Towards Detecting Trigger-based Behavior In Binaries: Uncovering the Correct Environment[⋆]

Dorottya Papp[1,2][0000−0002−9976−614X], Thorsten Tarrach[2][0000−0003−4409−8487], and Levente Buttyán[1][0000−0003−4233−2559]

[1] CrySyS Lab, Dept. of Networked Systems and Services, BME
{dpapp,buttyan}@crysys.hu
[2] AIT Austrian Institute of Technology GmbH.
thorsten.tarrach@ait.ac.at

Abstract. In this paper, we present our first results towards detecting trigger-based behavior in binary programs. A program exhibits trigger-based behavior if it contains undocumented, often malicious functionality that is executed only under specific circumstances. In order to determine the inputs and environment required to trigger such behavior, we use directed symbolic execution and present techniques to overcome some of its practical limitations. Specifically, we propose techniques to overcome the environment problem and the path selection problem. We implemented our techniques and evaluated their performance on a real malware sample that launches denial-of-service attacks upon receiving specific remote commands. Thanks to our techniques, our implementation was able to determine those specific commands and all other requirements needed to trigger the malicious behavior in reasonable time.

Keywords: Directed symbolic execution · Trigger-based behavior · Software verification.

1 Introduction

Trigger-based behavior is the execution of undocumented, potentially malicious features in an application upon reception of some inputs that satisfy pre-defined criteria. Such inputs are referred to as *trigger inputs*. The pre-defined criteria are hard-coded into the application in the form of checks and their semantic meaning

[⋆] The presented research has been partially supported by the SETIT Project (no. 2018-1.2.1-NKP-2018-00004), which has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.2.1-NKP funding scheme, and by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications). It has also been supported by the SECREDAS project, which receives funding from ECSEL Joint Undertaking under Grant Agreement No 783119. This Joint Undertaking received support from the European Union's Horizon 2020 Research and Innovation Programme.

can encompass all sorts of external requirements, e.g. specific system time or location, special text entered or message received. While not all instances of trigger-based behavior are malicious (take, for example, software easter eggs[3]), such behavior is often used by malware. For example, malware can evade in-depth analysis by scanning its environment and ceasing malicious activities if it finds hints of an analysis framework[4]. Trigger-based behavior also includes backdoors, a behavior prevalent in firmware images [9], in which case, special access is granted, if a specific string is received as input. These examples show that in many cases, the application to be analyzed is only available in binary form. Therefore, in this paper, we consider applications available as binaries. Due to the often malicious intent behind the implementation of trigger-based behavior, its detection is important. However, the combination of inputs required to trigger the hidden behavior is known only to its author, therefore, uncovering such behavior via testing is challenging.

Previous work in this field [4,10,12] have demonstrated the usefulness of symbolic execution [3] to uncover trigger-based behavior. Symbolic execution was originally developed to automate testing by analyzing execution paths and generating test cases, which lead execution down the analyzed execution path. In order to uncover trigger-based behavior, we need to analyze the application's interaction with its environment and how the environment influences its behavior. If data from the environment is replaced with symbolic variables, symbolic execution can analyze this interaction and can obtain the hard-coded conditions together with the trigger input values satisfying those conditions.

However, using symbolic execution has a limitation: the more symbolic variables are introduced into the analysis, the more execution paths must be analyzed, leading to the *path explosion problem*. Previous work addressed this problem by considering only a subset of potential trigger input types. In [4], for example, the human analyst is required to select possible trigger input types in advance. However, as only the malware author knows the exact trigger inputs, there is a chance that the human analyst fails to select all necessary types of input. In [10], the authors describe a technique that works on Android Bytecode but only consider time, location and SMS objects as trigger inputs. In [12], a lightweight version of symbolic execution is performed over JavaScript code, which analyzes the effects of potential values in the navigator's fields.

In this paper, we want to ovecome the path explosion problem without limiting the trigger input types. Our goal is to develop an approach, which can consider all external data as potential trigger inputs while relying on symbolic execution to calculate the inputs and environment required to reach a selected program point. The overview of our main idea is shown in Fig. 1. We assume that the analyzed binary is deterministic and interacts with the environment through the operating system and its API (system calls). Therefore, we consider invoked library functions as part of the analyzed binary. In real-life execution, the binary

[3] `https://electrek.co/2017/12/23/tesla-christmas-easter-egg/`

[4] `https://www.fireeye.com/blog/threat-research/2011/01/the-dead-giveaways-of-vm-aware-malware.html`

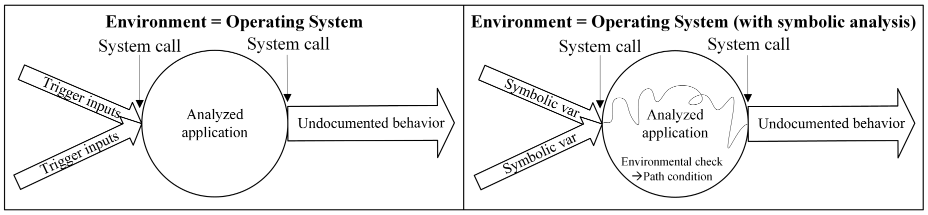

Fig. 1. Symbolic Execution for Uncovering Trigger-based Behavior

would invoke multiple system calls and the return values from a subset of those system calls would be interpreted by the binary as trigger inputs. The binary would then proceed to match those return values against the pre-defined criteria hard-coded into its logic and execute the potentially malicious behavior only if the result of the comparison(s) is a match. In order to analyze this interaction, the return values of system calls that return data from external sources must be replaced with fresh symbolic variables. Then, symbolic execution can be used to analyze this interaction.

Our contributions in this paper are the following:

1) We present an approach for uncovering trigger-based behavior in binaries, which is capable of considering all external data sources as trigger input types. Our approach replaces system calls with symbolic summary functions, which return fresh symbolic variables instead of external data.
2) Our approach relies on directed symbolic execution [14] to guide analysis towards a selected program point. However, directed symbolic execution expects a semantically correct and complete interprocedural control-flow graph. The generation of such a control-flow graph is a challenge for binary programs, mainly due to indirect jumps. Our approach is designed such that directed symbolic execution can be performed even if the interprocedural control-flow graph has incorrect/missing edges and/or nodes.
3) We implement our approach in `angr` [18]: we model 36 system calls for Linux and discuss modifications to `angr`'s workflow in order to make our approach feasible in practice.
4) We evaluate our approach on a real malware sample compiled for the ARM platform, which is known to exhibit trigger-based behavior. The program logic of the selected sample contains elements known to be challenging for symbolic execution and its execution relies on multiple sources of environmental input. Our approach is able to reach program points deep in the binary and obtain the environmental conditions required to trigger their execution. In addition, our analysis time is in the order of hours, which is a reasonable performance considering the complexity of the analyzed sample and the generality of our approach.

The paper is structured as follows. Section 2 provides an overview of symbolic execution: the main idea behind the technique, its limitations and current approaches to overcome those limitations. Section 3 discusses our approach to uncover environmental conditions without a priori assumptions about trigger input types. The implementation of the proposed approach is discussed in Section 4. In Section 5, we evaluate our approach on a real malware and discuss both its performance and the recovered environmental constraints. Section 6 concludes the paper and outlines future research directions.

2 Background

In this section, we discuss the concept of symbolic execution. The techniques has been well-researched over the years and as such, a full survey of the field is out of scope for this paper. We only summarize its main characteristics and discuss the challenges it poses for our research. Readers interested in a full overview of this field are kindly refered to [3] and [17].

Symbolic execution is an analysis technique originally proposed to automatically generate test cases and increase code coverage during software testing. During symbolic analysis, registers and memory addresses do not store exact values but instead special symbols called symbolic variables. When first introduced into the analysis, symbolic variables may take on any value, i.e. they are *unconstrained.* When analysis reaches a branch in the analyzed software, two execution paths are spawned for both sides of the branch, i.e. it *forks.* In each spawned execution path, constraints are placed on the symbolic variables to represent the chosen path. The set of constraints collected on an execution path is the *path constraint.* An execution path is *satisfiable*, if there exists an assignment to its symbolic variables such that the path constraint is satisfied. If no such assignment exists, the execution path is said to be *unsatisfiable.*

The challenges of performing symbolic execution on arbitrary software in binary form are manifold. Firstly, tools implementing the technique have to model the execution state on the platform the analyzed software is supposed to run on, including instruction set, registers, memory, interrupts, calling conventions, flags, etc. Tools implementing symbolic execution, e.g. DART [11], KLEE [5], S2E [8], MAYHEM [7] and angr [18], come with such a model of the target platform. Secondly, symbolic execution can only reason about code it analyzes and has no knowledge about library functions, system calls and their side effects. This challenge is better known as the *environment problem* and is typically tackled using summary functions, which are pieces of code that summarize the effects of the missing piece of code. Thirdly, as symbolic execution spawns execution paths to pursue at each encountered branch; the number of execution paths to analyze is exponential with respect to the number of conditional branches in the analyzed software. This challenge is known as the *path explosion problem* and it results in symbolic execution not being able to exhaustively explore all execution paths in all but the simplest of cases. This challenge is partially tackled by specifying which parts of the software are of interest to the analysis and only executing those parts symbolically. In such scenarios, the analysis engine keeps track of not only the symbolic state, but the concrete execution state as well, earning the name mixed concrete and symbolic execution. Lastly, since not all execution paths can be explored during symbolic execution, analysis has to decide which paths to pursue. This challenge is known as the *path selection problem* and it is usually tackled using a heuristic exploration strategy. The depth-first strategy explores an execution path to its completion before backtracking to the second deepest branch. The breadth-first strategy, on the other hand, seeks to explore all execution paths in parallel. There are also randomized approaches, where the next pursued path is selected randomly or with some probability. In certain application domains of symbolic execution, path selection algorithms have been tailored for a specific goal, e.g. maximizing coverage [5,13] or reaching a certain program point [14,16].

We use angr, which is capable of mixed concrete and symbolic execution and has a model for the ARM platform. However, angr in itself does not solve the environment and the path selection problems. A major part of our work was to address these problems, and in Section 3, we describe how we did so.

3 Methodology

Our methodology focuses on how to calculate the correct environmental conditions such that a certain behavior implemented by the analyzed malware can be triggered. We assume that the human analyst has a specific program point of interest and wishes to uncover the inputs required to trigger its execution. Towards this end, we employ two techniques:

1) Symbolic summary functions capturing the behavior of invoked system calls in order to introduce a model of environmental data to the analysis, and
2) Shortest-distance symbolic execution [14], a path selection strategy to guide analysis towards the selected program point.

We elaborate on these techniques in Sections 3.1 and 3.2, respectively.

3.1 Symbolic Summary Functions

As mentioned before, the environment is represented by operating system services, and the environment manifests itself as the result of invoking system calls. Therefore, we need symbolic summaries of system calls which model their effects. Such symbolic summary functions allow us to simulate the environment for the analyzed application and enable mixed concrete and symbolic execution to analyze how returned data influences execution.

Our summaries are semantically equivalent to the system calls they replace with two major exceptions. Firstly, if the system call writes into the environment (e.g. sends packets or writes in a file), the summary always returns with success. This allows us to contain the path explosion problem: if we simulated both success and failure, we would need to simulate the various conditions for failure, which would further increase the number of execution paths to analyze. However, we acknowledge the possibility of system call failures being used as triggers. Secondly, if the system call returns data from the environment (e.g. assigned process ID, system time, messages over the network), the summary function returns fresh symbolic variables instead. Using the fresh symbolic variables, the influence of the environment on the application can be analyzed.

Symbolic summaries can be written based on the semantic information available about the system calls in the operating system's documentation. These summaries need to be written only once for a particular platform. As an example, let us consider the Linux system call `fork`, responsible for duplicating processes. On success, it returns the PID of the child process in the parent and 0 in the child. On failure, it returns -1 to the parent, creates no child process and sets `errno` appropriately. In order to explore how the invocation of `fork` influences the analyzed binary, we need to replace its return value with a fresh symbolic variable. According to its manpage[5], its return value has the type `pid_t` which is a signed integer. On the ARM platform, a signed integer is 32 bits long, therefore, the model of this system call for analyzing ARM binaries must return a 32-bit long symbolic variable. The variable must be constrained as written in the documentation: it can be a positive number, 0 or -1. Two further constraints must be added to the model to capture its behavior faithfully. Firstly, if the return value is greater then 0, than semantically, analysis continues in the child process. Therefore, the PID and the parent PID of the execution state must be updated accordingly. Secondly, if the return value is -1, then semantically, the system call failed and a new symbolic variable is required to represent the error condition, and its value must be constrained to one of the potential error codes.

3.2 Approach to Symbolic Execution

Symbolic summary functions only introduce the model of environmental data in the form of fresh symbolic variables. The actual conditions required to trigger a specific behavior in the analyzed binary are encoded in its instructions. In order to calculate the correct environmental values, we need to recover and solve these conditions. To this end, we use mixed concrete and symbolic execution, capable of both recovering these conditions as path conditions and solving them thanks to Satisfiability Modulo Theory solvers. Specifically, we employ shortest-distance symbolic execution (SDSE) [14], designed to prioritize execution paths which are closer to a selected target according to some metric.

SDSE was originally proposed to solve the line reachability problem: how to reach a target line in the source code? It requires the interprocedural control-flow graph in order to guide symbolic execution towards the targeted line. The approach first translates execution paths to control-flow graph nodes, then computes the shortest distance from said nodes to the node corresponding to the target line. The computed metric is used as scores to prioritize execution paths. At branches, SDSE selects the execution path with the lowest score among all available paths for analysis.

Our scenario is similar to the one SDSE was developed for in the sense that we need a solution for the reachability problem in order to recover constraints placed on environmental data. However, there are key differences as

[5] `http://man7.org/linux/man-pages/man2/fork.2.html`

well. Firstly, SDSE was originally proposed and implemented at the source code level, while we apply it at the binary level. As a result, instead of a target line, we aim to reach a target binary instruction. Secondly, as stated in [14], SDSE can only work correctly, if the interprocedural control-flow graph recovered from the binary does not have mismatching calls and returns. Otherwise, semantically incorrect or infeasible paths may be computed as shortest paths, resulting in incorrect scores and priorities. In order to generate a semantically correct control-flow graph whose structure properly captures function calls and returns encountered during execution, the generator algorithm has to consider a lot of context-related information, including call sites, return sites and the call stack. There exist algorithms capable of handling that information [6,18], however, their usage in practice poses a challenge. As more context-related information is taken into consideration, the time and space required to generate and store the resulting control-flow graph also increases exponentially. Instead of generating such a control-flow graph, we implemented a heuristic algorithm to discard edges whose inclusion in the shortest path calculation might result in incorrect paths. This heuristic allows us to keep the required contextual information at a minimum by taking into consideration potential changes to the call stack at edges that result in semantically correct function calls and returns. We discuss the implementation of this heuristic in Section 4.2.

4 Implementation

We implemented our approach in angr (version 7.8.2.21), an open-source binary analysis tool written in Python, capable of analyzing binary formats of major operating systems, such as ELF, PE and Mach-0 files. The tool implements many analyses for binary code, including mixed concrete and symbolic execution, constraint solving, control-flow graph generation, program slicing, dependency analysis, etc. These analyses are performed over the intermediate representation (IR) of valgrind [15], called VEX, to provide platform independence. VEX translates a sequence of binary instructions into a block of IR instructions. As a result, most analyses are not performed on a per instruction basis, but rather on a per IR block basis. Our implementation uses the following features of angr:
1) mixed concrete and symbolic execution engine with a constraint solver,
2) control-flow graph generation, and
3) model of execution states, including registers, memory, and elements from POSIX, such as files and sockets.

There were cases, in which we needed to modify the workflow and execution of angr. We discuss these modifications in the rest of this section.

4.1 Symbolic Summaries for System Calls

angr supports system call invocations during mixed concrete and symbolic execution. However, developers focus more on defining the environment at the library level and therefore, the tool has more symbolic summaries for standard libc functions than it has for system calls. As a result, many system calls invoked during our tests were missing and had to be added to the tool manually. The list of 36 system calls we had to create symbolic summaries for is shown in Tab. 1.

Table 1. System Calls on Linux for Which Symbolic Summaries Were Created

`_newselect`	`arm_set_tls`	`brk`	`clone`	`close`	`connect`
`exit`	`exit_group`	`fcntl`	`fcntl64`	`fork`	`futex`
`geteuid32`	`getgid32`	`getpid`	`getppid`	`gettimeofday`	`getuid32`
`ioctl`	`kill`	`mmap2`	`nanosleep`	`open`	`read`
`recv`	`rt_sigaction`	`rt_sigprocmask`	`sendto`	`setrlimit`	`setsockopt`
`socket`	`time`	`ugetrlimit`	`uname`	`wait4`	`write`

4.2 Control Flow Graph

There are two algorithms to generate an interprocedural control-flow graph in angr. The first algorithm is called CFGFast and it relies on heuristics and assumptions to greatly decrease the time required for generation. The second algorithm is called CFGAccurate (CFGEmulated in later versions) and it performs lightweight symbolic execution to generate the control-flow graph, increasing accuracy. In our implementation, we used CFGAccurate as accuracy is important for using SDSE.

Extending the Control Flow Graph There are program constructs which pose a challenge during control-flow graph generation, e.g. indirect jumps. We encountered scenarios where CFGAccurate detected the indirect jumps but it was unable to accurately determine the address the analyzed code jumped to. The limitation is caused by the lightweight nature of its symbolic execution: if a read or write operation involves an operand which could be assigned multiple values, that operand is skipped and a fresh, unconstrained symbolic variable is used instead. However, angr's symbolic execution has an upper limit on the number of successor states it generates when analyzing an execution state. If the instruction pointer of the analyzed execution state has more than 256 solutions (by default), then the tool assumes that the instruction pointer was overwritten with unconstrained data, and flags the execution state as one producing unconstrained successors.[6] As a result, CFGAccurate may fail to analyze certain parts of the binary due to the inaccurate execution state used during construction. This scenario is illustrated with the following two instructions:

```
ldr r4, [r3, #4]    ; load function address from memory
blx r4              ; call function
```

[6] This assumption is included in angr's documentation together with the fact that it is not sound in general.

The code includes a call to the address contained in `r4`, whose value is loaded from memory. The address from where the value is to be loaded is influenced by `r3`. If `r3` holds an operand with multiple potential values while control-flow recovery analyzes this code segment, then analysis has to read a multi-valued operand from the register. However, as discussed before, instead of performing the read, the recovery algorithm creates a fresh, unconstrained symbolic variable to represent the result of the read operations. As a result, `r4` will also hold an unconstrained symbolic variable when the recovery algorithm tries to determine the jump address. Because the unconstrained symbolic variable has more than 256 solutions, the state is flagged as one producing unconstrained successors and address resolution fails.

Normal mixed concrete and symbolic execution, however, never skips operands and is much less likely to run into such a scenario. Execution states have operands with semantically correct values and correct path constraints. If control-flow graph generation is resumed from such a state, CFGAccurate can accurately identify the indirect jump addresses, if the value of `r4` has less than 256 solutions. Therefore, during control-flow graph generation, we take note of addresses where unconstrained successors were computed as potential extension points of the control-flow graph. When normal mixed concrete and symbolic execution reaches such an address, we use the accurate execution state to extend the control-flow graph on the fly.

Shortest Path Calculation The accuracy of CFGAccurate is influenced by its level of context-sensitivity. This parameter captures how deep the call stack is taken into consideration when determining the calling context of any given control-flow graph node. By default, the algorithm analyzes each address only once per distinct calling context. As a result, different levels of context sensitivity result in different graph structures, which in turn influence the available paths computed by generic shortest path algorithms. Fig. 2 shows the different contexts in which functions are analyzed at different levels of context sensitivity. Because of the different contexts, functions may be replicated multiple times in the control-flow graph. Note, that we demonstrate the effect of context sensitivity at the source code level only for ease of understanding, but our techniques work at the binary level.

In order for generic shortest path algorithms to compute semantically correct paths in the interprocedural control-flow graph, edges connecting mismatched call sites and return sites must be discarded. CFGAccurate can record the execution state from which a specific control-flow graph node was created, allowing access to its call stack. The algorithm also annotates edges with attributes recovered by VEX during lightweight symbolic execution. One of these attributes is the semantic meaning of the jump at the end of each IR block (e.g. function call, return, etc.). Inspired by the control-flow graph model of [2], a visibly pushdown automaton which keeps track of the calling context of functions, we rely on the call stack and the edge annotation to implement a heuristic that discards semantically incorrect paths violating the following rules:

```
void c(){
    printf(”Executing function c”);
}
void b(){
    printf(”Executing function b”);
    c();
}
void a(){
    printf(”Executing function a”);
    c();
}
int main(int argc, void* argv) {
    a();
    b();
    return 0;
}
```

	0-context sensitivity	1-context sensitivity	2-context sensitivity
a	a	main→a	(library init)→main→a
b	b	main→b	(library init)→main→b
c	c	a→c b→c	main→a→c main→b→c
printf	printf	a→printf b→printf c→printf	main→a→printf main→b→printf a→c→printf b→c→printf

Fig. 2. Different Contexts of Functions During Control-flow Graph Generation

1. The call stack depth difference between the source node and the destination can only change by -1, 0 or 1, corresponding to returning, staying in the function or calling another function, respectively.
2. In case of calls and returns, the edge's attributes must support the deduction made from the call stack depth difference. For example, if the call stack depth difference is 1, then the edge's attributes must state that the edge represents a function invocation.

If any of the above rules is violated, the edge is discarded during the shortest path calculation.

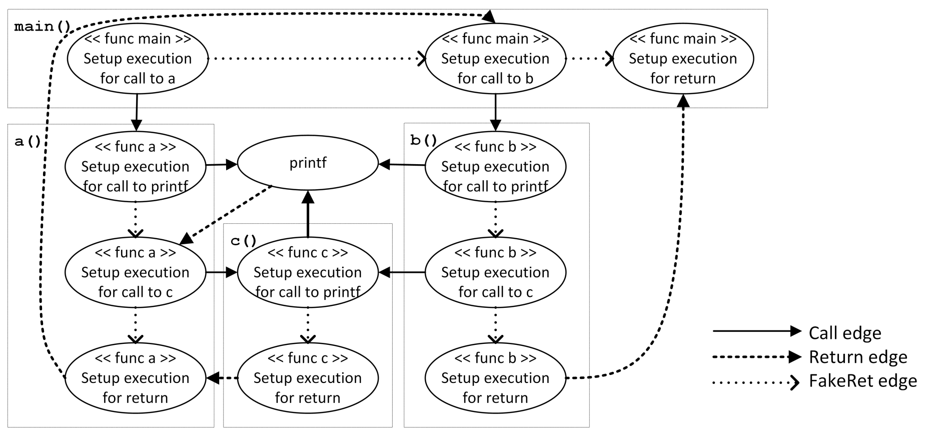
main()
<< func main >> Setup execution for call to a
<< func main >> Setup execution for call to b
<< func main >> Setup execution for return
a()
<< func a >> Setup execution for call to printf
printf
b()
<< func b >> Setup execution for call to printf
<< func a >> Setup execution for call to c
c()
<< func c >> Setup execution for call to printf
<< func b >> Setup execution for call to c
<< func a >> Setup execution for return
<< func c >> Setup execution for return
<< func b >> Setup execution for return
Call edge
Return edge
FakeRet edge

Fig. 3. Fake Return Edges in an Interprocedural Control-Flow Graph

Our approach can rely on generic shortest path algorithms thanks to special, so called *fake return* edges. These edges are directed edges from the call site to the return site and are automatically added by `angr` whenever a call is encountered. Their importance is highlighted in Fig. 3, which shows the fake return edges in the interprocedural control-flow graph of the source code shown in Fig. 2 when context sensitivity level is set to 0. For the sake of clarity, the actual instructions responsible for setting up the execution state for calling functions were omitted. By default, CFGAccurate analyzes each IR block once per distinct calling context. With 0 context sensitivity level, the calling context is only the currently analyzed function, which leads to each function being present in the graph exactly once. For each analyzed block, `angr` adds a call edge to the called function and a fake return edge to the return site. These special edges mainly serve the purpose of ensuring connectivity in the graph. Because each block is analyzed once per distinct calling context, each function has only 1 return edge. For example, consider the `printf` function. Even tough it is called from `a`, `b` and `c`, it is analyzed only once, the first time it is encountered when called from `a`. As a result, `printf` has only 1 return edge, leading to its return site in `a`. Without fake return edges, `printf`'s call site in `b` would not be connected to its return site in `b`.

Our edge discarding heuristic can also lead to loss of connectivity without fake return edges. For example, our heuristic discards the call edge between `c` and `printf` because the control-flow graph nodes' call stack depth does not support a function call. The call site has the context `main`→`a`→`c`, while `printf` has a the context `main`→`a`→`printf`. Because the call stack depth difference is 0, the edge should indicate staying in the function instead of calling another function. Without the discarded call edge, generic shortest path algorithms must rely on the fake return edge to calculate shortest paths. However, even if the fake return edge is used, simulation must execute the function represented by the said edge. In order to faithfully capture the cost of calling a function, we assign weights to fake return edges: the smallest number of IR blocks simulated between the call and return sites throughout analysis, i.e. the shortest path mixed concrete and symbolic execution uncovered. Thanks to this heuristic, we are able to keep context sensitivity at level 1.

4.3 Call Stack Management

During our work, we discovered mismatches between how the call stack is managed in CFGAccurate and how it is managed during mixed concrete and symbolic execution. The discrepancies between the algorithms hinders us in translating execution states into control-flow graph nodes.

In case of mixed concrete and symbolic execution, function calls are detected by statically looking at the semantic information about the jump at the end of the analyzed IR block. Function returns, on the other hand, are detected by looking at the stack pointer. The function returns if either the stack pointer has a lower value than it had at the call (which is the convention in e.g. Intel platforms) or execution has reached the return address recorded at the call and the stack pointer has the same value as it had at the call (which is the convention in platforms like ARM where the return address is stored in the link register).

CFGAccurate uses the same approach with an additional feature. For each IR block address encountered during CFG construction, it checks with `angr`'s loader whether the address corresponds to a symbol. If it does, it forcefully simulates a call to that symbol. This approach has the advantage of providing more meaningful nodes in the control-flow graph. However, it hinders us from accurately matching execution states to control-flow graph nodes as the calling contexts are different. As an example, consider the following instructions:

```
000105a4 <getspoof>:
   ...
   105bc: eb0022aa  bl 1906c <rand>
   ...
0001906c <rand>:
   1906c: ea000065  b 19208 <__GI_random>
   ...
00019208 <__GI_random>:
   ...
```

The `getspoof` function at `0x105bc` calls `rand`, which immediately jumps to `__GI_random`. In case of symbolic execution, the execution state at `0x19208` has the calling context `getspoof`→`rand`, while the control-flow graph node representing `0x19208` has the context `getspoof`→`rand`→`__GI_random`, because `0x19208` corresponds to a symbol. Due to the different calling contexts, the execution state cannot be translated to the control-flow graph node. Thus, we removed the forceful simulation of function calls from CFGAccurate.

We have also encountered call stack management issues in scenarios where mixed concrete and symbolic execution forks in functions with only one of the paths returning. The issues are caused by `angr` running its call stack management code before adding path constraints to the state. We illustrate the problem with an example. Consider the following snippet from the `strcasecmp_l` function.

```
   179c4: lsl r3, r3, #1     ; increment index for string1
   179c8: lsl r0, r0, #1     ; increment index for string2
   179cc: ldrsh r3, [lr, r3] ; load next char of string1
   179d0: ldrsh r0, [lr, r0] ; load next char of string2
   179d4: subs r0, r3, r0    ; compare the chars
   179d8: popne {pc} ; (ldrne pc, [sp], #4)
   179dc: ldrb r3, [ip], #1
```

The function iterates over two strings character by character to check whether they are equal. The comparison between two characters is implemented using subtraction. If the result of the subtraction is 0, i.e. the characters are the same and the function continues, otherwise, it returns. If any of the input strings consists of symbolic variables as characters, the comparison has two outcomes: equals and not equals. At the end of simulating the block starting at `0x179c4`, angr forks and creates the two successor states, one at 0x179dc and another at the return site. It then proceeds to check whether any of these states returned. However, the path condition has not been added to the successors yet, therefore, the stack pointer of the state at the return site is a symbolic expression encoding both staying in the function and returning. As a result, the call stack management code cannot deduce that the state returned and fails to pop strcasecmp_l from the call stack. To overcome this issue, we concretize the stack pointer after forks and re-run the call stack management code to get correct call stacks.

4.4 Model of the Execution State

In order to model the side effects of system calls and any additional data they might return, we extended the original execution state model provided by angr. The extended model includes additional POSIX elements on a per-path basis, such as group ID, thread ID and parent process ID.

We also modified how system time is tracked throughout mixed concrete and symbolic execution. Originally, angr used a monotonically increasing, global symbolic variable to model system time which is suitable for the default breadth-first exploration strategy. However, SDSE's prioritization strategy can backtrack to an earlier execution state, which semantically means taking us "back in time". In order to support such a backward flow of time, we model system time on a per-path basis with local symbolic variables.

Throughout mixed concrete and symbolic execution, we also monitor the execution state to detect whether branches are the result of references to uninitialized memory addresses. This scenario can be the result of a bug in the analyzed binary, but might also signal missing side-effects of system call models. As a result, we do not pursue such paths any further, but keep them separated from the rest of execution states for further analysis.

5 Evaluation

We evaluated our approach on a slightly modified sample from the Kaiten [7] malware family. Kaiten variants are Trojan horses which open backdoors on various platforms and perform malicious tasks when remotely instructed to do so. Our sample implements its own IRC protocol parser and expects remote commands to be delivered as IRC private messages. Some commands are used to launch denial-of-service attacks, execute shell commands and download files.

We chose this sample because its execution relies heavily on its environment. In order to trigger any malicious behavior, the sample must be able to communicate over the network. It needs to connect to the IRC server at the preprogrammed address and log into the also preprogrammed IRC channel. The sample uses randomly generated strings as nick and user name in the IRC communication; the seed is calculated from the system time, the process ID and the

[7] https://www.symantec.com/security-center/writeup/2015-102008-3612-99?tabid=2

parent process ID. Once connection to the IRC channel has been established, the correct IRC private message must be received in order to trigger any behavior implemented in the sample.

Our chosen sample poses two challenges. Firstly, due to our assumptions and the sample's implementation, a vast number of execution paths are available for analysis. There are three main sources for such a high number of paths:

1) Environmental data. The sample relies on the system time, process IDs and communication over the network. As we assume no prior knowledge about its functionality, our analysis has to analyze all those inputs using symbolic variables, leading to many branches.
2) String handling. The sample implements an IRC protocol parser and uses standard libc functions such as `strlen`, `strtok` and `strcasecmp` to manipulate the string messages received over the network. These functions typically loop over the string character by character. As their inputs are returned from the kernel, our analysis must consider each of the characters as symbolic variables. Such loops are known to contribute to the path explosion problem.
3) Infinite loop. The sample is implemented to run in an infinite loop, continuously listening for messages from the IRC server and trying to reconnect in cases of communication failure. As a result, exploring all execution paths cannot be done in a finite amount of time.

Another challenge is in the sample's logic. In case of receiving a well-formed IRC message, the sample dispatches the message to the appropriate handler function via jump tables. These jump tables are represented in the control-flow graph by nodes with many call edges leading to different handler functions. The use of jump tables decreases the accuracy of shortest path calculation, as the shortest path is always to take the correct call edge, even if said edge is infeasible.

5.1 Setting Up Our Experiment

Modifications to the Sample Before we applied our implementation to the chosen sample, we made a few modifications to it which we describe here. First, we downloaded its publicly available source code[8]. Then, we shortened all strings in the jump tables of the source code to contain only a single character and the terminating null. With this modification, we can contain the path explosion of looping over strings to a certain extent. Note, however, that the modified sample still includes multiple jump tables organized into layers with each layer requiring multiple characters with specific values. Therefore, even with this modification, the sample still requires a string with multiple characters to invoke the necessary handler functions. We also set the address of the IRC server to `127.0.0.1` in order to avoid symbolically analyzing a DNS lookup. Finally, we recompiled the modified source code for the ARM platform and performed our analysis on the resulting binary. Both the original and the modified source code are available as supplementary materials [1].

[8] `https://packetstormsecurity.com/files/25575/kaiten.c.html`

Target Behavior As the target behavior, we selected one of the functions launching denial-of-service attacks (`tsunami` in the source code). The attack is executed in a child process and sends spoofed packets to the target IP specified in the command. We inserted a call to the `kill` libc function before the child process is created and set the underlying `kill` system call as our target. Note, that this system call is used in other functions as well, therefore, we only accept reaching it, if it is done via the `tsunami` function.

In order to reach this function, mixed concrete and symbolic execution has to simulate the communication with the IRC server and “send” a specific string to the sample. The string must meet the following requirements:

1) The sample must interpret its first part as an IRC private message, i.e. it must start with the corresponding code from the jump table of IRC message-handling functions (`4` in our case).
2) It must contain the preprogrammed name of the IRC channel to which the sample logged into (`#` in our modification).
3) It must be intended for the sample, either by specifically mentioning the sample's IRC nick (randomly generated) or by using a wildcard character.
4) The sample must interpret its last part as a command for launching the DoS attack implemented in `tsunami`, i.e. it must contain the corresponding code from the jump table of command-handling functions (`0` in our case).

Unfortunately, while generating the control-flow graph with context sensitivity level 1, `angr` did not flag the IR blocks implementing the jump tables as producing unconstrained successors. As a result, jump tables were not treated as potential extension points, forcing us to specify the missing edges manually.

Parameters of the Machine We ran the sample on a machine with two Xeon E5-2680 CPUs of 10 cores each, running at 2.8 GHz. The machine has 378 Gb of RAM available. Note that `angr` is not multithreaded and uses only a single core. We also restricted `angr` to run with 100 Gb of memory.

5.2 Results

Table 2. Runtime Performance of Each Stage of Approach on Modified Kaiten Binary Sample

Stage	Runtime (hh:mm:ss)
Control-flow graph generation and extension	0:10:42
Simulation of execution paths	19:08:54
Shortest distance calculation	8:05:44
Other management tasks	5:05:11

Runtime Performance Tab. 2 shows the performance of our prototype implementation on the modified Kaiten binary sample. The execution path reaching

the targeted program point at the source code level is available as supplementary material [1]. The execution time of a single run consists of four components:
1) generation and extension of the control-flow graph,
2) simulating execution paths,
3) calculating scores during backtracking, and
4) other management tasks, e.g. concretizing stack pointers when necessary, logging events, checking if our target was reached, etc.

The measured execution time of our analysis was 32.5 hours. Most of the time was spent with either simulating execution paths or calculating shortest distances.

The execution time of simulating execution paths can be accredited to the logic of the sample. During our tests, analysis encountered addresses, whose simulation took hours for mixed concrete and symbolic execution. These addresses were part of libc, including `rand` and multiple string-manipulating functions whose simulation involved computations with complex symbolic values. `rand` is used by the modified sample to generate random 1-character-long strings for communication with the IRC server. While the generated string for the nick has to be analyzed in order to reach the target system call, its value does not matter: the symbolic string representing network input either matches it, or it does not. Therefore, we replaced `rand` with `angr`'s built-in symbolic summary and used a fresh, unconstrained symbolic variable to represent its result. However, the results of string manipulations contribute directly to the execution path leading towards the selected target behavior: they affect how long the symbolic string representing network input is and what constraints are placed on its characters. Therefore, we did not influence the execution of string manipulations and settled for the increased execution time.

Recovered Path Condition The execution state which first reached the target system call had 76 constraints, encoding the network conditions and the remote command required to trigger the target behavior. We checked their correctness manually by looking at the source code.

Depending on their complexity, some constraints are intuitive to interpret. For example, `<Bool socket_retval_23127_32 == 0x3>` can be interpreted as the requirement for successfully creating sockets. `socket_retval_23127_32` is the symbolic variable introduced in the `socket` system call. The two numbers are appended by `angr`: the first is a unique identifier, while the second is the length of the variable in bits. The return value of `socket` in case of success is a file descriptor (positive integer) and -1 in case of failure, it is -1. Given that the right-hand side of the equation is positive, we can deduct that the sample invoked the system call to create a socket which had to be completed successfully.

The human interpretation of other constraints, however, is quite challenging due to their complexity. For example, our modified sample sets an upper limit of 4096 on the number of characters it reads from a socket in one go. Therefore, our symbolic summary of `recv` returns a 4096-character-long string made up of symbolic variables. The sample then invokes multiple string manipulating functions which loop over the string character by character. The corresponding binary instructions are conditional in many cases, which means that in real life, the CPU would execute them only if necessary. During simulation, however, one of their operands is a symbolic character and therefore, they cannot be skipped. Instead, when possible, their results are encoded into `If-Then-Else` structures: if the flag evaluates to true, then the result is the `Then` value, else the `Else` value. These structures can be nested into each other, leading to constraints whose evaluation is tedious manually. In such cases, the constraint solver can be used to calculate the assigned values, giving the inputs required to trigger the targeted program point.

6 Conclusion

In this paper, we proposed an approach to determine what inputs and environmental conditions must be met in order to trigger undocumented, hidden behaviors in binary programs. Our approach consists of two techniques. Firstly, we model the environment at the operating system level by providing symbolic summary functions of system calls. Our summary functions have the same number and type of arguments as their real-world counterparts, but introduce fresh symbolic variables in order to model the effects of system calls. Secondly, we use shortest-distance symbolic execution to find a feasible path to a selected program point and collect the constraints along said path to acquire insight into the required input values and environmental settings. This technique relies on a semantically correct, complete inter-procedural control-flow graph, which is often unavailable for binary programs due to indirect jumps. Therefore, our approach is designed to allow for incorrect/missing edges and/or nodes.

We implemented our approach using `angr` and evaluated it on a sample from the Kaiten malware family. The sample implements an IRC bot client, which, among other things, launches denial-of-service attacks when remotely instructed to do so. The logic of the chosen sample poses additional challenges as many of its implementation details are known to be hard to analyze symbolically. Nevertheless, our approach successfully found a feasible path within reasonable time. The path condition along that path gave additional insight as to what kind of environment is needed to trigger a specific attack.

The manual interpretation of conditions can be tedious, so we recommend to automate this process as much as possible, but we leave the details of such automated evaluation of trigger conditions for future work. Another possible future research direction is to alleviate the task of manually finding program points whose trigger condition is of interest to the human analyst. Our recommendation is to identify patterns of suspicious behaviors in an off-line manner, e.g. by syntactic analysis. Given a set of such patterns, their presence in the analyzed sample could be determined by automated static analysis and the corresponding program points could be listed as targets for our approach described in this paper. We also leave as future work the evaluation of our approach on a larger sample set. We envision a study of other malware families and their variants, studying the differences in their environmental requirements.

References

1. Supplementary materials, `https://www.crysys.hu/~dpapp/publications/files/PappTB2019sefm.zip`
2. Babić, D., Martignoni, L., McCamant, S., Song, D.: Statically-directed dynamic automated test generation. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. pp. 12–22. ISSTA '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/2001420.2001423
3. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. **51**(3), 50:1–50:39 (May 2018). https://doi.org/10.1145/3182657
4. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., Yin, H.: Automatically Identifying Trigger-based Behavior in Malware, pp. 65–88. Springer US, Boston, MA (2008). https://doi.org/10.1007/978-0-387-68768-1_4
5. Cadar, C., Dunbar, D., Engler, D.: 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. pp. 209–224. OSDI'08, USENIX Association, Berkeley, CA, USA (2008), `http://dl.acm.org/citation.cfm?id=1855741.1855756`
6. Caselden, D., Bazhanyuk, A., Payer, M., McCamant, S., Song, D.: Hi-cfg: Construction by binary analysis and application to attack polymorphism. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) Computer Security – ESORICS 2013. pp. 164–181. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40203-6_10
7. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing mayhem on binary code. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy. pp. 380–394. SP '12, IEEE Computer Society, Washington, DC, USA (2012). https://doi.org/10.1109/SP.2012.31
8. Chipounov, V., Kuznetsov, V., Candea, G.: S2e: A platform for in-vivo multi-path analysis of software systems. In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 265–278. ASPLOS XVI, ACM, New York, NY, USA (2011). https://doi.org/10.1145/1950365.1950396
9. Costin, A., Zaddach, J., Francillon, A., Balzarotti, D.: A large-scale analysis of the security of embedded firmwares. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 95–110. USENIX Association, San Diego, CA (2014), `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/costin`
10. Fratantonio, Y., Bianchi, A., Robertson, W., Kirda, E., Kruegel, C., Vigna, G.: Triggerscope: Towards detecting logic bombs in android applications. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 377–396 (May 2016). https://doi.org/10.1109/SP.2016.30
11. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 213–223. PLDI '05, ACM, New York, NY, USA (2005). https://doi.org/10.1145/1065010.1065036
12. Kolbitsch, C., Livshits, B., Zorn, B., Seifert, C.: Rozzle: De-cloaking internet malware. In: 2012 IEEE Symposium on Security and Privacy. pp. 443–457 (May 2012). https://doi.org/10.1109/SP.2012.48

13. Li, Y., Su, Z., Wang, L., Li, X.: Steering symbolic execution to less traveled paths. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications. pp. 19–32. OOPSLA '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2509136.2509553
14. Ma, K.K., Yit Phang, K., Foster, J.S., Hicks, M.: Directed symbolic execution. In: Yahav, E. (ed.) Static Analysis. pp. 95–111. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23702-7_11
15. Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 89–100. PLDI '07, ACM, New York, NY, USA (2007). https://doi.org/10.1145/1250734.1250746
16. Parvez, R., Ward, P.A.S., Ganesh, V.: Combining static analysis and targeted symbolic execution for scalable bug-finding in application binaries. In: Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering. pp. 116–127. CASCON '16, IBM Corp., Riverton, NJ, USA (2016), `http://dl.acm.org/citation.cfm?id=3049877.3049889`
17. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: 2010 IEEE Symposium on Security and Privacy. pp. 317–331 (May 2010). https://doi.org/10.1109/SP.2010.26
18. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: Sok: (state of) the art of war: Offensive techniques in binary analysis. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 138–157 (May 2016). https://doi.org/10.1109/SP.2016.17