Avatar^2: A Multi-target Orchestration Platform

Marius Muench, Dario Nisi, Aurélien Francillon and Davide Balzarotti
EURECOM, France
{muench, nisi, francill, balzarot}@eurecom.fr

***Abstract*—Dynamic binary analysis techniques play a central role to study the security of software systems and detect vulnerabilities in a broad range of devices and applications. Over the past decade, a variety of different techniques have been published, often alongside the release of prototype tools to demonstrate their effectiveness. Unfortunately, most of those techniques' implementations are deeply coupled with their dynamic analysis frameworks and are not easy to integrate in other frameworks. Those frameworks are not designed to expose their internal state or their results to other components. This prevents analysts from being able to combine together different tools to exploit their strengths and tackle complex problems which requires a combination of sophisticated techniques. Fragmentation and isolation are two important problems which too often results in duplicated efforts or in multiple equivalent solutions for the same problem – each based on a different programming language, abstraction model, or execution environment.**

In this paper, we present `avatar`^{2}, a dynamic multi-target orchestration framework designed to enable interoperability between different dynamic binary analysis frameworks, debuggers, emulators, and real physical devices. `Avatar`^{2} allows the analyst to organize different tools in a complex topology and then "move" the execution of binary code from one system to the other. The framework supports the automated transfer of the internal state of the device/application, as well as the configurable forwarding of input/output and memory accesses to physical peripherals or emulated targets.

To demonstrate `avatar`^{2} usage and versatility, in this paper we present three very different use cases in which we replicate a PLC rootkit presented at NDSS 2017, we test Firefox combining Angr and GDB, and we record the execution of an embedded device firmware using PANDA and OpenOCD. All tools and the three use cases will be released as open source to help other researchers to replicate our experiments and perform their own analysis tasks with `avatar`^{2}.

I. Introduction

Our societies are increasingly depending on computing systems, from low end embedded devices to large cloud-based systems. These systems are used in very diverse areas ranging from complex industrial settings to smaller consumer devices. However, the significant number of software vulnerabilities that are regularly discovered in all computing systems is making this dependency a significant issue. The software running on those systems therefore needs to be carefully tested and verified, which is commonly done not only by the manufacturers themselves, but also by third party companies and security researchers. While an important body of work exists on analyzing programs from the source code, binary-only analysis is very popular among software authors and third parties alike. The motivation for third parties to engage in this activity are diverse, varying from consultant contracted to perform additional black box testing, individuals searching for security vulnerabilities as part of bug bounty programs, or researchers just generally interested in reducing the number of bugs in popular software products.

Binary-only testing is also a very popular option when source code is available to the analyst or when the software is available as open source. This is the case as some subtle bugs cannot be discovered by source code inspection only and because binary analysis allows to test software independently from the source code language that was used to develop the system. Moreover, not relying on the source code means the analysis does not depend on the compiler to be correct, i.e., "what you fuzz is what you ship" [1].

As a result, over the past decade, testing and analyzing software with binary-only approaches has become more and more popular and a variety of different techniques have been published for binary analysis [23]. It is often the case that these techniques are implemented in independent tools to demonstrate their effectiveness. While some tools are simple prototypes, others are more mature solutions that have earned a significant popularity over the years. Existing tools may rely on testing to be performed directly on the target analysis host, within an emulator, or even inside a dedicated specific execution engine (e.g., for symbolic execution).

Unfortunately, most of these techniques' implementations are deeply coupled with their dynamic analysis frameworks and are not easy to integrate into other frameworks. Quite often, they even re-implement binary analysis techniques part of other tools, to benefit from those techniques, as every framework aims to obtain the best analysis possible. Unfortunately, little effort has been invested so far towards a better interaction between different frameworks, and not only in terms of re-using analysis results, but also by sharing the internal analysis state to external components. This prevents analysts from being able to combine together different tools to exploit their strengths and tackle complex problems which requires a combination of sophisticated techniques. Fragmentation and isolation are two important problems which too often results in duplicated efforts or in multiple equivalent solutions for the same problem – each based on a different programming language, abstraction model, or execution environment.

Workshop on Binary Analysis Research (BAR) 2018
18th February 2018, San Diego, CA, USA

In this paper we present avatar^2, a framework we developed to facilitate the interoperability among multiple binary analysis tools, such as debuggers, symbolic execution engines, and emulators. The basic principle behind our solution is to let the analyst choose multiple execution environments, which are later used to perform their tasks on the *same* execution state. Indeed, we will show in this paper that there are a variety of scenarios in which it is beneficial to keep a specific analysis not only local to one tool, but using its state as a basic block inside other analysis environments. Avatar^2 is a successor of Avatar [29], a system originally designed to perform embedded devices analysis, which we completely re-designed and extended to allow an easy orchestration of arbitrary components that can be combined to perform sophisticated binary analysis tasks.

We believe avatar^2 is the first attempt to provide a generic framework for such *multi-target orchestration*. In particular, this paper makes the following contributions to the area of binary analysis:

- We show that state forwarding between analysis environments is not only useful for embedded devices analysis but is a more generic technique that allows orchestration among multiple environments.
- We present avatar^2, a completely redesigned system for orchestrating executions between multiple testing environments.
- We show how avatar^2 scripting can help to quickly replicate previous research and at the same time improve repeatability of research performed using avatar^2.
- We demonstrate avatar^2 usage and versatility with three very different use cases in which we replicate previous firmware research, we record and replay the execution of an embedded device, and we analyze a large desktop application.
- We highlight the benefits of orchestrating the execution across multiple tools: GDB, OpenOCD, QEMU, angr and PANDA.
- We release all tools and the three use cases as open source[1] to help other researchers to replicate our experiments and perform their own analysis tasks with avatar^2[2].

The rest of the paper is structured as follows. First, in Section II we provide a general background on program analysis, and summarize the original Avatar [29], the predecessor of avatar^2. Afterwards, we present the avatar^2 framework (Section III) and then show its usefulness with three case studies (Section IV) highlighting interesting aspects of the framework. We then discuss the results (Section V) compare avatar^2 with related work (Section VI) and close the paper with concluding remarks.

II. Background

Before describing the avatar^2 framework, we provide some background information on the topic of dynamic binary analysis, to emphasize the need for multi-target orchestration. Afterwards, we briefly discuss the original Avatar framework, the predecessor which served as inspiration for avatar^2. Although the purpose of the original tool was solely to enable dynamic binary analysis for embedded devices by connecting S2E [5] to a physical device, it introduced a number of important concepts for building an orchestration framework suitable for coordinating multiple dynamic binary analysis tools.

A. Dynamic Binary Analysis

Program analysis is commonly divided in *static* and *dynamic* analysis. Static analysis reasons about the program just by examining its code, whereas dynamic analysis relies on the actual execution of the program. Therefore, static analysis can provide sound results by analyzing all possible execution path of a program.

However, with static analysis the whole runtime state of the program is not available (e.g., heap structures, some pointers or threading) and approximations are often needed to decide otherwise undecidable problems, which may lead to a high number of false positives. Dynamic analysis avoids those problems by analyzing the actual behavior of a program while it executes on a given input (at the price of being able to observe only a small portion of the program state and code).

One common application of dynamic analysis is security testing and evaluation, which aims at discovering bugs that impact the security of the program under analysis. Common examples of those bugs are authentication bypasses, memory corruptions, or memory disclosure, which can all be beneficial to a potential attacker. In this context, binary program analysis, which is based on machine code only, is especially important for two reasons. First, binary code is the most accurate representation of the program as it is the code that is executed directly on the processor. Second, despite the existence of a steadily growing open source movement, a large number of programs are distributed only in binary form. This applies especially to programs written in memory-unsafe languages, such as C and C++, which are an excellent target for attackers, due to their widespread deployment.

In recent years, the state of the art in dynamic binary analysis has largely improved and a variety of tools were developed and made available to assist security testing. The classic approach requires the instrumentation of the binary under examination – either dynamically or statically – before the execution of the program. Valgrind [18] and AddressSanitizer [22] are popular examples of tools following this approach.

Another relevant use of dynamic analysis is in combination with a symbolic execution engine, either as a way to provide concrete inputs to drive the symbolic exploration (in what is traditionally called *concolic execution*) or as a way to execute a program along multiple paths driven by a symbolic exploration and constraints solving. The development of this kind of hybrid engines experienced a renaissance in the last decade and an analyst has to choose between several frameworks, such as angr [24], BAP [2], Manticore [27], Miasm [10], BINSEC/SE [8] or Triton [21].

While each solution has its own strengths and weaknesses, they all share a property: each analysis task is bound to

[1] All code and examples are available at: https://github.com/avatartwo/

[2] We apologize in advance for any confusion between avatar^2 typography and the inevitable footnotes references in superscript.

its own dedicated tool. This is due to a variety of reasons, including incompatible design choices (such as the use of an *intermediate representation* specific to a certain tool or the abstract representation of the program state in a custom format) or the fact that developers often implement tools as standalone systems that are implemented to be flexible to use but are nevertheless difficult to integrate with other solutions. Besides leading to duplication of work, as different tools are often implementing the same dynamic binary analysis techniques independently from each other, this prevents the full potential of binary analysis from being unleashed.

B. Avatar One

Avatar [29] is a dynamic binary analysis framework for embedded devices. In essence, it allowed partial emulation of firmware inside S2E, a symbolic execution engine based on QEMU. To achieve this goal, I/O requests which can not be emulated are forwarded to the actual embedded device, either via dedicated debugging ports or by using a debugging stub manually injected into the device.

While Avatar focused solely on allowing S2E to work on embedded devices, it introduced the following concepts which play an important role for the design of a more general binary dynamic analysis orchestration framework:

- **Target Orchestration.** Avatar introduced the concept of orchestration, not simply as a way to control its two targets (S2E and the physical system), but also as mean to automatically transfer the execution from one tool to the other, based on certain events specified by the analyst.

- **Separation of Execution and Memory.** While the execution of a software and its memory space are tightly linked together in traditional analysis approaches, Avatar decouples them. This, among others, allows the framework to use a so called *remote memory*, whereby the execution proceeds on one target, while memory reads and writes are forwarded to another target. This allowed Avatar to achieve partial emulation, whereas the main firmware is executed in an emulator, while accesses to memory-mapped peripherals are forwarded to the actual device.

- **State transfer and synchronization** Next to the orchestration of execution, Avatar provided the possibility to selectively transfer the state from one of its targets to another, where the state is defined by the combination of the content of the memory as well as the CPU registers. This allowed Avatar to execute initialization functions on the physical device under analysis, before transferring the state to S2E to perform symbolic execution.

III. The avatar^2 framework

As discussed in section II-A, dynamic binary analysis can greatly benefit from the interconnection of existing tools. In this section, we will present avatar^2, the framework we developed to enable flexible dynamic binary analysis by interconnecting debuggers, emulators and other dynamic binary analysis frameworks.

A. General Overview & Terminology

Combining and connecting a variety of distinct tools requires a careful planned design to cope with the inherent challenges arising from the large diversity of tools. For example, such tools often use both asynchronous and synchronous communications.

On an abstract level, avatar^2 consists of four distinct elements, as visualized in Figure 1. The avatar^2 *core*, *targets* and *protocols* are python libraries (available as open source at `https://github.com/avatartwo/avatar2`) while *endpoints* are third-party software (such as other analysis frameworks, emulators, or solutions to talk to physical devices) controlled and interconnected by avatar^2.

The avatar^2 core has three purposes: i) to serve as the main interface for the analyst using the framework, ii) to carry out the actual orchestration and control all the underlying elements, and iii) to catch, dispatch, and react to events generated by the various protocols while communicating with the respective endpoints.

Targets play the role of abstracting each endpoint and providing high-level interfaces to the avatar^2 core. However, these python abstractions do not directly communicate with their associated endpoints. In fact, since the actual communication often requires similar patterns that would otherwise be duplicated in multiple targets, it is mediated by a layer of specific *protocols* objects. This architecture makes individual protocols easy to reuse when prototyping new targets. This is for instance the case for a variety of debuggers and emulators that, while typically equipped with their own communication interface, often incorporate also a `gdbserver`, which can be controlled by gdb's *remote serial protocol*.

The protocols themselves are divided according to their purpose. In most of the cases, a target needs at least a memory protocol, an execution protocol, and a register protocol. These protocols are responsible, respectively, for dispatching memory reads and writes, controlling the execution of the target, and accessing its CPU registers. Additionally, avatar^2 provides the possibility to define additional protocols, such as *monitor* protocols specifically dedicated to monitor the status of an endpoint or specialized *remote memory* protocols that can provide a custom high bandwidth channel for memory accesses from one endpoint to another.

Finally, *endpoints* can be anything worth orchestrating for an analysis, and the initial implementation of avatar^2 supports five different options, which we will present in more details in Section III-C.

The strict separation and abstraction of the different components allow a flexible configuration of a variety of different targets. Thus, in comparison to the first version of Avatar, the scope of the framework is extended far beyond the initial target of dynamically analyzing embedded devices firmware. This is due to the drastic shift of paradigm in avatar^2: instead of orchestrating specific tools with a specific goal, the core goal of avatar^2 is to enable a general interoperability among an arbitrary number of different tools frequently used for dynamic binary analysis.

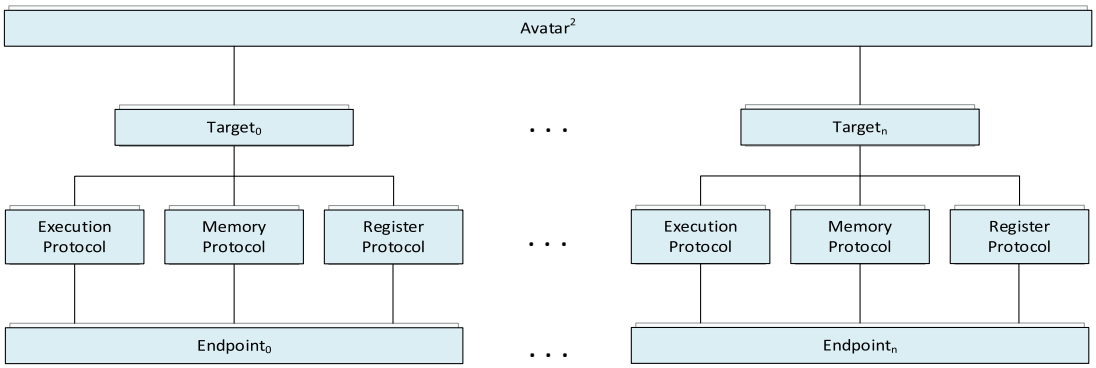

Fig. 1: Overview of avatar²

B. Under the Hood

So far we introduced the general design of the framework. We now highlight and discuss in more details five specific features provided by avatar². These features are intended to provide additional flexibility in order to cope with different dynamic binary analysis tools.

1) Architecture Independence: With the emerging interconnectivity of software not only on commodity computers, but also on embedded systems, the variety of architectures and instruction sets of interest for program analysis systems is broader than ever. Intuitively, as several dynamic binary analysis frameworks already come with support for multiple architectures, an orchestration framework should also be able to cope with those. Avatar² handles this problem by relying on a flexible description of the architectures in a modular manner, with the additional possibility to provide annotations for specific targets (i.e., special variables defined in the architecture that are fetched and consumed by targets). While the current implementation is shipped with descriptions for *x86*, *x86_64* and *ARM*, the modular approach allows to easily extend the framework to support additional architectures or even intermediate representations used by specific tools.

2) Internal Memory Representation: Avatar² is designed to interconnect a variety of targets, which internally rarely use the same representation of memory. However, to synchronize the analysis across different frameworks and platforms, a consistent view of a program's memory is required. Hence, avatar² provides interfaces to the analyst for defining and updating the memory layout, which is then pushed to the targets. For instance, unlike other tools, avatar² does not represent memory on page granularity. This is because its goal to be able to cope with embedded devices which often consist of memory mapped peripherals and CPU registers that use only a fraction of a common page. Instead, avatar² works by combining *memory ranges* of arbitrary and non-uniform sizes.

3) Legacy Python Support: The initial prototype of avatar²'s core was written in Python 3.x. While we believe that a future migration to Python 3 is inevitable, several popular dynamic analysis frameworks, such as angr, manticore, and triton are either based completely on – or export bindings only to – python 2.7. Therefore, we decided to work with Python 3 but still maintaining legacy python support to enable a flawless and performant integration to the aforementioned tools.

4) Peripheral Modeling: Embedded devices often consist of custom peripherals which are not implemented inside other endpoints or – even worse – that cannot be represented equivalently in other endpoints at all. Like the first Avatar, avatar² is able to solve this issue by memory forwarding. However, as memory forwarding can quickly become a performance bottleneck, avatar² provides an additional way to face this issue by adding prototypes of simple peripherals models. These models can be easily developed in python by the analyst and are, in essence, simple objects that respond to memory reads and writes at specified offsets. Facilitating these modeling mechanism, avatar² provides for instance an implementation for a universal serial port interface (USART) which models an interface present in a particular ARM based microcontroller by STMicroelectronics. Hereby, the model receives and transmits input and output over a tcp connection, instead of a physical peripheral.

5) Plugin System: Avatar² is designed to have a minimalistic and easy-to-maintain core, whereas the more complex logic is provided by the specific targets and protocols. However, a variety of tasks required by most dynamic analysis procedures are repetitive and therefore would be very inefficient if the analysts would need to re-implement them in every experiment. Therefore, to provide a common code base for these repetitive tasks and to execute them automatically, avatar² adopts a rich event-driven plugin-system. Within a plugin, the various events processed during an analysis can be hooked by custom callbacks, or completely new features can be added to the avatar² core. Examples for already existing plugins are an assembler and a disassembler, a forwarding plugin for single instructions, which is for instance useful to dispatch co-processor accesses, or a plugin for an *automated* orchestration of the analysis.

C. Supported Targets

Avatar² is designed to integrate new targets with low effort. Currently, it supports five targets, which already provide a large number of analyses combinations.

The Gnu Debugger (GDB).
The ability to communicate with GDB is probably one of the most essential features of avatar^2. The stand-alone target allows to debug GNU/Linux software. Moreover, a variety of endpoints are offering a gdbserver for debugging purposes. Due to the separation of targets and protocols, avatar^2 is able to communicate to all of these endpoints.

OpenOCD.
Modern CPUs and MCUs, and in particular those used in embedded devices, expose standardized debugging ports, such as Joint Action Test Group (JTAG) or Serial Wire Debug (SWD) ports. OpenOCD is an open source tool able to control debug dongles which can be attached to these ports. Those dongles, together with OpenOCD, can be used for fine-grained debugging of the executed software. Naturally, avatar^2 supports OpenOCD to perform analysis of embedded devices.

Quick Emulator (QEMU).
QEMU is a popular emulator, which, at its core, uses dynamic binary translation to allow emulation of software written for different architectures. Although it allows to emulate single GNU/Linux programs in its user mode via dynamic system call translation, avatar^2 uses its full system emulation mode to handle hardware peripherals, too. We add two noteworthy components to QEMU which we maintain as part of the avatar^2 project. First, we introduced a new emulation machine, the *configurable machine*, which is able to configure the memory layout in a flexible way. This facilitates the integration with the memory representation of avatar^2 but also to support variety of embedded devices. Second, we added a set of dedicated avatar peripherals, which are responsible for the interaction with other targets during an analysis.

PANDA.
The Platform for Architecture-Neutral Dynamic Analysis (PANDA) [12] is a dynamic analysis engine with focus on enabling repeatable reverse engineering. PANDA is based on QEMU, and hence, reuses the same components incorporated in the avatar^2's QEMU target. The strength of PANDA lies on its additional capabilities to record, replay, and analyze a previously-recorded concrete execution.

angr.
The symbolic execution and program analysis framework angr [24] provides a number of powerful dynamic analysis capabilities. Several small additions to angr have been performed for a better integration with avatar^2. Those modifications are mainly on angr's state and memory management. More precisely, a special representation for memory pages to provide access to the memory of other targets has been added, together with several automatic procedures to set up an analysis state that can be used for avatar^2. Furthermore, as the angr target in avatar^2 inherits angr objects, no direct modifications of angr are required.

Listing 1: Re-implementation of HARVEY, using avatar^2.

```
from avatar2 import Avatar, ARMV7M, OpenOCDTarget

input_hook = '''mov r5,0xffffffc
                b 0x20001E30'''

output_hook = '''mov r5,0xfffffffd
                 mov r4, r5
                 mov r5, 0
                 b 0x2000233E'''

avatar = Avatar(arch=ARMV7M)
avatar.load_plugin('assembler')

t = avatar.add_target(OpenOCDTarget,
        openocd_script='harvey.cfg',
        gdb_executable='arm-none-eabi-gdb')

t.init()

t.set_breakpoint(0xd270)
t.cont()
t.wait()

t.inject_asm('b 0x2000250E',addr=0x20001E2E)
t.inject_asm('b 0x20002514',addr=0x20002338)

t.inject_asm(input_hook, addr=0x2000250E)
t.inject_asm(output_hook,addr=0x20002514)

t.cont()
```

IV. Use Cases

In this section, we present three case studies to demonstrate the versatility of avatar^2 and the usefulness of dynamic multi-target orchestration. Like the rest of avatar^2, the full source code of those examples is publicly available at https://github.com/avatartwo/bar18_avatar2. Those case studies provide an in-depth view of several possible uses of avatar^2 but is not exhaustive. In fact, we designed avatar^2 to be as versatile as possible, allowing a variety of applications and many more configurations are possible than those presented here.

A. Facilitating replication and reproduction of previous studies

Recent initiatives are pushing more and more towards developing processes for facilitating replication[3] of scientific studies in the system security field [6]. However, when no example source code is publicly available reproduction of the results is required. Additionally, when embedded devices are involved, reproduction of previous work is often complicated.

We choose HARVEY [14], a recently presented PLC rootkit as a reproduction example. Using this example we show that avatar^2 can be used as a lightweight mechanism to prototype scripts for reproduction of previous studies (or share scripts to facilitate reliable replication). This is a good example for a reproduction study, as this publication contains all the necessary details to reproduce it while no source code or scripts are publicly available.

[3]Following the terminology from https://www.acm.org/publications/policies/artifact-review-badging

Here, we only use avatar^2 for scripting a single target (rather than orchestrating multiple targets). As such, this example does not perform any dynamic binary analysis.

The HARVEY rootkit is designed to be inserted in the firmware of an Allen Bradley PLC. It modifies the functions responsible for forwarding updates of physical inputs and outputs to other parts of the hardware, such as the LEDs and the Human Machine Interface (HMI). As a result, the rootkit is able to tamper with the PLC's I/O in a stealthy manner without reporting suspicious behavior on the LEDs or HMI. For deploying the required modifications, the authors of HARVEY describe two ways of deploying the compromised firmware on the PLC: a) by using JTAG debug access to patch the firmware in memory b) by abusing the firmware update functionality to persistently upload the malicious firmware. Option (a) is possible in this scenario, as parts of the firmware are loaded into and executed from the on-chip SRAM of the main MCU, which - by design - can be modified during runtime. Option (b), on the other hand, requires to exploit the firmware update process. However, the firmware for this PLC is cryptographically signed and the installation of a modified firmware would either require the ability to create a new firmware with colliding SHA-1 hash, the knowledge of the PLC's manufacturers private key, or the presence of a flaw in this signature verification mechanism.

Although we use a slightly different version of the PLC than in the original[4], we were able to reproduce the base proof of concept implementation of HARVEY using avatar^2 and with about 30 lines of python (Listing 1). Figure 2 shows the PLC after infection by HARVEY: the two orange LEDs indicate the presence of input signals on Port 1 and 2, albeit no inputs are connected to the PLC.

This example provides several interesting insights into the avatar^2 framework. Line 12 loads the assembler plugin, which adds the capability to assemble and inject code into memory, as done in Line 24-28. Then a target is configured (L. 14-17), and it is initialized as a standalone target (L. 18). However, the hooks for HARVEY can not be inserted right away, as the secondary firmware code has first to be loaded into the SRAM. Hence, we insert a breakpoint after those initialization functions (L. 20), before starting the execution (L.21) and waiting for the breakpoint to be hit (L. 22). The hooks are then inserted (L. 24 to L.28) and the execution is resumed (L. 30).

While the full rootkit can be injected into the firmware without the presence of avatar^2, we want to stress that the usage of the framework provides a *centralized* and *unified* interface for dynamic instrumentation, which greatly eases replicability and, in turn, reproducability.

B. Extending Symbolic Execution to complex Software

Although the state-of-the-art in symbolic execution of binary software is steadily progressing, its application is often bound to rather simple software for a variety of reasons. The limiting factors can either be implementation constraints, such as missing environment abstractions, or more general problems to symbolic execution in general, such as state explosion or limitations of SMT-Solvers.

[4]We used an Allen Bradley 1769-L16ER-BB1B CompactLogix 5370 PLC, while HARVEY was initially implemented on an Allen Bradley 1769-L18ER-BB1B CompactLogix 5370

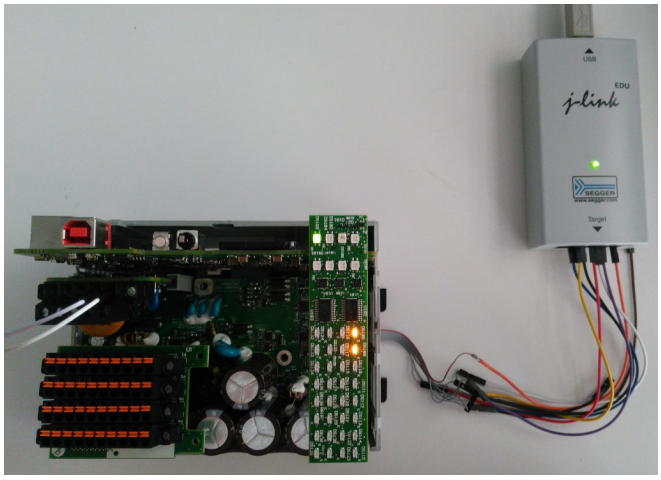

Fig. 2: PLC Infected with HARVEY using avatar^2 as instrumentation framework. The communication with avatar^2 is performed with a JTAG debugger.

Although we do not aim to provide a generic solution for such limitations, we show that avatar^2's state transfer and synchronization capabilities can be used to dynamically transfer states from concretely executed software into symbolic execution engines which would not be otherwise able to reach such state of execution. More specifically, we will analyze *Firefox*, a popular browser. We choose this particular program because of its size and the complexity, which make it nearly impossible to analyze using a symbolic engine such as *angr*.

To show the implications for binary analysis, we will use avatar^2 to discover an artificially inserted bug in the GL-rendering engine of the browser's javascript engine. While the inserted bug itself is a trivial null pointer dereference, its location and trigger condition are inspired by CVE-2017-5459[5].

The base idea how to leverage the capabilities of avatar^2 is extremely simple: we execute Firefox in GDB until the function of interest, and then transfer the concrete state into angr, symbolize some attacker controlled arguments to the function and explore until we eventually find a bug. This approach has a number of advantages with respect to a plain symbolic execution. First, it enables the execution of the initialization phase of the program under analysis at nearly[6] native speed. Second, the symbolic execution starts from a valid concrete state of the program, which reduces the need for over-approximations and, subsequently, reduces the exploration space. This is in contrast to letting angr analyze one single function of a binary by considering symbolic all the environment around the execution, including the memory, which would also lead the analysis through paths which are not possible in a concrete execution of the program.

[5]https://bugzilla.mozilla.org/show_bug.cgi?id=1333858

[6]Running the process in GDB introduces some slowdowns.

While the idea sounds simple, avatar^2 has to overcome several challenges. First of all, *angr* stores a variety of metadata associated to each memory location. As a result a full state transfer would blast the memory representation capabilities of angr. Hence, on state transfer we only transfer the *memory mapping* into angr, while leaving the memory uninitialized. The transfer of memory contents happens when angr actually accesses the memory, effectively resulting into a *copy-on-read* primitive. In contrast, writes are local to angr and they are not forwarded back to the Firefox process, as this would introduce inconsistencies between the different symbolic states inside angr.

Another problem during state transfer are the segment registers, which are not exported by GDB and which can not easily be transferred into general purpose registers. We created a shared library injected into the process under analysis, which uses the `arch_prctl` syscall to retrieve the values for those registers. Upon a state transfer, avatar^2 snapshots the register state, executes the library functions to retrieve the segment registers, forwards them to angr and restores the original state of the process under analysis.

A third challenge we had to overcome are commonly used library functions. Indeed, many symbolic execution environments such as KLEE [3] or angr do not symbolically execute standard libraries functions (e.g., printf, fread) because of their complexity and unwanted interaction with the environment. Instead, simpler function stubs are executed, which are called "SimProcedures" in the angr terminology. To benefit from the advantages of such abstractions, we automatically match the SimProcedures known to angr with the symbols present in the analyzed process. Once a match is found, the according address in the process is hooked with the SimProcedure, effectively retrofitting the function abstractions onto the state transferred to angr.

The interested reader can find our avatar^2 script for finding the bug in the appendix. This script runs in approximately 9 minutes in our test environment consisting of a VM with four Intel Xeon E5-2650L cores and 16GB of RAM. During the symbolic exploration 36 basic blocks are executed and 21 unique pages are copied from GDB to angr. The observable effects of this itself are rather trivial to detect in this example, as it causes an invalid memory access, which both angr and avatar^2 can easily recognize.

However, the path causing this invalid memory access can not easily be found utilizing angr alone. In fact, we were not able to detect this bug when advising angr to begin the exploration from the beginning of the function containing the bug without inheriting a concrete state from avatar^2. The reason for this lies within the fact that the GL rendering functionalities are involving frequent accesses to global objects, which is hindering the exploration when annotated fully symbolically. The other approach, of executing Firefox from the beginning in angr, is likewise not able to uncover this bug. This is because of two reasons: First, this bug is located deeply within the execution and too many paths would have to be explored first before realistically finding the bug. Second, a variety of mechanisms used internally in Firefox can currently not be expressed in angr, such as inter-thread communications.

Listing 2: Recording embedded device's execution.

```
from avatar2 import  ARMV7M, Avatar, OpenOCDTarget,
    PandaTarget

avatar = Avatar(arch=ARMV7M)
avatar.load_plugin('orchestrator')

nucleo = avatar.add_target(OpenOCDTarget,
          openocd_script='nucleo-l152re.cfg',
          gdb_executable='arm-none-eabi-gdb')

panda = avatar.add_target(PandaTarget,
          executable='panda/qemu-system-arm',
          gdb_executable='arm-none-eabi-gdb')

rom = avatar.add_memory_range(0x08000000, 0x1000000,
        file=firmware)
ram = avatar.add_memory_range(0x20000000, 0x14000)
mmio= avatar.add_memory_range(0x40000000, 0x1000000,
        forwarded=True, forwarded_to=nucleo)

avatar.init_targets()

avatar.start_target = nucleo
avatar.add_transition(0x8005104, nucleo, panda,
         synced_ranges=[ram], stop=True)
avatar.start_orchestration()

panda.begin_record('panda_record')
avatar.resume_orchestration(blocking=False)

[...]

avatar.stop_orchestration()
panda.end_record()
```

As a result, angr can outperform the other approaches by utilizing an initial concrete state, which can either be manually injected by the analyst, or, as seen in the example, automatically extracted from gdb using avatar^2. This perfectly demonstrates the advantages of state transfer and synchronization primitives provided by the framework.

C. Recording and Exchange of Firmware Execution

Traditionally, dynamic analysis of firmware for embedded devices requires either the presence of the physical device or the ability to emulate the firmware. In this example, we show that avatar^2 is capable of recording parts of the execution of a firmware by partially emulating it. This recording can then be replayed and analyzed without the presence of the physical device. To achieve this, we instruct avatar^2 to use the `PandaTarget` to emulate and record the core parts of the firmware, while memory accesses to hardware peripherals are forwarded to a physical device, controlled by avatar^2's `OpenOCDTarget`.

In this example, we use a Nucleo STM32L152RE development board as physical target, which comes with an ARM Cortex-M3 MCU whose JTAG connection is available over the on-board USB interface. We use an example firmware form *ARM mbed*[7]. As often with embedded code, it performs many low-level memory accesses to the hardware's peripherals. PANDA alone would not be able to emulate the execution of

[7] https://www.mbed.com

this firmware, because the firmware would not execute properly without proper emulation of the device's specific peripherals.

The according avatar^2 script is shown in Listing 2 and exhibits a couple of notable points. Line 3-12 are responsible for the usual general setup of avatar^2 and targets. Note that this time the orchestration plugin is loaded, which allows a different way to control the execution of targets as seen in the previous example. Line 14-18 define an *explicit* memory layout, including the ROM memory, backed by the firmware, and Memory-Mapped I/O (MMIO), which will be forwarded to the physical device, using the *remote memory* functionalities of avatar^2.

After the definition of memory, the targets are initialized (L. 20) and the actual orchestration is set up: Line 22 defines which target shall be used for execution first. Line 23 adds *transition*, in which avatar^2 will switch the execution from one target to another, while synchronizing the registers and memory ranges specified by the synced_ranges argument. In this case, only the RAM range needs to be synchronized, as this is the only dynamic memory local to more than one target within this analysis. The last argument, stop, instructs avatar^2 to stop the orchestration after this transition. The reason for this state transfer lies in the desire to execute the initialization functionalities of the physical device on the device itself, as they are not of interest for the analysis of the main firmware.

The following line (L. 24) starts the automatic orchestration. Avatar^2 will run until a transition with the stop flag is hit, which is trivial in this case, as only one transition is defined. As a result, line 27, which enables the execution recording of PANDA, is executed after the transition finished and the automatic orchestration has to be resumed (L. 28). Once the interesting parts of the firmware's execution have been recorded, both the orchestration and the recording can be stopped (L. 32f). Afterwards, the execution recording is available and can be reused in PANDA (e.g., to perform further analysis) without using the embedded device.

This demonstrates the importance of avatar^2's unique separation between execution and memory, which allows the forwarding of MMIO in the first place. Without this, the initial recording, even with the presence of the physical device, would not be possible as long the underlying hardware platform can not fully be emulated.

V. Discussion

In order to effectively combine and orchestrate different frameworks, avatar^2 needs to be generic enough to support a variety of frameworks with different design philosophies, execution primitives and scopes. Even though we think that it is unfeasible to be generic enough to allow support for every dynamic binary analysis frameworks, we believe that our abstraction and distinction of targets, protocols and endpoints enable a variety of frameworks to be potential targets for avatar^2.

In fact, we designed avatar^2 to keep the implementation overhead for adding a new target as simple as possible. To add a new target, an analyst needs first to decide over which protocol instances it communicates to the associated endpoints. In case the endpoint can not be controlled over already existing protocols, the implementation of the additional protocols will require the majority of the effort. However, we believe that by providing protocol implementations for both GDB and QEMU-based targets, already a decent amount of potential endpoints can be integrated into avatar^2 without the need to add new protocols. Once the right protocols are chosen, the actual target class can be written, which needs to provide interfaces for functionalities specific to the target, and an initialization function, which sets up the endpoint and connects the different protocols to it.

One of the main goals of the framework is to enable popular dynamic binary analysis frameworks to interoperate with embedded devices firmware. While two out of the three examples were targeting embedded devices, both were relying on the presence of a JTAG interface. Unfortunately, when analyzing real world hardware, such an interface is not always available. However, even in those cases avatar^2 can be used together with these hardware instances if, for example, a gdbserver can be launched directly on the device or a GDB stub can be injected at runtime, for instance using a bootloader. However, those stubs are highly dependent on the architecture of the analysed target and are hard to abstract in a generic way. Although such stubs already exist for some targets [8] However, the stubs are not currently part of the current version of avatar^2, but we plan to provide such debuggers as dynamic loadable plugins in the future.

An additional challenge for embedded devices is given when embedded devices do not communicate over MMIO, but trigger interrupts, e.g., upon arrival of new data. avatar^2 provides an experimental support for forwarding interrupts on some hardware. However, the lack of genericity of interrupt handling is a limitation of the framework. Indeed, the way interrupts are triggered and served is tightly coupled to the hardware they are occurring on. As a result, interrupts need to be implemented in a per-target manner by using dedicated protocols for dispatching interrupts.

VI. Related Work

Even though avatar^2 is - up to our knowledge - the first attempt to flexibly combine debuggers, emulators and dynamic binary analysis frameworks in a generic manner, a few tools have been directed to solve specific subproblems also tackled in avatar^2.

First and foremost, several existing tools embed or integrate other, third-party tools. So does Driller [25] for instance combine angr [24] with AFL [30] to benefit from both the advantages of symbolic execution and fuzzing. Similarly, FrankenPSE [28], allows sharing of snapshots between PySymEmu [13], a symbolic execution tool and the GRR Fuzzer [26]. Unfortunately, as of time of writing, no open source version of FrankenPSE has been made available, which makes a direct comparison to other approaches difficult.

Independently, angr recently deployed a modular design-philosophy, allowing to exchange different parts within the symbolic execution framework. As a result, different execution

[8] https://github.com/avatarone/avatar-gdbstub or qcdebug [9]

engines, Intermediate Representation (IR) or constraint solvers can be plugged into the framework.

Another example for a tool benefiting from external projects is given by *radare2* [19]. Although being a reverse engineering framework as its core it facilitates code emulation thanks to a custom IR. Nevertheless, it is designed to be enabled to control a variety of debuggers, and provide implementations for GDB and WinDBG. On top of this, community based plugins are integrating frameworks like Miasm [10] or emulators like Unicorn [20] into the radare2-ecosystem. Although this, just like angr, already enables powerful analysis, the specific tool is, together with its initial purpose, in the foreground.

Next to interconnecting independent targets, approaches for decoupling execution and memory have been incorperated by other tools, for instance by Surrogates [17] and Prospect [16], which both use memory forwarding to enable the analysis of embedded devices. Hereby, Prospect focuses on allowing partial emulation of Linux based targets by forwarding syscalls accessing device drivers. Surrogates, on the other hand, uses custom hardware to forward MMIO accesses with a high bandwidth, effectively enabling near-realtime analysis of embedded devices.

Additionally, a lot of modern dynamic binary analysis frameworks with different purposes are based on QEMU. DECAF [15] for instance focuses on just-in-time virtual machine introspection and tainting, while PANDA [12] provides primitives for recording and replaying executions, whereas advanced analysis plugins are used during the repeatable replay of an execution. Rev.ng [11], on the other hand, is most notably known for recovering control flow graphs and function boundaries as basic block for subsequent analyses and S^2E [5] expands full-system emulation with the capability of symbolic and concolic execution. Furthermore, even tools for analyzing embedded devices firmware without having the actual device are based on QEMU, such as Firmadyne [4], which emulates a generic kernel for Linux-based firmware, and LuaQEMU [7] which provides prototyping of hardware boards in Lua.

While all of those tools have their strength and are already quite powerful, they are rarely designed with interoperability in mind. As a result, the majority of those tools heavily modify QEMU for their purposes, effectively denying an easy integration with other tools. The patches for QEMU done by `avatar`^{2}, on the other hand, are minimalistic and centralized in the code base, which leads to an easy integrability of those tools as future targets for `avatar`^{2}.

VII. Conclusion

In this paper, we showed the need to be able to interconnect popular dynamic binary analysis frameworks. We presented `avatar`^{2}, a python based tool, designed to be able to dynamically orchestrate a wide range of frameworks, debuggers, and emulators. The initial implementation of the framework is already able to orchestrate the execution between GDB, QEMU, angr, OpenOCD, and PANDA, and provides easy-to-use interfaces to transfer the state across one of the tools into another. We demonstrated the versatility of `avatar`^{2} with three use cases, which by no means exhaust the possibilities provided by the framework. More specifically, we used `avatar`^{2} as dynamic instrumentation tool to reproduce HARVEY, a novel PLC rootkit, we found an artificially inserted bug in Firefox by combining GDB with angr, and we recorded the execution of an embedded device firmware, by using PANDA together with OpenOCD.

All in all, we believe that `avatar`^{2} can greatly improve state of the art dynamic binary analysis, as now the strengths of different tools can be combined into single analysis scenarios.

References

[1] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE, 2013.

[2] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *International Conference on Computer Aided Verification*. Springer, 2011.

[3] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation*, 2008.

[4] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware." in *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[5] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," in *16th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2011.

[6] C. Collberg and T. A. Proebsting, "Repeatability in computer systems research," *Communications of the ACM*, vol. 59, no. 3, 2016.

[7] Comsecuris, "Luaqemu," https://github.com/comsecuris/luaqemu.

[8] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M.-L. Potet, and J.-Y. Marion, "BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis," in *23rd International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2016.

[9] G. Delugr, "Reverse-engineering a qualcomm baseband," 28th Chaos Communication Congress, 2011, https://code.google.com/archive/p/qcombbdbg/.

[10] F. Desclaux, "Miasm : Framework de reverse engineering," in *SSTIC, Symposium sur la scurit des technologies de l'information et des communications*, 2012, https://github.com/cea-sec/miasm/.

[11] A. Di Federico, M. Payer, and G. Agosta, "rev. ng: a unified binary analysis framework to recover cfgs and function boundaries," in *Proceedings of the 26th International Conference on Compiler Construction*. ACM, 2017.

[12] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering with PANDA," in *5th Program Protection and Reverse Engineering Workshop*. ACM, 2015.

[13] feliam, "PySymEmu: An intel 64 symbolic emulator," https://github.com/feliam/pysymemu.

[14] L. Garcia, F. Brasser, M. H. Cintuglu, A.-R. Sadeghi, O. Mohammed, and S. A. Zonouz, "Hey, my malware knows physics attacking PLCs with physical model aware rootkit," in *Proceedings of the Network and Distributed System Security Symposium*, 2017.

[15] A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin, "Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014.

[16] M. Kammerstetter, C. Platzer, and W. Kastner, "Prospect: peripheral proxying supported embedded code testing," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM, 2014.

[17] K. Koscher, T. Kohno, and D. Molnar, "Surrogates: Enabling near-real-time dynamic analyses of embedded systems." in *USENIX Workshop on Offensive Technologies*, 2015.

[18] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Conference on Programming Language Design and Implementation*. ACM, 2007.

[19] pancake, "radare2: unix-like reverse engineering framework and commandline tools," http://radare.org.

[20] N. A. Quynh and D. H. Vu, "Unicorn-the ultimate CPU emulator."

[21] F. Saudel and J. Salwan, "Triton: A dynamic symbolic execution framework," in *SSTIC, Symposium sur la sécurité des technologies de linformation et des communications*, 2015.

[22] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker." in *USENIX Annual Technical Conference*, 2012.

[23] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy*. IEEE, 2016.

[24] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[25] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[26] Trail of Bits, "GRR: High-throughput fuzzer and emulator of DECREE binaries," https://github.com/trailofbits/manticore.

[27] Trail of Bits, "Manticore: Symbolic execution for humans," https://github.com/trailofbits/manticore.

[28] Trail of Bits, "A fuzzer and a symbolic executor walk into a cloud," 2016, https://blog.trailofbits.com/2016/08/02/engineering-solutions-to-hard-program-analysis-problems/.

[29] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "Avatar: A framework to support dynamic security analysis of embedded systems' firmwares." in *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[30] M. Zalewski, "American fuzzy lop," 2014, http://lcamtuf.coredump.cx/afl/.

Appendix A: Avatar^2-script for finding the inserted bug in firefox.

```
import avatar2
import angr as a
import logging

firefox_binary = "./firefox"
trigger = "./trigger"

breakpoint_function = "mozilla::WebGLContext::ReadPixelsImpl"

print "[+] Creating the Avatar object"
ava = avatar2.Avatar(arch=avatar2.archs.X86_64)
ava.log.setLevel(logging.INFO)
ava.load_plugin('gdb_memory_map_loader')
ava.load_plugin("x86.segment_registers")

print "[+] Creating the GDBTarget"
gdb = ava.add_target(avatar2.GDBTarget, local_binary=firefox_binary,
                     arguments=trigger)

# setup angr target
print "[+] Creating the AngrTarget"
load_options = {}
angr = ava.add_target(avatar2.AngrTarget, binary=firefox_binary,
                      load_options={'main_opts': {'backend':'elf'}})

print "[+] Initializing the targets"
ava.init_targets()

# Additional setup for GDB
gdb.disable_aslr()

print "[+] Running Firefox until %s" % breakpoint_function
gdb.bp(breakpoint_function, pending=True)

gdb.cont()
gdb.wait()

print "[+] Firefox reached %s" % breakpoint_function
print "[+] Avatar loads the memory ranges"
ava.load_memory_mappings(gdb, forward=True)

print "[+] Switching the execution to angr"
angr.hook_symbols(gdb)

options = a.options.common_options | set([a.options.STRICT_PAGE_ACCESS])
s = angr.angr.factory.avatar_state(angr, load_register_from=gdb,
                                   options=options)

s.regs.rsi = s.solver.BVS("x", 64)
s.regs.rdx = s.solver.BVS("y", 64)
simmgr = angr.angr.factory.simgr(s)

print "[+] Starting symbolic exploration."
while True:
    simmgr.step()
    if len(simmgr.errored) != 0:
        break

print "[+] Done, found error %s" % simmgr.errored[0].error
ava.shutdown()
```