SYMBION: Interleaving Symbolic with Concrete Execution

Fabio Gritti, Lorenzo Fontana, Eric Gustafson, Fabio Pagani, Andrea Continella,
Christopher Kruegel, and Giovanni Vigna

University of California, Santa Barbara
{degrigis, lfontana, edg, pagani, conand, chris, vigna}@cs.ucsb.edu

***Abstract*—Symbolic execution is a powerful technique for exploring programs and generating inputs that drive them into specific states. However, symbolic execution is also known to suffer from severe limitations, which prevent its application to real-world software. For example, symbolically executing programs requires modeling their interactions with the surrounding environment (e.g., libraries, operating systems). Unfortunately, models are usually created manually, introducing considerable approximations of the programs behaviors and significant imprecision in the analysis. In addition, as the complexity of the system under analysis grows, additional models are needed, making this process unsustainable. For these reasons, in this paper we propose a novel technique that allows interleaving symbolic execution with concrete execution, focusing the symbolic exploration only on interesting portions of code. We call this approach *interleaved symbolic execution*. The key idea of our approach is to re-use the concrete environment to run the input program, and then synchronize the results of the environment interactions with the symbolic execution engine. As a consequence, our approach does not make any assumption about such interactions, and it is agnostic with respect to the concrete environment. We implement a prototype for this technique, SYMBION, and we demonstrate its effectiveness by analyzing real-world malware, showing that it allows us to effectively skip complex portions of code that do not need to be analyzed symbolically.**

***Keywords*—Computer Security, Reverse engineering, Malware.**

I. INTRODUCTION

Symbolic execution has been shown to provide great benefits in analyzing programs and understanding their properties and behaviors, especially because of its ability to generate inputs that drive the execution toward interesting program states [1]–[4]. In particular, in the computer security community, this technique has been used to improve the code coverage of fuzzers [2], to reverse engineer malware [5], to discover backdoors in closed-source software [6], and to automatically generate exploits [2], [7]. Unfortunately, symbolic execution is haunted by the *state explosion* problem [1], which prevents this technique from being effectively applied to real-world software. In fact, if used without any precaution, symbolic execution tends to generate an exponentially increasing number of states that depend on symbolic data. For this reason, most of the uses of symbolic execution need custom-tailored heuristics to drive the exploration and to discard uninteresting states. Moreover, to further reduce the number of states, the interactions of the program under analysis with the surrounding environment are approximated by manually implemented routines. These routines, known as *models*, emulate the effects of a specific environment interaction (e.g., syscalls or calls to third-party libraries) on the program's (symbolic) state. This approximation represents another distinct problem for symbolic execution since models must be developed for every interaction the program has with the environment.

For these reasons, the task of symbolically executing a specific portion of an application's code can be quite challenging. In fact, symbolically executing the program from its entry point to the interesting part is often infeasible or prohibitively slow. On the other hand, a mere "jump into the middle" of the application (as proposed in *under-constrained symbolic execution* [8]) introduces the risk of encountering invalid program states caused by missing initialization code or unsatisfied data dependencies.

However, symbolic reasoning about specific portions of a program's code is a very useful, and needed task, in various fields. For example, consider a malicious program that, before unpacking the code responsible for the communication with a C&C server, executes some checks to fingerprint the execution environment [9]–[12]. In this case, even if the interesting code is the one that implements the communication with the C&C, there is no other choice than starting to execute the program inside the symbolic engine from its entry point, and waiting for the communication routine to be unpacked. However, this analysis is infeasible because the environmental checks performed by the malware sample might leverage un-modeled OS interactions, which are required to successfully reach the unpacking code.

Previous work proposed different techniques to either narrow symbolic execution to a particular portion of code [7], [13], [14], or avoid the execution of expensive portions of code that are unnecessary at a specific point of the analysis [15]. However, most of these approaches cannot perform a *runtime analysis*, i.e., they cannot modify data in memory to direct execution through specific branches at runtime [7], [16], [17]. For instance, after a symbolic analysis, one could be interested in flipping the condition of a branch to drive the concrete program execution toward another path. Similarly, tools that are tightly coupled to their dynamic analysis framework cannot be used with arbitrary or non-traditional execution environments (e.g., embedded devices). For instance, most of the existing tools leverage QEMU [18] for program emulation, but if the program cannot be emulated (e.g., due to hardware dependencies), the only remaining solution is to run the application on bare-metal. Unfortunately, this solution is often not compatible

with the design of the system used to symbolically analyze the target program.

To address the aforementioned problems, in this paper we introduce *interleaved symbolic execution*, a novel approach for symbolically executing complex binaries. Our technique provides granular control over the interleaving of concrete and symbolic executions of the analyzed program, and removes the need for implementing models for every environment interaction. Differently from the concolic testing approach [19], where the goal is to both symbolically and concretely execute a full trace of the target program to eventually improve code coverage, we aim at relieving the environment interaction modeling problem by re-using the concrete environment to symbolically execute only interesting portions of code. Nonetheless, the problem we discuss in this paper also affects concolic testing, and we, therefore, consider *interleaved symbolic execution* orthogonal to concolic testing.

We implemented our approach in a tool, SYMBION, that uses a modular, programmatically-controlled concrete environment where we execute the program under analysis until a specified point of interest. Afterward, SYMBION synchronizes the concrete state of the program with the symbolic execution engine, allowing for further exploration and precise data-flow reasoning. Finally, leveraging the information gained from the symbolic exploration, the program can then be steered toward a chosen branch by concretizing any symbolic data in the program's state and synchronizing its value back into the concrete environment. By iterating this execution strategy, we can delegate the execution of the complex portions of code to the concrete environment, and apply symbolic execution over arbitrary parts of the program. Moreover, our approach is agnostic with respect to the concrete environment, providing support for native targets running in any environment or architecture (e.g., debugger, emulators, and hardware devices).

We implement SYMBION on top of `angr` [4], and we evaluate it by analyzing real-world malware, showing how *interleaved symbolic execution* can effectively support analysts when reverse-engineering malicious code. Specifically, we show how SYMBION can be effectively used, without requiring the modeling of complex environment interactions, to detect the usage of a Domain Generation Algorithm (DGA), to reverse-engineer the C&C commands required to trigger malicious functionality, and to study the evasion techniques employed by malicious code.

In summary, we make the following contributions:

- We introduce *interleaved symbolic execution*, a novel approach that fuses concrete and symbolic execution by synchronizing the concrete state of a program under analysis with a symbolic execution engine, allowing the concrete execution to be steered by the results of the symbolic exploration.
- We implement our approach in SYMBION, a system that we release as an open-source project under `angr`'s master[1].
- We show the effectiveness of SYMBION by presenting analyses of real-world malware samples.

[1] https://github.com/angr/angr

II. MOTIVATION

When analyzing real-world programs, focusing on a smaller portion of code makes the analysis more practical. However, two main issues affect symbolic execution engines: *Missing Environment Models* and *Missing Data Dependencies*. The combination of these two problems often blocks the possibility to study a specific functionality of the program: it is not possible to initialize a symbolic execution engine at a specific point because of missing data dependencies, nor it is possible to reach that code from the entry point because of missing environment models.

A. Missing Environment Models

This issue arises when the models required to simplify a target program's execution are either missing or partially implemented. Depending on the design choices made to develop a specific symbolic execution engine, this can happen at three different levels: the *API-level*, the *syscall-level*, and the *instruction-level*.

API-level. Some symbolic engine implementations provide models for the most used functions (i.e., strcmp, strcpy, etc.) included in the C standard library (libc). The calling sites of these functions inside the program are then substituted with calls into the corresponding model implementation. While this choice has the benefit of relieving the symbolic execution engine from executing code related to those functions (i.e., analyses are faster, and propagate symbolic data more accurately), it puts the burden on the developers, who have to implement a large number of models to support the analysis of complex programs. If a model for a specific function is missing, there are two possible solutions. The first one is to let the symbolic engine explore the real code of the function, while the second one is to skip the function body altogether and return a symbolic variable at the call site. Unfortunately, both solutions can introduce enough complexity to trigger a state explosion.

Syscall-level. Similarly to API-level models, the syscall-level models are designed to mimic the effects of system calls made to a particular OS kernel. The majority of symbolic execution engines provide models at this level because it avoids the need to place the entire kernel in the emulated environment. Since system calls have substantial side-effects on the system and the user-land programs, models at this level must be complete and precise or the symbolic execution will reach incorrect states.

Instruction-level. A common technique to support symbolic execution on different architectures is to *lift* a program's code (i.e., express assembly instructions with a higher-level representation). In this way, symbolic execution engines can reason on this lifted representation and use it to update the program's state accordingly. Unfortunately, modern CISC architectures, such as Intel x86, count more than a thousand valid instructions and most of them have side-effects. This still represents an important challenge for the symbolic execution of intermediate representations, because instructions can be unsupported, lifted incorrectly, or have an incorrect or partial implementation. In all these cases, invalid program states are generated and the analysis results are unreliable.

```
1  func:
2    mov rax, [0x0000555555774000];
3    add rax, 0x10;
4    mov [0x0000555555774008], rax;
5  _loop:
6    mov rcx, [0x0000555555774008];
7    dec rcx;
8    mov [0x0000555555774008], rcx;
9    jnz _loop;
```

Fig. 1: Data dependency example

B. Missing Data Dependencies

Starting the symbolic exploration at arbitrary functions allows to skip code not relevant for the analysis or containing missing environment models. This operation is allowed by many symbolic execution engines, which let their users initialize and start the execution at specific, arbitrary addresses in the program. However, to provide reliable results, this approach requires forging a *valid* program state (i.e., the values of the registers and the memory) and to correctly express its data dependencies. These dependencies are generated when the code that we are symbolically executing refers to data produced by the execution of previous code. This makes the option of manually providing the initial state an error-prone process. For example, as shown in Figure 1, the effect of missing data dependencies amplifies the problematic aspects of symbolic execution even more, and generally leads to a state explosion problem. In this example, if the data required for the computation is not present at address `0x0000555555774000` the symbolic execution engine will store symbolic data at `0x0000555555774008`. This value is later used to control the loop termination (Line 9), and, since it is an unconstrained symbolic variable, it can theoretically execute an unbounded number of times, leading to a state explosion.

C. Motivating Example

Consider the snippet of code in Figure 2 that implements malicious code. The goal is to symbolically explore the function called `heavily_packed_function`. We could start to symbolically execute the malware from `main`; however, let us assume that we stumbled into the missing API-level model problem, and that the engine does not have a symbolic implementation for the WinAPI at Lines 11, 12, and 14 (*InternetOpenA, InternetOpenUrlA*, and *InternetReadFile*, respectively). As a result, the `download_config` function returns a buffer containing symbolic data, which is then passed as a parameter to `parse_config` at Line 34. This function uses this buffer to initialize the variable `xor_key_len` (Line 20) that therefore will be symbolic as well. Unfortunately, since this variable controls the loop's termination, this causes an unlimited number of iterations, and a consequent early state explosion, that prevents the symbolic executor from reaching the target function `heavily_packed_function`.

As we previously discussed, a possible way to avoid these problems is to start the exploration from `heavily_packed_function` itself. However, this function calls another routine (`evasion_check` at Line 2) that depends on specific data (`config`) processed by a

```
1  void heavily_packed_function(char *config){
2    evasion_checks(config);
3    /* [ block of packed/encrypted code] */
4    }
5  char* download_config()
6  {
7    HINTERNET hOpen = NULL, hFile = NULL;
8    DWORD dwBytesRead = 0;
9    char config[2000];
10   /* [ omitted code ] */
11   hOpen = InternetOpenA();
12   hFile = InternetOpenUrlA(hOpen,
13   "www.evildomain.com/conf.cfg");
14   InternetReadFile(hFile, (LPVOID)config, ...);
15   /* [ omitted code ] */
16   return config;
17 }
18 void parse_config(char *malware_config){
19   int config_len = atoi(malware_config[0])
20   int xor_key_len = atoi(malware_config[1])
21   char xor_key[xor_key_len];
22
23   for(int i=2, j=0; i<xor_key_len; i++,j++){
24     xor_key[j]= malware_config[i]
25   }
26
27   for(int i=xor_key_len, i<config_len; i++){
28     malware_config[i]= malware_config[i] ^
29     xor_key[i%xor_key_len]
30   }
31 }
32 int main( int argc, char **argv){
33   char *malware_config = download_config();
34   parse_config(malware_config);
35   decrypt_function(&heavily_packed_function,
36   malware_config.decryption_key)
37   heavily_packed_function(malware_config)
38 }
```

Fig. 2: Malware pseudocode example

previous routine (`parse_config`). Moreover, the code in `heavily_packed_function` is partially packed, and in the normal execution flow of the program, is decrypted by the unpacking routine `decrypt_function`. Therefore, starting the exploration from `heavily_packed_function` would cause the analysis to fail because the code that we want to analyze is revealed only after executing `decrypt_function`.

Finally, another possible and commonly used solution to these problems is to implement the missing models to enable the symbolic execution to safely reach the interesting portion of code. However, this approach requires time and effort, and it is not always scalable if the number of required models is very large. Consider, for example, complex programs like browsers, media players, or even a kernel; the amount of work necessary to implement the missing models can be unmanageable.

III. Approach

Our motivating example demonstrates the challenges of applying simple symbolic execution approaches to real malware samples, highlighting the need for techniques that allow one to effectively explore arbitrary program functions. For this reason, in this paper, we propose *interleaved symbolic*

execution, a technique that aims to enable a precise symbolic exploration of arbitrary functions. The key idea behind this approach is to symbolically execute a specific target function in its correct context. In practice, we concretely run the program under analysis until the initialization of such context is complete, and, after that, we *synchronize* the concrete state with the symbolic execution engine, replicating the concrete environment. Finally, starting from our synchronized state, we bootstrap our symbolic execution analysis. Essentially, our approach allows for the *interleaving* of symbolic execution with concrete execution, focusing the benefits of the symbolic exploration on specific portions of code. More precisely, given a binary program, *interleaved symbolic execution* is carried out in three phases.

Phase 1: Initial Concrete Execution. We concretely execute the input binary from a Concrete Starting Point (`CSP`) to an arbitrary Point of Interest (`PoI`). Generally, at the beginning of the analysis, the `CSP` is the entry point of the binary. When the program's execution reaches the `PoI`, we stop it in that specific state (i.e., memory and registers).

Phase 2: Interleaved Symbolic Execution. Once the concrete execution reaches our `PoI`, we synchronize the concrete execution state with the symbolic engine. In practice, we instantiate a new symbolic state, we replicate the concrete state (i.e., by reconstructing the memory layout and the registers' values), and we set parts of the state (i.e., memory or registers) as symbolic. For instance, we might set as symbolic specific memory areas to uncover their data dependencies within the next blocks of code. We then explore the program symbolically until we reach a Target Point (`TP`).

Phase 3: Re-Synced Concrete Execution. When the symbolic execution reaches our target point, `TP`, we collect the constraints introduced during the exploration and evaluate them to obtain, for each symbolic variable, a concrete value that satisfies the collected constraints. Finally, we store the results of our symbolic exploration into the concrete process memory to drive the concrete execution from `PoI` to `TP`, thus achieving the exploration of our target point.

Note that we can iterate over our three phases by considering the `TP` as a new `CSP` (or a new `PoI`), therefore further exploring the input program.

Our approach enables the delegation of the execution of code affected by missing environment models to a concrete environment, and reconstructs an execution state that allows for the symbolic execution of the program from the `PoI` to the `TP`, without being affected by the missing data dependencies issue. Moreover, our system is agnostic with respect to the specific concrete environment and only requires a simple interface to control the program's execution. This allows for the flexible application of this approach to a broad range of scenarios, e.g., running firmware on native hardware and obtaining its execution state through JTAG debug ports.

Comparison with Existing Approaches. We compare SYMBION with the most similar related approaches. In our comparison, we take into account four different features that are needed to support symbolic execution-based analyses of complex programs (Table I) across different environments.

- *Run-time analysis*: whether the approach allows users to modify the target's memory at runtime, therefore steering the execution toward specific paths.
- *Multi-Targets Support*: whether the approach supports the execution of targets running inside different environments (e.g., embedded systems).
- *Programmatic Context Switching*: whether the tool provides a programmatic interface to change the execution context of the program from concrete to symbolic, and vice-versa.
- *Instrumentation Independency*: whether the approach is self-contained or strictly relies on a specific environment or on another component to analyze the target binary (e.g., QEMU, Pin). This makes the approach less general, as it is limited by the capabilities of the underlying instrumentation layer.

As we can see in Table I, the most similar approach is implemented by Avatar^2. The tool can be used to implement run-time analyses and its flexible interface can be used to easily support different execution environments, moreover, Avatar^2 does not need specific instrumentation of the target binary to perform analyses. However, a programmatic context switching is not officially supported by the tool. Triton implements a similar approach, but due to its dependency from an instrumentation framework (Pin [20]), it does not have fully multi-targets support, moreover, it lacks an interface to implement programmatic context switching. Finally, S^{2}Eand Mayhem, were not conceived to perform these kinds of analyses and they miss a proper interface to perform run-time analyses and programmatic context switching. Table I summarizes the comparison of such tools using the described features.

IV. SYSTEM DETAILS

We implemented SYMBION on top of the multi-platform binary analysis framework `angr` [4]. The software is written in Python, and its versatility and flexibility are well-suited to implement and prototype new research ideas.

A. Design Requirements

One of SYMBION's goals is to support switching between concrete and symbolic execution, avoiding expensive transfers

TABLE I: Features comparison between SYMBION and related works. (✓) means the tool has the specified feature. (✗) means the tool does not have the specified feature. (∼) means the tool partially implements the specified feature.

Tool	Run-time Analysis	Multi-Targets Support	Programmatic Context Switching	Instrumentation Independency
SYMBION	✓	✓	✓	✓
Avatar^2	✓	✓	∼	✓
Triton	✓	∼	∼	✗
S^{2}E	✗	✗	✗	✗
Mayhem	✗	✗	✗	✗

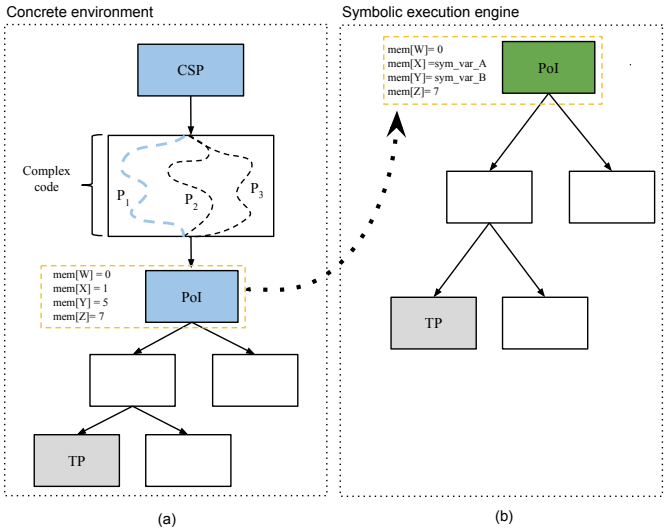

Fig. 3: An example of a Concrete→Symbolic transition.

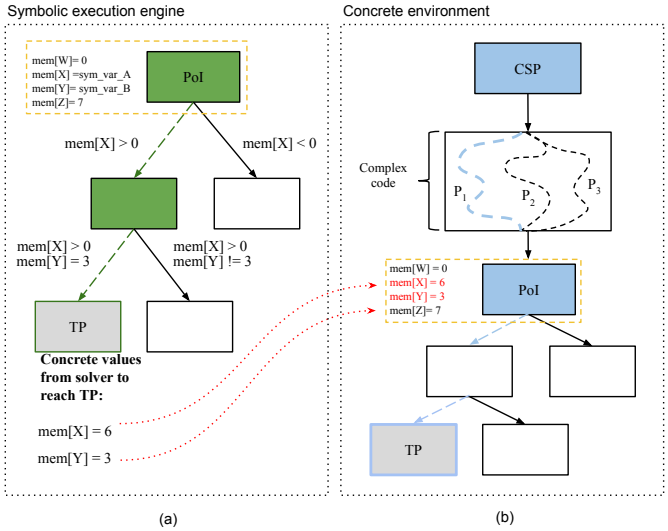

Fig. 4: An example of a Symbolic→Concrete transition.

of state. To implement the state synchronization from the concrete world to the symbolic world, we need a way to stop the program's execution at a `PoI`, copy the values of all the registers, and import them inside the symbolic engine. Moreover, we need to efficiently synchronize the memory of the concrete state inside the symbolic engine, and take care of all the details related to the underlying architecture (e.g., in the x86 architecture we need to synchronize the segment registers). We refer to this transition using the notation Concrete→Symbolic and we show an example of this transition in Figure 3. In Figure 3(a), we show three paths (P_1, P_2, P_3) that start from the `CSP`, end in the `PoI`, and pass through a block of complex code. The concrete execution of P_1 causes the memory of the program at the `PoI` to contain 0,1,5,7 at addresses W,X,Y,Z, respectively. Now that the `PoI` is reached, SYMBION automatically imports the concrete state into the symbolic engine. To begin our symbolic analysis, using SYMBION, we set the memory at addresses X and Y to be symbolic Figure 3(b) to eventually reason about paths in the program influenced by these memory locations.

Now the symbolic engine can explore different paths until the `TP`. Once the `TP` is reached, a Symbolic→Concrete transition is started. To implement this transition, SYMBION, under the user guidance, applies the memory modifications needed in the concrete process to reach the `TP`.

Figure 4 shows an example of this type of transition. In Figure 4(a) we have symbolically reached the `TP`, and by asking the constraint solver for possible values for `mem[X]` and `mem[Y]` we get respectively 6 and 3. In Figure 4(b), these values are synchronized into the concrete process, which is eventually resumed to reach the `TP`. At this point, `TP` can become a `CSP` state itself, effectively resuming the interleaved symbolic execution of the program.

Finally, we want to design a versatile interface to decouple our tool from the execution environment of the target program.

B. *angr*

Here we want to discuss the relevant concepts of `angr`, which plays a central role in SYMBION architecture.

SimState & SimEngine. `angr` executes programs by creating simulated states (*SimStates*). A SimState is logically a snapshot of a program in a specific state. A SimState memory can be manipulated to define some data as symbolic. To symbolically execute the program, `angr` receives as input a SimState, and, by using an Execution Engine (*SimEngine*), applies the effects of the program's instructions on that state to produce new SimStates (called successors).

SimPlugin. A SimPlugin is an object that holds additional data about a SimState. Moreover, it implements an interface to deal with the SimState lifecycle. The easiest example is the `state.globals` plugin that can be used to propagate information and data from one SimState to its successor.

Simulation Manager & Exploration Technique. A simulation manager (*SimManager*) is an interface that affects how successors are generated by applying search strategies (*Exploration Techniques*) to explore the program's execution paths.

SimProcedures. SimProcedures are Python routines that `angr` uses to synthesize the effect of syscalls and external libraries functions on an input SimState. These routines are imported during the initialization of the target program, and they are executed during the symbolic execution of the target.

C. *Symbion*

To develop SYMBION, we implemented a new SimEngine called *SimEngineConcrete*, a new Exploration Technique named SYMBION, and a new SimPlugin called *Concrete*. Moreover, we introduced a new concept, called *ConcreteTarget*, that exposes an interface that has to be implemented to support concrete execution in a new environment. An overview of SYMBION's architecture is shown in Figure 5.

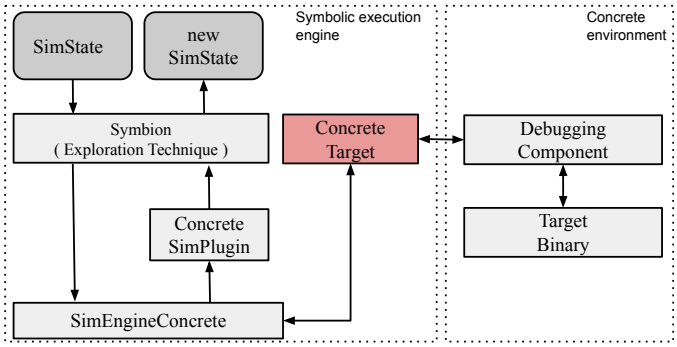

Fig. 5: Overview of SYMBION architecture.

SYMBION Exploration Technique. This new exploration technique exposes the basic APIs to interact with the system. Using these APIs, the user can define the `PoI`, and, therefore, specify where to stop the concrete execution in the concrete environment. Moreover, this interface lets users change memory and registers' values in the concrete environment, effectively controlling the execution of the target program from `angr`.

ConcreteTarget. This object exposes basic methods to interact with a new concrete environment. One of SYMBION's key features is that it can be easily extended to support new execution environments. To do so, developers only need to implement a ConcreteTarget that exposes the following methods:

- `read_memory(address, nbytes)`: returns `nbytes` bytes from the concrete process memory, starting from `address`.
- `write_memory(address, value)`: writes `value` in the concrete process memory at `address`.
- `read_register(register)`: returns the content of the specified `register` of the concrete process.
- `write_register(register, value)`: writes `value` in `register`.
- `set_breakpoint(address)`: sets a breakpoint at `address` in the concrete process.
- `remove_breakpoint(address)`: removes a breakpoint previously set at `address`.
- `run()`: resumes the concrete process execution.

A possible implementation of this interface is the `GDBConcreteTarget`, which is used to programmatically control the target program, running on a possibly remote machine, under a gdbsever.

SimEngineConcrete. The SimEngineConcrete leverages an implementation of the ConcreteTarget interface (e.g., the GDBConcreteTarget) to perform the following steps:

- Modify the concrete process memory with the values provided by the user at specific memory addresses.
- Modify the concrete process registers with the values provided by the user.
- Set the new `PoI` by translating them into breakpoints.
- Resume the execution of the program until a new `PoI` is reached, and give the control back to `angr`.

This execution flow effectively implements the transition Symbolic→Concrete.

Concrete SimPlugin. This plugin implements the transition Concrete→Symbolic. The execution of this plugin happens as soon as the target program, running in the concrete environment, hits a `PoI` set by the SimEngineConcrete. The plugin leverages the ConcreteTarget to import the concrete state of the program inside `angr`, thus creating a new SimState. The steps performed by this object are the following:

- Set the new SimState's memory backend to redirect all the future memory reads, performed by the program during the symbolic execution, to the memory of the concrete process.
- Copy the values of all the registers in the concrete state inside the newly generated SimState.
- Synchronize the memory mapping of the concrete process with the memory mapping loaded by `angr` during the startup (e.g., where the main module and libraries are loaded).

V. USE CASES

Here we show how SYMBION can be successfully used to analyze real-world malware. As we summarized in Table II, for our experiments we chose three different malware samples affecting two different operating systems (Windows and Linux). The main focus of these use cases is to show how *interleaved symbolic execution* can help perform a symbolic analysis of complex software.

Experiment setup. To run the samples we used two different virtual machines - one running Windows 7 and the other running Ubuntu 18.04 - both equipped with 4GB of RAM and 2 CPUs. For all the use cases, we leverage a gdbserver to run the target program inside the concrete environments. As a ConcreteTarget object, we re-used the *GDBTarget* made by Avatar^2 [21] with some minor modifications. We call this specific concrete target *AvatarGDBConcreteTarget*, and we eventually initialize the SimEngineConcrete using this object. The symbolic execution engine (angr) is running on an Ubuntu 18.04 host with 32GB of RAM and an Intel Core i7-4770 CPU.

The source code of these examples is publicly available at https://github.com/degrigis/symbion-use-cases.

A. Tracking C&C Domains Generation

For this example, we analyze a malicious PE32 binary that implements a password stealer distributed by spam emails.

TABLE II: Use Cases

Type	Family	Use case	OS	MD5
Trojan	Symmi	Detect DGA	Win	221c235bc70586ce4f4def9a147b8735
Bot	Bashlite	C&C commands	Linux	3d257d80963c9c905e883b568f997550
Ransom	Credle	Evasion	Win	53f6f9a0d0867c10841b815a1eea1468

```
<BV32 (if ((((0x0 .. __add__(0xfe624e21,
     SystemTimeAsFileTime_0_64[63:32], 0x0 .. (if
     (SystemTimeAsFileTime_0_64[31:0] <=
     (0x2ac18000 +
     SystemTimeAsFileTime_0_64[31:0])) then 0
     else 1)) ... [ omitted data ]
```

Fig. 6: Symbolic data in `gethostbyname` parameters.

Goal. During the first step of manual analysis, we saw that the sample makes requests to domains possibly generated by a Domain Generation Algorithm (DGA). This technique allows bots to periodically generate a new C&C domain to contact as a means to evade defenses based on domain blacklisting. Since the implementation of a DGA is quite often time-dependent, we want to confirm this hypothesis by trying to identify a connection between the WinAPI call `GetSystemTimeAsFileTime` at address `0x0040c493` and the call `gethostbyname` at address `0x00407f58`.

Challenges. The malware is not packed, but during a preliminary reverse-engineering we confirm that, during startup, a lot of complex and superfluous code (e.g., calls to random WinAPIs, allocation, and de-allocation of memory) are used to confuse an analyst, and to slow down the analysis performed by automated tools. Moreover, the sample includes a heavy initialization step during which four different threads compute the address of the main function of the malicious code. This complex initialization step would be enough to hinder any approach based on pure symbolic execution of the code, since symbolic execution would certainly incur in the state explosion problem.

Analysis. By leveraging SYMBION, we let the malware run until the initialization step is done, precisely, until the call to `GetSystemTimeAsFileTime`. After that, we perform a Concrete→Symbolic transition, and we mark the buffer returned from `GetSystemTimeAsFileTime` as symbolic. Now we let the symbolic engine explore the binary until it calls the function `gethostbyname`. Finally, when we reach `gethostbyname`, we inspect the data used as a parameter to check if symbolic data flows between the two calls.

To summarize:

- $\texttt{CSP}_1$ = Binary Entry Point (`0x40f827`),
- $\texttt{PoI}_1$ = call to `GetSystemTimeAsFileTime` (`0x0040c493`),
- $\texttt{TP}_1$ = call to `gethostbyname` (`0x00407f58`).

By implementing this analysis using SYMBION, we can see that the parameter for the `gethostbyname` call is indeed symbolic, and that it depends on the symbolic variable returned by the `GetSystemTimeAsFileTime` call (Figure 6).

B. Tracking Malicious Bot Commands

In this use-case, we show how SYMBION can be used to analyze an instance of a Bashlite backdoor (know as Qbot or LizardStresser). This Linux malware targets IoT devices, and it is used to build botnets exploited for DDoS attacks.

Goal. When analyzing a malware sample, a common problem is the ephemerality of the malicious infrastructure behind it. This affects most of the dynamic analysis techniques - for example when the sample is run in a sandbox - because the sample does not show its malicious behavior. In this use-case, we show how SYMBION can circumvent this problem, and how it can be used to identify which command triggers a specific functionality (e.g., the `ovhflood` action).

Challenges. When we run this sample, we can see that it attempts to connect to its C&C server at `185.244.25.213:3437`. However, this server is not working anymore, and thus, simply debugging the binary to understand which functionality is triggered by a specific command is not trivial. Moreover, this sample creates a new child process to handle every request received from the C&C server. This behavior represents a problem for symbolic execution engines because forking introduces considerable complexity (i.e., the engine must model the presence of another child process).

Analysis. To start our analysis, we manually identified the routine (called `echocommand`) that implements the dispatcher for the received commands at address `0x407ec4`. However, to reach this part of the code, a particular sequence from the C&C needs to be received. Given the different challenges involved, we break down this analysis in two different steps. First of all, we must understand and identify the sequence that we need to provide to the binary to reach the `echocommand` function. Then, when this sequence is uncovered, we can symbolically explore the program to understand which command triggers the specific functionality (`ovhflood`).

For this reason, we start the analysis by concretely executing the malware right after the connection attempt to the C&C; after that, we switch control to the symbolic execution engine, we modify the memory as if the connection has been successful, and we resume the concrete execution until the instruction before the call to `recvline` at `0x40abd3`. At this point, we perform another Concrete→Symbolic transition, and we mark the buffer returned by `recvline` as symbolic. To conclude this stage, we let the symbolic engine explore the binary until it calls the `echocommand` function.

The malware code execution is split in the following way:

- $\texttt{CSP}_1$ = Binary Entry Point (`0x40a488`).
- $\texttt{PoI}_1$ = Code right before the `recvline` (`0x40abd3`).
- $\texttt{TP}_1$ = First instruction of `echocommand` (`0x407ec4`).

Finally, by concretizing the symbolic variable returned by `recvline` (Figure 7), we can see the string the malware expects to receive to eventually start parsing malicious commands.

To begin the second step of our analysis, we simply concretize this sequence in the program memory, and resume the concrete execution until the beginning of `echocommand`. Once there, we perform another Concrete→Symbolic transition. Finally, we set the command buffer back to being a symbolic variable, and symbolically execute the program until we hit the basic block where the malware calls the `ovhflood`

```
print(next_state.solver.eval(cmd_buffer_symbolic,
                             cast_to=bytes))
b'!\xff \n\r\x00'
```

Fig. 7: Dumping the sequence of chars that will bring the execution to the echocommand functionality.

```
print(next_state.solver.eval(cmd_buffer_symbolic,
                             cast_to=bytes))
b'OVH\x00'
```

Fig. 8: Command we need to send to the backdoor to activate the ovhflood functionality.

function (address `0x409db2`). By solving the constraints for the new symbolic buffer we can extract the command to reach this specific functionality in the program.

For this step, we split the malware execution in this way:

- $\texttt{CSP}_2$ = $\texttt{TP}_1$ = First instruction of `echocommand` (`0x407ec4`).
- $\texttt{PoI}_2$ = $\texttt{CSP}_2$.
- $\texttt{TP}_2$ = call to `ovhflood` (`0x409db2`).

Note that, in this phase, we do not need any concrete execution from $\texttt{CSP}_2$ to $\texttt{PoI}_2$, rather we want to symbolically explore the code immediately starting from $\texttt{TP}_1$ (reached in the previous phase). To do this, we logically define that the new $\texttt{PoI}_2$ equals to the $\texttt{CSP}_2$. Unfortunately, since this function is quite complex, we run into the state explosion problem. We decided to help `angr` symbolically explore this piece of code by leveraging a technique known as *directed symbolic execution* [22], that basically prunes paths that do not connect $\texttt{CSP}_2$ (`echocommand`) and the $\texttt{TP}_2$ (call to `ovhflood`).

To conclude our analysis, once the $\texttt{TP}_2$ (call to `ovhflood`) is reached, we solve the constraints over the symbolic buffer containing the command, and we extract the string we need to use to reach this functionality, which, in this example, is the string "OVH" (Figure 8).

C. Bypassing Malware Evasion

In this example, we analyze a ransomware sample that uses a dynamic anti-analysis check to detect a virtualized environment.

Goal. First, we manually confirm that the ransomware behavior starts at address `0x4214e4`. However, this code is conditionally executed depending on the results returned by a call to the `GetProcessAffinityMask` WinAPI. The goal of this analysis is to understand which value this WinAPI has to return to eventually trigger the malicious code.

Challenges. The binary is packed, and therefore applying symbolic execution from the entry point of this sample would certainly run into the space explosion problem.

Analysis. We manually identify the point in the program where the code is fully unpacked and we spot the call to the Windows API `GetProcessAffinityMask` (address: `0x7502a889`). Here, we trigger a Concrete→Symbolic transition. To symbolically analyze this part of code, we need to write one simple SimProcedure to hook the `GetProcessAffinityMask` call[2]. In particular, this model just fills the buffers used to store the call's results with symbolic data. Using the constraints collected over those symbolic variables we will later understand which values force the malware to show its real behavior. Note that, without using SYMBION, the amount of SimProcedures that a user needs to write to reach the `TP` would have been considerably larger. Now, we symbolically execute the program until the first instruction of the malicious behavior: `0x4214e4`. At the end of the exploration, we ask the constraint solver to concretize the symbolic variables set by the `GetProcessAffinityMask` SimProcedure. In particular, we get the value `0x2902b319` for `lpSystemAffinityMask`, which reports a value greater than 0x2 for the CPU's number of cores (the sample is checking CPU features to fingerprint virtual environments). By concretizing these solutions in the memory of the concrete process we can observe the execution of the malicious code.

We split the malware code execution in the following way:

- $\texttt{CSP}_1$ = Malware main function (`0x40fae6`).
- $\texttt{PoI}_1$ = Call to `GetProcessAffinityMask` (`0x7502a889`).
- $\texttt{TP}_1$ = Start of malicious behavior (`0x4214e4`).

VI. DISCUSSION

Program Execution Correctness. SYMBION does not keep track of the constraints needed to reach the multiple `TP` defined during the analysis. Therefore, since users have full control capabilities over the target program (they can change the memory at any time in inconsistent ways), it is possible to generate infeasible program's executions. The analyses developed on top of SYMBION are responsible to handle this problem if the program execution correctness is a strict requirement.

Desynchronized Environment Interactions. Any environment interaction made by the concrete process that involves resources outside of the process memory is not shared with the symbolic engine. For instance, consider a program that opens a file during the concrete execution, and, after that, a Concrete→Symbolic transition is performed. During the symbolic execution, if a read over that file happens, our symbolic engine uses an abstraction for that file that is completely desynchronized from the one in the concrete process. The same effect occurs in the case of open sessions with remote servers or device drivers. It is possible to support a synchronization mechanism on top of SYMBION to update the symbolic engine's state when a certain behavior (e.g., a file is opened) happens, but we consider the development of this solution outside of the scope of this work.

[2]Note that here we use a SimProcedure only to programmatically break the execution and set a symbolic buffer, conceptually implementing a breakpoint. Instead, using SYMBION we do not need to write SimProcedures that model all the environment interactions required to bring the execution to the state of interest.

Concretization Policy. After reaching the `TP`, SYMBION performs a Symbolic→Concrete transition. During this process, the symbolic variables need to be concretized, and their concrete values committed to the concrete process memory. During this process, we need to specify the subsets of variables to be concretized and committed, and also which concrete values should be used (a satisfiable SMT expression can have multiple valid solutions, by default, SYMBION, picks the first solution). We decided to not force the users to concretize all the symbolic variables but rather give them the freedom to choose the ones they think are important for the analysis.

Targets Support. Currently, SYMBION supports analyses of binaries on x86 and ARM architectures, and only a GDBTarget has been implemented. Nevertheless, we do not consider this an inherent limitation of SYMBION, because supporting more architectures only requires more engineering effort.

VII. Related Work

Applying symbolic execution only to specific parts of a program has been tackled by researchers in the past.

Chipounov et al. [14] proposed S^2E, an automated path explorer that lets users define which part of a program has to be executed symbolically and which part as to be executed concretely. Cha et al. [7] developed Mayhem, an AEG (automatic exploit generation) system that is based on strict cooperation between a Concrete Executor Client (CEC) that runs the code inside a virtual machine, and a Symbolic Executor Server (SES) that symbolically explores traces provided by the CEC. Inputs generated by the SES are sent back to the CEC to increase the code coverage of the target binary. Differently from Mayhem and S^2E, SYMBION does not work with traces, nor defines a priori which parts of the program have to be symbolically executed.

Saudel et al. [13] proposed Triton, a dynamic binary analysis framework based on Intel's Pin [20]. The target program is instrumented using Pin, and users can use selective symbolic execution leveraging the API provided by the tool. Fioraldi developed angrdbg [23], a library to generate `angr`'s states from a concrete state of a program under analysis. Specifically, `angr` is imported inside a running instance of gdb, and after that, it is possible to create a symbolic state given the current concrete state of the program. Differently from angrgdb and Triton, in SYMBION the symbolic execution engine does not live inside a debugger, nor it is dependent on a particular dynamic execution environment (e.g., Pin or gdb).

Muench et al. [21] developed the multi-target orchestration platform Avatar^2. This platform allows for the synchronization of multiple executions and analysis targets, such as emulators, hardware, and even symbolic execution engines such as `angr`, in a target-agnostic way. To accomplish this, the system supports both the forwarding and synchronization of registers and memory between targets. The main focus of Avatar^2 is to support the analysis of embedded systems, while SYMBION provides a programmatic interface to support *interleaved symbolic execution* of complex targets.

Baldoni et al. [24] applied symbolic execution to real-world malware by implementing a tool that imports the concrete state of the malicious program inside a symbolic execution engine. SYMBION implements an idea similar to Baldoni et al., but in a completely automated way. Moreover, it is possible to carefully steer the concrete execution from a symbolic analysis to bring the execution at a specific point in the code.

VIII. Conclusions

In this paper, we showed how symbolic execution of complex programs is generally problematic because of missing environment models and missing data dependencies, which together block the possibility to perform a symbolic analysis on a target program. To solve this problem, we proposed *interleaved symbolic execution*, a novel approach to execute a target binary by interleaving concrete and symbolic execution. We implemented our approach in SYMBION, a system built on top of the `angr` framework. We demonstrated how this technique solves the presented issues, and how it supports the analyses of real-world binaries. As a further demonstration, we have been publicly notified that SYMBION is currently being used by the community for different research projects, and by a large corporation for internal projects.

IX. Acknowledgments

We would like to thank our reviewers for their valuable comments and inputs to improve our paper.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This material is based upon work supported by AFRL under Award No. FA8750-19-C-0003. This material is also based on research sponsored by DARPA under agreement number HR001118C0060. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, the U.S. Government, or the other sponsors.

References

[1] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later." *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[2] 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 (NDSS)*, 2016.

[3] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *Proceedings of the International Conference on Information Systems Security*. Springer, 2008.

[4] 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 *Proceeding of the IEEE Symposium on Security and Privacy (S&P)*, 2016.

[5] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Proceeding of the IEEE Symposium on Security and Privacy (S&P)*, 2007.

[6] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware." in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[7] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2012.

[8] D. A. Ramos and D. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *Proceedings of the USENIX Security Symposium*, 2015.

[9] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi, "A fistful of red-pills: How to automatically generate procedures to detect cpu emulators," in *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[10] M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti, "Detecting environment-sensitive malware," in *Proceedings of the International Symposium Recent Advances in Intrusion Detection (RAID)*, 2011.

[11] D. Kirat, G. Vigna, and C. Kruegel, "Barecloud: Bare-metal analysis-based evasive malware detection," in *Proceedings of the USENIX Security Symposium*, 2014.

[12] M. Polino, A. Continella, S. Mariani, S. D'Alessio, L. Fontana, F. Gritti, and S. Zanero, "Measuring and defeating anti-instrumentation-equipped malware," in *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2017.

[13] F. Saudel and J. Salwan, "Triton: A dynamic symbolic execution framework," in *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*. SSTIC, 2015.

[14] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," in *ACM SIGARCH Computer Architecture News*. ACM, 2011.

[15] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, "Chopped symbolic execution," in *Proceedings of the ACM International Conference on Software Engineering (ICSE)*, 2018.

[16] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, 2008.

[17] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[18] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[19] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *ACM SIGSOFT Software Engineering Notes*. ACM, 2005.

[20] Intel, "Pin - A Dynamic Binary Instrumentation Tool," https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, 2019.

[21] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "Avatar 2: A multi-target orchestration platform," in *Proceedings of the Workshop on Binary Analysis Research (BAR)*, 2018.

[22] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *International Static Analysis Symposium*. Springer, 2011.

[23] A. Fioraldi, "Symbolic Execution and Debugging Synchronization," Bachelor's thesis, Sapienza University of Rome, October 2018.

[24] R. Baldoni, E. Coppa, D. C. D'Elia, and C. Demetrescu, "Assisting malware analysis with symbolic execution: A case study," in *International Conference on Cyber Security Cryptography and Machine Learning*. Springer, 2017.