes, and suppression of force paddings.

We present a simple solution to this problem in Algorithm 1. For each architecture we support we begin by identifying a list of necessary structures manually. Using Poke-a-Hole, the definitions of these structures are added to a header file. This file and the corresponding shared object is provided into CFFI which may generate errors. These errors will identify the structures referenced, but not defined. When we identify a reference, but not defined structure we add it when necessary. Nested structures must be incorporated into the header file to capture the structure size and relative offsets of its members. On the contrary, referenced structures are merely pointers with an assigned type. We do not need to understand their structure layout to understand the previous type because a pointer is a constant size on a given architecture. To remove unnecessary structure reference we simply represent pointers to structures which are unnecessary as void pointers. Since these structures were not in the initial list of required structures, skipping their definition is acceptable. Our process continues adding structures until CFFI parses the header file without any errors. This solution produces a header file containing a minimal number of structures required (about 60) to operate PyPANDA, which in turn allows it to function with minimal overhead while maintaining in-depth knowledge of all necessary structures. Once generated, CFFI can consume this core header file along with other PANDA headers when running the relevant architecture.

Fig. 2. PyPANDA Lifecycle



V. Key Features

PyPANDA was created to enable rapid development of whole-system dynamic analysis through a simple, yet powerful Python interface. In this section, we walk through the sample code presented in Listing 1, its position in the lifecycle flow with Figure 2, and discuss the key features of PyPANDA.

```python
 1 from pandare import Panda
 2 panda = Panda(generic='i386') # or x86_64, arm, etc.
 3
 4 @panda.queue_blocking
 5 def drive_guest():
 6   panda.revert_sync('root')
 7   panda.copy_to_guest('host_directory')
 8   print(panda.run_serial_cmd('host_directory/a_binary'))
 9   panda.end_analysis()
10
11 @panda.cb_before_block_exec
12 def demo_before_block_exec(cpu, translation_block):
13   # Called before PANDA executes a basic block of code
14   print(f'About to run block at 0x{translation_block.pc:x}')
15   if translation_block.pc == 0x1234:
16     panda.disable_callback('demo_before_block_exec')
17
18 @panda.ppp('callstack_instr', 'on_call')
19 def calling(cpu, dest_addr):
20   #Called by callstack_isntr plugin on function call
21   print(f'Call to 0x{dest_addr:x}')
22
23 @panda.ppp('callstack_instr', 'on_ret')
24 def returning(cpu, dest_addr):
25   # Called by callstack_isntr plugin on function return
26   print(f'Return to 0x{dest_addr:x}')
27
28 panda.load_plugin('asidstory', {'width': 80})
29 panda.run() # Run emulation  until call to end_analysis
```

Listing 1: Example PyPANDA script.

A. PANDA Constructor

As shown in Listing 1 on Lines $1-2$, a script interacts with PANDA by importing the pandare package and constructing a Panda object. The constructor may specify an architecture, the guest memory size, a QCOW file (the QEMU virtual machine disk format which is used by PANDA), and other configurable properties. For convenience, a user may alternatively request a generic image from a list of nine images covering each architecture PyPANDA supports. For each generic image, we provide a QCOW file of a Linux system complete with a snapshot taken post-login and an OSI profile. The provided QCOW will be downloaded to disk prior to use.

B. Guest Interaction

Once a PANDA object has been created, the guest can be controlled through numerous functions as shown on Lines 4 – 9. Instead of waiting for the guest system to boot and then authenticating, an analysis can begin by reverting to a provided snapshot taken after log in. From there, a script can interact with the guest by sending commands over the serial console.

PyPANDA is running both the emulator as well as interacting with the guest. Guest interactions are labeled with @panda.queue_blocking which queues the function to run in a separate thread after the emulation begins. This multithreaded design avoids the risk of deadlocking when the analysis waits for the guest to complete a command. It can be seen visually in Figure 2 through its registration process with @queue_guest and its Guest Interaction loop. Only a function with such a decorator is allowed to call synchronous functions that depend on guest behavior such as revert_sync or run_serial_cmd.

PyPANDA supports copying files into a guest by creating an ISO file, connecting it to the guest system's emulated CD drive, mounting the drive and copying the files to a writable directory. For example, if the host machine has a directory called host_directory containing a binary named binary, the example code would copy that directory into the guest, run the binary, and end the analysis.

C. Starting and Stopping

Once the PANDA object is created and a function is queued up to drive the guest's behavior, a script must start the emulation with a call to panda.run() (Line 29). Once this function is called, the main Python thread will block until the guest powers off or the analysis is ended by another thread such as the call to end_analysis on Line 9 which correspond to Figure 2 edges 10, 14, and 15.

D. PANDA Callbacks

A PyPANDA script can register Python functions to run at any of PANDA's 48 callback functions through a decorator. Lines 10 – 16 show a function running on the before_block_exec callback. This will be called before each basic block of code is executed inside the guest. This callback function is run with two arguments, a CPUState structure which contains details of the virtual CPU and memory as well as a TranslationBlock which describes the block of code to be executed.

The decorator registering a function as a callback can take arguments to give the callback a specific name or to disable it. Figure 2 shows a call to @callback which registers callback logic below it after edge 2. By default the callback is enabled and named the same as the decorated function. A user can enable or disable callbacks by name.

E. PANDA Plugins

PyPANDA may load or unload compiled PANDA plugin and provide arguments as shown on Line 28. PyPANDA may also register callbacks through the PANDA's PPP interface (Lines 18 – 26) which will be triggered by a plugin. Plugins referenced by PPP decorators will be automatically loaded if necessary.

F. Record and Replay

Not shown in Listing 1 is PyPANDA's support for PANDA record and replay. PyPANDA provides a helper function to revert the guest to a snapshot, copy a directory from the host machine into the guest, and record the execution of a command.

```
@panda.queue_blocking
def take_recording(cpu, tb)
  panda.record_cmd('./host_directory/mybin',
           copy_dir='host_directory',
           recording_name='my_recording')
```

A PyPANDA script can register callbacks to introspect guest behavior and then—instead of running a live system—replay a recording using panda.run_replay('replay_name').

VI. Evaluation

In this section we present three case studies using PyPANDA and a comparison of its performance to PANDA. Code and a Docker container to reproduce these examples are available at https://github.com/panda-re/bar2021.

A. Dynamic Oracle IR Testing

1) Motivation: Modern binary analysis tools and techniques are often predicated on the ability to lift raw bytes encoding native processor instructions to a higher-level symbolic form, e.g. an Intermediate Representation (IR). IRs strive to faithfully represent the operational semantics of the underlying code while simultaneously abstracting away architecture-specific details.

Due to the sheer range of static and dynamic analysis tasks a binary analysis IR might support, from static decompiliation to dynamic taint analysis, there is no widely-accepted default criteria for evaluating the efficacy, correctness, or overall design of a given IR. Martignoni et al. [24] utilize differential fuzzing, comparing CPU state between IR-based emulators and physical processors. This technique has drawback of relying on random test case generation, the empirical likelihood of observing meaningful divergence is low. Kim et al. [21] introduce N-version IR Testing, essentially diffing the output of independently developed IRs for a given set of single-instruction inputs. This approach is hampered by a lack of semantic context, it only considers differences between single-instruction IR outputs.

In this PyPANDA case study, we briefly prototype a novel approach for comparing IRs that has practical applications for real-world reverse engineering: Dynamic Oracle IR Testing. QEMU, and, transitively, PANDA use TCG IR to emulate a range of native ISAs. Although this means that neither can be treated as absolute ground truth, both emulators are capable of booting entire operating systems kernels and running complex desktop and sever applications. Thier fidelity is sufficient for real-world tasks like performing malware analysis [22] and collecting code coverage metrics [8].

We assert that this fact makes PANDA a suitable oracle from which to derive practical conclusions about program execution, even if, like prior work in IR evaluation, the derived results do not carry formal guarantees. Our analysis evaluates disparate IRs against dynamic execution events observed for real-world programs under PANDA. Unlike differential fuzzing, we use realistic and non-random inputs. Unlike N-version IR testing, we evaluate against the outcomes of concrete dynamic emulation—not between the static IR lift results.

2) Experiment Design: In our demonstration of Dynamic Oracle IR Testing, we seek to compare the efficacy of leading binary analysis IRs for a real-world reverse engineering task: identifying function call and return semantics in an architecture-neutral fashion. Collecting dynamic call-stack traces is useful for general program understanding—it is a semantically meaningful metric of coverage and shines light on higher-level logic.

Whereas traditional userspace instrumentation utilities can trace system or library calls (e.g. strace and ltrace respectively), they do not have the ability to hook arbitrary internal functions. OS-assisted solutions, like patching bytes to trigger interrupts caught by a debugger, are intrusive. Moreover, static function identification in stripped binaries is, in the general case, an unsolved research problem—knowing where to patch a priori is difficult.

A whole-system emulator, on the other hand, can pause any user or kernel space program at an arbitrary point in execution without patching the target program. Our only remaining problem is function entry and exit identification. Assuming scope is a single concrete execution and not identification of all functions present: how, when analyzing stripped binaries, do we efficiently identify function boundaries and correlate them back to native-ISA program counters? While using a disassembler to check the encoding of every executed instruction might work for a single architecture, the ideal architecture-neutral solution is lifting to an IR to capable of encoding function call semantics.

Thanks to PyPANDA's ecosystem composability, an analyst is able to leverage any binary analysis IR that offers Python bindings. We select three such IRs and evaluate their ability to perform just-in-time call tracing as follows:

- Immediately following the execution of every basic block, filtering by process, we read the bytes encoding that block's instructions out of the guest.
- Each of three different IRs under test lift this basic block "shellcode" to their respective representation.
- We query each representation to find direct (e.g. offset-based) calls, indirect (e.g. register-based) calls, and returns.
- To measure correctness, we evaluate the IR's output when the subsequent basic block is executed—did the IR successfully identify a call and resolve the correct target?

PANDA serves as our dynamic oracle, generating both realistic non-random inputs (instruction byte streams from real-world programs) and expected conclusions (the next basic block executed is the correct destination for any direct call).

This is just one example application of Dynamic Oracle Testing. Our approach, essentially a feedback loop of concrete inputs and concrete results produced by faithful emulation, can be generalized to test any conclusion that can drawn from lifting native instruction streams to an IR.

3) IRs Under Test: We select three IRs based on three criteria:

- Readily available Python bindings: can be rapidly integrated into PyPANDA without the need to compile a library from source.
- Meaningful adoption: forms the basis of tooling used in industry or the by the security research community.
- Multi-architecture: capable of lifting, at minimum, x86, x64, ARM, and MIPS binaries.

VEX, our first selection, was originally introduced in the Valgrind memory profiling and debugging tool [41]. It represents a solution with wide general-purpose adoption, often appearing in industry software development workflows [44]. We test PyVEX v9.0.4663 [32], the modified variant used in the Angr symbolic execution toolkit and countless other open-source security tools.

PCODE [30], our second selection, underpins Ghidra—a reverse engineering application released by the NSA in 2019 and quickly gaining community traction as an alternative to paid, closed-source solutions. PCODE represents a government-generated technology developed in largely in isolation, though its roots trace back to academic work [35]. Despite being relatively new to the public, we have already seen community support for lifting new processors via PCODE's underlying SLEIGH specification format. We test PyPCODE v0.0.2 [31] and use SLEIGH definitions from Ghidra 9.2.

BAP [4], our final selection, powered the winner for DARPA's 2016 Cyber Grand Challenge [9]. BAP represents an academia-generated solution, widely citied in

TABLE I
Userspace Call/Return Semantic Detection Rates by IR

IR	DCall TP	IDCalls	Rets	Fails
VEX (x86)	637	75	342	0
PCODE (x86)	635	85	342	0
BAP (x86)	637	75	342	0
VEX (x64)	474	47	333	0
PCODE (x64)	396	45	333	1323
BAP (x64)	474	47	333	0
VEX (ARM)	532	58	338	0
PCODE (ARM)	529	58	338	0
BAP (ARM)	529	0	338	0
VEX (MIPS)	252	608	526	9
PCODE (MIPS)	244	608	531	0
BAP (MIPS)	DNF	DNF	DNF	DNF

TABLE II
Kernelspace Call/Return Semantic Detection Rates by IR

IR	DCall TP	IDCalls	Rets	Fails
VEX (x86)	3631	0	1922	4
PCODE (x86)	3644	0	1925	0
BAP (x86)	DNF	DNF	DNF	DNF
VEX (x64)	4448	0	2504	9
PCODE (x64)	4172	6	2501	4418
BAP (x64)	DNF	DNF	DNF	DNF
VEX (ARM)	2119	158	1490	0
PCODE (ARM)	2117	158	1484	0
BAP (ARM)	DNF	DNF	DNF	DNF
VEX (MIPS)	2635	212	1581	236
PCODE (MIPS)	2614	238	1590	0
BAP (MIPS)	DNF	DNF	DNF	DNF

relevant research. Though BAP does not enjoy a level of community adoption comparable to our other two IRs, it remains actively developed. We test BAP's 2.2.0 Debian release along with the 1.3.1 pip package — the latter subprocesses the former.

In the interest of reproducibility, we opt not to evaluate closed-source, commercial IRs that we cannot redistribute—notably Binary Ninja's LLIL and MLIL [2].

4) Experiment Results: Testing is performed on a Linux guest executing a sample binary (GNU coreutils program whoami). For all supported architectures, we leverage PyPANDA's record and replay functionality to make all results fully reproducible. For each IR, we measure ability to detect call and return semantics for both userspace and kernelspace code as shown in Tables I, II.

DCall TP is the number of direct call true positives found. IDCalls is the number of indirect calls found (we do not count these against a static IR, which has no way to determine call target). Rets is the number of returns found. Fails is the number of times an IR failed to disassemble a basic block, meaning either an exception occurred or no statements were produced. The first three metrics are for unique basic blocks, meaning we do not count multiple loop iterations—each block is lifted only once and the results are cached. Failure count does not represent unique blocks: it is a raw total that can increase with loop iterations.

These data indicate that VEX is the most accurate and robust IR of the three, particularly for x64 lifting. PCODE performs slightly better in x86 kernel space due to support for privileged instructions missing from the other IRs (e.g. sysexit) and better ability to resolve call encodings with high destination addresses. Where DNF appears for BAP, our analysis run did not finish due to a Python subprocess API, Popen, hanging indefinitely when attempting to run the BAP binary. We checked a random selection of these failures and found that the BAP binary produced correct output when run manually via a terminal. Unfortunately, this means our results for BAP are inconclusive — although it's correctness is comparable to that of VEX in analyses which finish.

In addition to lift accuracy measures, we consider lift performance. VEX is consistently the fastest of the three ($\approx$0.3 ms/block), lifting blocks at approximately 1.25x the speed of PCODE ($\approx$0.4 ms/block) and 20x the speed of BAP ($\approx$7.0 ms/block). BAP's relative sluggishness is the result of Python interoperability, not the IR itself: BAP's Python bindings subprocess an OCaml binary and collect it's output over a pipe. By contrast, VEX and PCODE's Python bindings call into C/C++ libraries via CFFI.

PCODE has a considerable setup cost, taking about 800 ms to parse a SLEIGH specification and initialize the lifter. We chose to exclude this setup time from our measurements because it is a one-time initialization step and does not factor into how long it takes to subsequently lift basic blocks, whereas we are concerned with measuring per-block performance. It is worth noting that this initialization strategy gives PCODE a potential advantage: the core library does not need to be re-compiled to support new architectures, as the lifter is dynamically configured using the SLEIGH specification file.

B. Heap Monitoring

Many dynamic analysis tasks depend on the ability to track function calls, arguments, and return values from specific library or kernel functions. For example, Valgrind [41], Dr. Memory [3], and other systems monitor calls related to heap allocations (e.g., malloc, free, ___kmalloc, kfree, etc.) in order to identify memory errors. Such analyses have been traditionally been built by modifying compilers (e.g., Purify [37]), preloading libraries at runtime (e.g., Malt [45]), or through virtual machine introspection (e.g., Undangle [6]).

We implement a heap tracking analysis using PyPANDA in under 200 lines of Python code. The script hooks kernel heap allocation made with ___kmalloc and freed with kfree as well as userspace heap allocations made with malloc and freed with free through libc. To simplify this example, we do not attempt to track all possible heap allocations (e.g., those made with calloc).

The key components of PyPANDA that enable this analysis are its ability to drive guest execution, hook specific guest program counters to trigger callback functions, and its ability to extract process-level information using PANDA.

1) Hook Setup: PANDA's hooks plugin allow for registering a function to run before the guest executes an instruction a given program counter. To begin this analysis, the script must first identify the addresses that must be hooked. The address of kfree, __kmalloc and vmalloc can be extracted from the System.map file using standard Linux utilities controlled by PyPANDA's run_serial_cmd function. These addresses are in the kernel and will not change while data are collected.

To record the necessary information, the requested size must be recorded when __kmalloc and vmalloc are entered and the return values must be recorded when they returns. This script includes a simple helper function to store this information and then call a user-provided function when the function returns:

```
def hook_ret_with_args(panda, name, entry_addr, func=None,
                                  asid=None, kernel=False):
  hooked_args = []

  @panda.hook(entry_addr, asid=asid, kernel=kernel)
  def _enter(cpu, tb, h):
    # Grab ret_addr off stack and first two args
    sp = panda.arch.get_reg(cpu, 'rsp')
    ret_addr = panda.virtual_memory_read(cpu, sp, 8, fmt='int')
    hooked_args = [panda.arch.get_arg(cpu, i) for i in range(2)]

    # Setup a new hook to run just once at ret_addr
    # and call user-provided func
    @panda.hook(ret_addr, asid=asid, kernel=kernel)
    def _return(cpu, tb, h):
      h.enabled = False
      retval = panda.arch.get_reg(cpu, 'rax')
      func(cpu, hooked_args, retval)
```

The script also needs to hook libc functions for tracking userspace heap behavior. Like with __kmalloc, the analysis for malloc requires tracking both the size passed into the function as well as the pointer that is returned. The hook_ret_with_args function can be used again to do this.

The script captures the offsets into libc to malloc and free using the run_serial_cmd function again. Since libc can be loaded at different base addresses for each process, the hooked address for each process must be the libc base address plus the offsets of these functions.

2) Process Hooking: Whenever a process may load libc into memory, we must check if it was loaded and, if so, add a new set of hooks for that process. We may discover libc has been loaded after a context switch or after a process uses the mmap or brk syscall to load a library. The script registers PPP-style callbacks for these events with the OSI and syscalls plugins and calls add_hooks_if_necessary after each. At each callback, we check if the current process lacks hooks, and if so use the OSI plugin to determine if a library named libc has been loaded into its memory space. If so, it calculates the address of malloc and free and adds the relevant hooks.

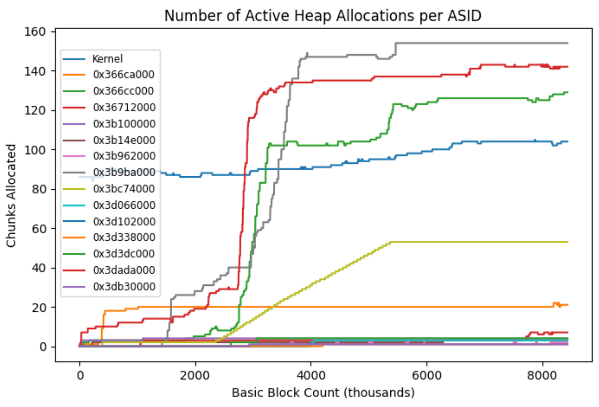
```
def add_hooks_if_necessary(cpu):
  if not analysis_active: return # Don't setup hooks until we're ready

  asid = panda.current_asid(cpu)
  if asid in hooked_asids: return # Already hooked this process

  name = panda.get_process_name(cpu)
  # Find current libc address and update hooks
  for mapping in panda.get_mappings(cpu):
    if mapping.file != panda.ffi.NULL and \
        b'/libc-' in panda.ffi.string(mapping.file):
      hooked_asids.add(asid)
      hook_ret_with_args(panda, f'{name}_malloc', mapping.base +
          malloc_offset, asid=asid,
          func=lambda cpu, in_args, retval: add_alloc(retval, in_args[0],
              asid=panda.current_asid(cpu),
              name=panda.get_process_name(cpu)))

      # Note for free we don't need return val, just the entry args
      @panda.hook(mapping.base+free_offset, asid=asid, kernel=False)
      def process_free(cpu, tb, h):
        buf = panda.arch.get_reg(cpu, 'rax')
        asid = panda.current_asid(cpu)
        rem_alloc(buf, asid=asid)
```

3) Allocation Tracking: With the hooks set up, the implementation of heap allocation tracking is trivial: when objects are allocated, their size, address, and requesting ASID is recorded in a dictionary. Kernel allocations are indicated by storing an ASID of 0. When objects are freed they are deleted from the dictionary using the ASID and address.

4) Guest Control: After all the hooking and allocation tracking logic is configured, a binary is copied into the guest and executed. During its execution, the hooks are enabled and a callback is setup to report the current allocations every 1,000 basic blocks.

5) Visualization: The recorded data are visualized through the Python library matplotlib [17] as shown in Figure 3. The total number of active heap allocations is presented for each ASID running in the guest system.

Fig. 3. Visualizing heap objects allocated over time.



6) Summary: This section describes building a standalone dynamic analysis tool for heap analysis with PyPANDA. The hook_ret_with_args helper function

demonstrates how a complex functions not supported by the PANDA API can easily be implemented in PyPANDA. The entire analysis is built in under 200 lines of Python code and runs with a wall time of 17.24s on average (n=100).

C. Extracting and Transferring Packed Executables to Ghidra with human-in-the-loop Entropy Analysis

A common technique seen in malicious software is runtime packing. In malware, runtime packing is used as a means of obfuscating the binary to make it more difficult to identify it as malicious [19]. This is typically accomplished by providing a binary to a "packer" program. The packer program takes the binary and either compresses or encrypts it and produces a new program with the binary data. When run this program takes the encrypted or compressed binary data from its data section, reverses the operation applied to the data, and then transfers control to the original program. This has the effect of making reverse engineering the program statically very difficult. However, the process of packing leaves a mark: packed regions typically have high entropy [23]. This characteristic can be used to make an educated guess as to whether or not a binary may be packed. PANDA is uniquely suited to analyze malware as its position in the hypervisor makes it more difficult to detect and its record and replay features allow for reproducibility of functionality.

In this section we will discuss a PyPANDA script which records a whole-system running a packed binary. The script is given a process name to consider and computes the entropy of all segments of memory throughout the life of the process. As it performs this analysis, the script spawns a Python Flask[7] web server which serves live-updating graphs the entropy of the process memory. If a user is interested in looking at the program state at some point along the graph they may click on the point. The Flask server will receive their selection and spawn a new process which replays the recording up to the point of interest, extracts all mapped virtual memory in the process, and transfers it to static analysis platform Ghidra through the use of ghidra_bridge[8], It does all of this, including generating the recording, in fewer than 250 lines of Python.

1) Entropy Analysis:

Entropy analysis is quite simple in PyPANDA. We begin by identifying the ASID for the process of interest by registering a callback on asid_changed. After each asid change, we check if the name of the current process matches the provided name.

Once the ASID is identified, the entropy analysis can begin. We set a callback on before_block_exec which first cheks that the correct ASID is running and the guest is in userspace. We provide an tunable parameter, granularity, which controls how frequently we compute the process' entropy. When it is time to compute entropy, the script reads all memory in the process by iterating over the mappings provided by the OSI plugin. We then format the data, compute a value for the entropy using the scipy library [46], and record it along with the current instruction count.

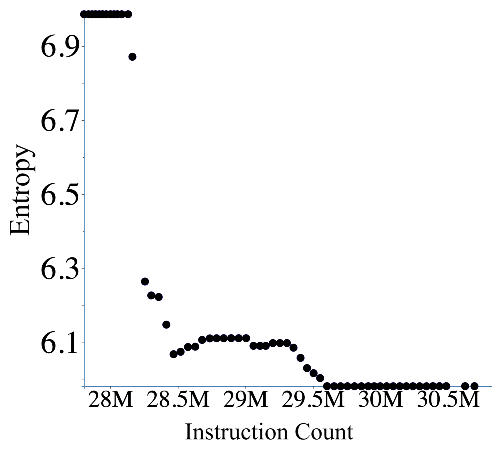
[7]https://flask.palletsprojects.com
[8]https://github.com/justfoxing/ghidra_bridge

2) Flask server:

In order to display this data to the analyst we use the Flask Python module. The Flask server runs using multiprocessing with a dedicated thread per connection to enable live-updating of entropy visualization as new data become available. When a point on the graph is clicked, the Flask server starts a new process to transition the program state at that point in time into Ghidra.

Figure 4 shows this graph for a binary packed by UPX [28] running. After unpacking, the binary reaches a steady state and performs normal operations.

Fig. 4. Entropy graph of a packed process unpacking itself and running as seen through Flask.



3) Transitioning to Ghidra:

Our transition to Ghidra begins by establishing a connection to ghidra_bridge. We do this through the use of ghidra_bridge, a Python 3 module for Remote Procedure Calls (RPC) to Ghidra using their Jython interface. When we establish a connection through ghidra_bridge we declare the namespace to be globals(). This populates our main namespace with RPC variables and allows us to do scripting with Ghidra in a very similar manner to normal Ghidra scripting. Next, we run our replay until we reach the instruction count selected by the user. At this point we can to move our program's virtual memory to Ghidra. This is a fairly simple process. In PyPANDA, we simply use our OSI capabilities to list the virtual memory mappings for our program and read them. In Ghidra, we establish a memory transaction, create a memory segment per virtual memory mapping, and then populate it with our data. Lastly, we take the program counter at this point in time and set cursor position in Ghidra to that program counter and perform an auto-analysis of the program.

TABLE III
Runtime of PANDA vs PyPANDA analyses, averaged (n=10).

Replay	Analysis	Total Callbacks Run	PANDA Runtime	PyPANDA Runtime	% Difference
grep (4M insns.)	no-op	0	$0.88s \pm 0.03$	$0.93s \pm 0.05$	$6\% \pm 7$
	asid_logger	14	$0.86s \pm 0.01$	$0.93s \pm 0.03$	$8\% \pm 4$
	unique_bbs	0.94M	$0.92s \pm 0.03$	$1.40s \pm 0.03$	$53\% \pm 6$
wget (8M insns.)	no-op	0	$1.02s \pm 0.02$	$1.08s \pm 0.04$	$5\% \pm 5$
	asid_logger	21	$1.04s \pm 0.03$	$1.08s \pm 0.03$	$4\% \pm 4$
	unique_bbs	1.83M	$1.10s \pm 0.03$	$2.02s \pm 0.03$	$83\% \pm 5$
sleep (426M insns.)	no-op	0	$8.20s \pm 0.16$	$8.28s \pm 0.25$	$1\% \pm 4$
	asid_logger	108	$8.36s \pm 0.24$	$8.23s \pm 0.23$	$2\% \pm 4$
	unique_bbs	89.45M	$13.46s \pm 0.24$	$51.67s \pm 0.35$	$284\% \pm 7$

TABLE IV
Source Lines of Code for Simple Analyses

Analysis	PANDA (C++)	PyPANDA
unique_bbs	20	9
asid_logger	18	8

4) Summary: In this section we described a PyPANDA script which performs graphs the entropy of a binary in real time and allows a user to transition program state from any point in time into Ghidra for subsequent analysis. This integration between Ghidra and PANDA allows PyPANDA users to leverage Ghidra's decompiler and its analyses. PyPANDA enables such integrations by easily synchronizing state between PANDA and other projects with Python interfaces. This integration between a disprate set of tools shows the value of using Python for a scripting language. This analysis was built in under 250 lines of Python code.

D. Performance comparision of PyPANDA to PANDA

Unfortunately, the case studies presented are too complex to reasonably reimplement without PyPANDA. To compare PyPANDA's performance to that of PANDA, we developed two simple plugins in both PANDA and PyPANDA. The first plugin, unique_bbs, tracks every distinct program counter executed by storing it in a set. At the end of the execution, the size of the set is printed. The second, asid_logger, reports every time the ASID changes by printing out the new ASID. The lines of source code for each plugin are presented in Table IV.

Three PANDA recordings were captured on an i386 guest (using PyPANDA for convenience):

1) wget downloading from example.com.
2) grep searching through /etc/passwd.
3) sleep running for 20 seconds.

For each recording, we ran both the PANDA plugin and the PyPANDA plugin 10 times and recorded the time taken[9]. We also run each recording through PANDA and PyPANDA with no analyses enabled (labeled no-op). The average run time along with standard deviation for each plugin is presented in Table III.

[9]We use the Linux time utility and combine the CPU time with the kernel time to capture how long the process was actually executing.

Unsurprisingly, PyPANDA runs slower than compiled PANDA plugins. There are additional costs when running PyPANDA: a fixed startup cost to launching a Python interpreter, loading the Python module, and a cost for each time PANDA switches into PyPANDA (i.e., in a callback). It is this last cost which is the most significant to consider when developing PyPANDA scripts. As can be seen in Table III the no-op and asid_logger analyses have near-identical run times to native C. Only when the number of callbacks run increases dramatically do we see a significant performance cost. We believe this performance is sufficient to enable many analyses to be completed in PyPANDA.

When high-speed analyses are desired (i.e., for algorithmically complex analyses of live systems) we tend to consider two avenues. First, a PyPANDA prototype could be developed and then transitioned into C or C++ to meet specific performance needs. Second, move more speed-critical portions of code into a C or C++ plugin. We have found that for many callbacks there is a series of conditions (e.g. not in kernel, in specific process, etc.) for the machine which must be met before complex analysis is done. Writing a simple C plugin which handle this more primitive initial checking before calling into Python can lead to drastic performance improvements. For examples of plugins of this type consider hooks, which calls a callback when a program counter reaches a particular value and mem_hooks which calls a callback when a memory transactions takes place that meet specified criteria.

VII. Discussion

Having successfully used PyPANDA to enable three distinct dynamic analyses, we now consider if PyPANDA met our design goals.

1) High Performance Scripting: PyPANDA's meets our goal of enabling high performance scripting as shown in §VI-D. The core runtime overhead of PyPANDA with few callbacks enabled is consistently below 10%. However, there is a slight performance difference between callbacks implemented in C/C++ code vs Python code. Generally, compiled code will execute faster than Python code implementing the same behaviors. As the complexity of the logic run in a callback increases or the number of times callbacks are run increases, the PyPANDA overhead will grow more

pronounced. As previously shown in Table III, we see an overhead as high as 284% when running nearly a simple callback 90 million callbacks, but much lower overhead for callbacks that run less frequently. This performance trade-off fits well within our objective.

2) Unified Perspective: We sought to build a system to unify the interfaces to interact with a guest and to control its analysis. This is a shift in perspective from the plugin system normally used for analysis with PANDA. Traditionally PANDA is run from the command line with various arguments which dictate the plugins to be loaded and their various settings. Instead of compiling in an interpreter to a plugin in PANDA we intentionally inverted the normal control structure of a plugin. We accomplished this through compiling PANDA as a shared object. In doing so we made PyPANDA encapsulate PANDA and provided a path for other languages to encapsulate PANDA as well through our PANDA API. In PyPANDA the process begins with a Python script executed which sets up an emulated machine, various plugins and their arguments, its own python callbacks, and then chooses to either run the machine or a replay. PyPANDA maintains the ability to interact with and control the emulated machine through a separate thread which can interact with various PANDA interfaces including the PANDA monitor and the serial console. In doing so we have provided a reproducible mechanism to simultaneously control guest behavior and analyze it from a single script.

3) Ease of Use: Through building PyPANDA, we took several steps to improve the usability of PANDA. The largest change is that we provided "generic" QCOW images for every PANDA supported architecture which are automatically downloaded along with a PANDA OSI profile prior to use. All the required settings to run each generic image are built into PyPANDA so only the image's name must be specified by a user. Another significant improvement is that we created a docker container and Python package for PyPANDA that can be installed through the Python package manager, pip.

4) Integrations: A goal of this project was to enable integrations between PANDA other software projects. By selecting Python as our scripting language, we found a significant number of relevant projects that could be easily imported. As described in § VI, VEX, PyPCODE, Flask, and Ghidra were all seamlessly integrated with PyPANDA to enable program analyses.

5) Access to Internal PANDA Data Structures: By using CFFI with the PANDA shared libraries and autogenerated headerfiles, PyPANDA can present parsed structures as Python objects. This allows for nested structure access as well as easy modification of structure elements by name.

VIII. Related Work

There are many projects which are closely related to PyPANDA. Other emulators have Python bindings to enable program analysis but focus on execution of shellcode of single binaries as opposed to a whole system. For example, the Unicorn Engine [34] is another QEMU fork with Python bindings capable of running shellcode from many architectures. The Qiling framework [33, 20] is a binary emulation framework built on top of Unicorn with Python bindings which supports running individual binaries in an analysis platform, but not a full operating system.

Many dynamic analysis tools have been created with Python interfaces. Frida [36] injects custom hooking and tracing logic into compiled binaries using a Python interface. Angr [47] is a Python-based framework that enables static analyses of binaries and symbolic execution. Recent advances to Angr [14] allow transfering state to and from concrete execution environments (e.g., QEMU) which enables transitioning between concrete and symbolic executions.

Volatility [12] and Libvmi [49] enable virtual machine introspection from Python which is akin to PANDA's OSI plugin. Both systems have OS-specific profiles which are used to bridge the semantic gap [18] to provide meaningful information about an operating system running in an emulator.

avatar^2 [26] is an orchestration platform designed to transfer the state of a whole system between different analysis platforms or a physical system. It initially supported transferring state into and out of PANDA as well as limited interactions with both the guest through PANDA's gdbstub as well as PANDA itself (loading plugins, recording/replaying) through the PANDA monitor. Recent updates to avatar^2 have added support for a "PyPANDA target" which enables avatar^2 scripts to use the PANDA API and callback system. Through this interface, the avatar^2 developers used PyPANDA's hooks plugin for what they describe as a "huge speedup" as compared to avatar^2's traditional, GDB-based hook system [16].

Another related project to PyPANDA is PyREBox, a dynamic analysis system built on top of QEMU with Python-based callbacks. Although similar on the surface, there are three significant differences between PyREBox and PyPANDA. First, PyREBox is built on QEMU so it cannot leverage PANDA's plugin system, callbacks, nor its record/replay system. Beyond this, PyREBox is implemented by embedding a Python interpreter within the emulator. As previously discussed in §III, this apparoch is limiting as it prevents both our unified perspective of dynamic analysis and access to emulator internal structures. Finally, PyREBox currently supports only Python 2 for scripting and is limited to the x86 and x86_64 architectures.

IX. Conclusions

Through this work, we identified that existing whole-system dynamic analysis platforms largely lack scripting interfaces, and have a split view of guest behavior and analyses. We designed PyPANDA to provide a Python interface to the PANDA dynamic analysis platform and to unify the analysts view of a system. Using PyPANDA, we describe the process of implementing 3 distinct whole-system dynamic analyses: evaluating binary analysis frameworks, tracking heap allocations in all processes, and dynamically moving program state into the Ghidra reverse engineering framework. We examine the analyses and discuss how PyPANDA achieved the requisite goals. PyPANDA has been merged into the PANDA project and is now publicly available at https://github.com/panda-re/panda. We hope this platform will make the field of dynamic analysis more welcoming for beginners while allowing experts to conduct complex analyses and integrations.

Acknowledgements

The authors wish to acknowledge the following individuals for their contributions and support: Chris Connelly, Marius Muench, Wesley Van Pelt, Andrew Mueller, and Michael Rusnak.

This research was supported in part by National Science Foundation (NSF) Award 1657199. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the NSF.

References

[1] Fabrice Bellard. “QEMU, a fast and portable dynamic translator.” In: USENIX Annual Technical Conference, FREENIX Track. Vol. 41. 2005, p. 46.

[2] Binary Ninja Intermediate Language Series, Part 0: Overview. url: https://docs.binary.ninja/dev/bnil-overview.html.

[3] Derek Bruening and Qin Zhao. “Practical memory checking with Dr. Memory”. In: International Symposium on Code Generation and Optimization (CGO 2011). IEEE. 2011, pp. 213–223.

[4] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. “BAP: A Binary Analysis Platform”. In: Proceedings of the 23rd International Conference on Computer Aided Verification. CAV’11. Snowbird, UT: Springer-Verlag, 2011, pp. 463–469. isbn: 9783642221095.

[5] Prashanth P. Bungale and Chi-Keung Luk. “PinOS: A Programmable Framework for Whole-System Dynamic Instrumentation”. In: Proceedings of the 3rd International Conference on Virtual Execution Environments. VEE ’07. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 137–147. isbn: 9781595936301. doi: 10.1145/1254810.1254830. url: https://doi.org/10.1145/1254810.1254830.

[6] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. “Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities”. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. 2012, pp. 133–143.

[7] Cisco-Talos. Cisco-Talos/pyrebox: Python scriptable Reverse Engineering Sandbox, a Virtual Machine instrumentation and inspection framework based on QEMU. url: https://talosintelligence.com/pyrebox.

[8] Couverture. url: http://www.open-do.org/projects/couverture/.

[9] Cyber Grand Challenge (CGC). url: https://www.darpa.mil/program/cyber-grand-challenge.

[10] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. “Repeatable reverse engineering with PANDA”. In: Proceedings of the 5th Program Protection and Reverse Engineering Workshop. 2015, pp. 1–11.

[11] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. “Virtuoso: Narrowing the semantic gap in virtual machine introspection”. In: 2011 IEEE symposium on security and privacy. IEEE. 2011, pp. 297–312.

[12] The Volatility Foundation. Volatility. url: https://www.volatilityfoundation.org/.

[13] Isaac Gouy and B Fulgham. The computer language benchmarks game. 2017. url: https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html.

[14] Fabio Gritti, Lorenzo Fontana, Eric Gustafson, Fabio Pagani, Andrea Continella, Christopher Kruegel, and Giovanni Vigna. “SYMBION: Interleaving Symbolic with Concrete Execution”. In: 2020 IEEE Conference on Communications and Network Security (CNS). IEEE. 2020, pp. 1–10.

[15] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. “Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform”. In: Proceedings

of the 2014 International Symposium on Software Testing and Analysis. 2014, pp. 248–258.

[16] Grant Hernandez, Marius Muench, Tyler Tucker, Hunter Serle, Weidong Zhu, Patrick Traynor, and Kevin Butler. Emulating Samsung's Baseband for Security Testing. Blackhat. 2020.

[17] John D Hunter. "Matplotlib: A 2D graphics environment". In: Computing in science & engineering 9.3 (2007), pp. 90–95.

[18] Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E Porter, and Radu Sion. "Sok: Introspections on trust and the semantic gap". In: 2014 IEEE symposium on security and privacy. IEEE. 2014, pp. 605–620.

[19] G. Jeong, E. Choo, J. Lee, M. Bat-Erdene, and H. Lee. "Generic unpacking using entropy analysis". In: 2010 5th International Conference on Malicious and Unwanted Software. 2010, pp. 98–105. doi: 10.1109/MALWARE.2010.5665789.

[20] Lau KaiJern. Qiling Framework. url: https://www.qiling.io/.

[21] Soomin Kim, Markus Faerevaag, Minkyu Jung, SeungIl Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. "Testing Intermediate Representations for Binary Analysis". In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. ASE 2017. Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 353–364. isbn: 9781538626849.

[22] David Korczynski. Building a custom malware sandbox with PANDA - Part 1. 2019. url: https://adalogics.com/blog/Building-a-custom-malware-sandbox-with-PANDA-Part-1.

[23] Robert Lyda and James Hamrock. "Using Entropy Analysis to Find Encrypted and Packed Malware". In: IEEE Security and Privacy 5.2 (Mar. 2007), pp. 40–45. issn: 1540-7993. doi: 10.1109/MSP.2007.48. url: https://doi.org/10.1109/MSP.2007.48.

[24] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. "Testing System Virtual Machines". In: Proceedings of the 19th International Symposium on Software Testing and Analysis. ISSTA '10. Trento, Italy: Association for Computing Machinery, 2010, pp. 171–182. isbn: 9781605588230. doi: 10.1145/1831708.1831730. url: https://doi.org/10.1145/1831708.1831730.

[25] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts". In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE. 2019, pp. 1186–1189.

[26] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. "Avatar2: A multi-target orchestration platform". In: Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.) Vol. 18. 2018, pp. 1–11.

[27] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices." In: NDSS. 2018.

[28] Markus FXJ Oberhumer. UPX the Ultimate Packer for eXecutables. 2004. url: https://upx.github.io/.

[29] John K Ousterhout. "Scripting: Higher level programming for the 21st century". In: Computer 31.3 (1998), pp. 23–30.

[30] P-Code Reference Manual. url: https://ghidra.re/courses/languages/html/pcoderef.html.

[31] PyPCODE. url: https://github.com/angr/pypcode.

[32] PyVEX. url: https://github.com/angr/pyvex.

[33] Nguyen Anh Quynh and Lau KaiJern. QiLing. 2019.

[34] NGUYEN Anh Quynh and DANG Hoang Vu. "Unicorn: Next generation cpu emulator framework". In: BlackHat USA (2015).

[35] Norman Ramsey and Mary F. Fernandez. "Specifying Representations of Machine Instructions". In: ACM Trans. Program. Lang. Syst. 19.3 (May 1997), pp. 492–524. issn: 0164-0925. doi: 10.1145/256167.256225. url: https://doi.org/10.1145/256167.256225.

[36] Ole Andrè Ravnas. Frida. url: https://www.frida.re/.

[37] Bob Joyce Reed Hastings. "Purify: Fast detection of memory leaks and access errors". In: In proc. of the winter 1992 usenix conference. Citeseer. 1991.

[38] Goldwyn Rodrigues. Poke-a-hole and friends. June 2009. url: https://lwn.net/Articles/335942/.

[39] Triton: A Dynamic Symbolic Execution Framework. SSTIC, 2015, pp. 31–54.

[40] Giorgio Severi, Tim Leek, and Brendan Dolan-Gavitt. "Malrec: compact full-trace malware recording for retrospective deep analysis". In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer. 2018, pp. 3–23.

[41] Julian Seward and Nicholas Nethercote. "Using Valgrind to Detect Undefined Value Errors with Bit-Precision." In: USENIX Annual Technical Conference, General Track. 2005, pp. 17–30.

[42] Manolis Stamatogiannakis, Herbert Bos, and Paul Groth. "PANDAcap: a framework for streamlining collection of full-system traces". In: Proceedings of the 13th European workshop on Systems Security. 2020, pp. 1–6.

[43] Frederick Ulrich. "Exploitability Assessment with TEASER". MA thesis. Northeastern University, 2017.

[44] Using Valgrind to troubleshoot possible ndsd memory leaks. url: https://support.microfocus.com/kb/doc.php?id=7005905.

[45] Sébastien Valat, Andres S Charif-Rubial, and William Jalby. "MALT: a Malloc tracker". In: Pro-

ceedings of the 4th ACM SIGPLAN International Workshop on Software Engineering for Parallel Systems. 2017, pp. 1–10.

[46] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: Nature Methods 17 (2020), pp. 261–272. doi: 10.1038/s41592-019-0686-2.

[47] Fish Wang and Yan Shoshitaishvili. "Angr-the next generation of binary analysis". In: 2017 IEEE Cybersecurity Development (SecDev). IEEE. 2017, pp. 8–9.

[48] David Wheeler. SLOCCount. 2001. url: http://www.dwheeler.com/sloccount/.

[49] Haiquan Xiong, Zhiyong Liu, Weizhi Xu, and Shuai Jiao. "Libvmi: a library for bridging the semantic gap between guest OS and VMM". In: 2012 IEEE 12th International Conference on Computer and Information Technology. IEEE. 2012, pp. 549–556.