11-2021

Concolic Execution of NMap Scripts for Honeyfarm Generation

Zhe Li
Portland State University, zl3@pdx.edu

Bo Chen
Portland State University, chenbo@cs.pdx.edu

Wu-chang Feng
Portland State University, wuchang@pdx.edu

Fei Xie
Portland State University, xie@pdx.edu

Citation Details

Li, Z., Chen, B., Feng, W. C., & Xie, F. (2021, November). Concolic Execution of NMap Scripts for Honeyfarm Generation. In Proceedings of the 8th ACM Workshop on Moving Target Defense (pp. 33-42).

Concolic Execution of NMap Scripts for Honeyfarm Generation

Zhe Li
Department of Computer Science
Portland State University
Portland, Oregon, USA
zl3@pdx.edu

Bo Chen
Department of Computer Science
Portland State University
Portland, Oregon, USA
chenbo@pdx.edu

Wu-chang Feng
Department of Computer Science
Portland State University
Portland, Oregon, USA
wuchang@pdx.edu

Fei Xie
Department of Computer Science
Portland State University
Portland, Oregon, USA
xie@pdx.edu

ABSTRACT

Attackers rely upon a vast array of tools for automating attacks against vulnerable servers and services. It is often the case that when vulnerabilities are disclosed, scripts for detecting and exploiting them in tools such as `Nmap` and `Metasploit` are released soon after, leading to the immediate identification and compromise of vulnerable systems. Honeypots, honeynets, tarpits, and other deceptive techniques can be used to slow attackers down, however, such approaches have difficulty keeping up with the sheer number of vulnerabilities being discovered and attacking scripts that are being released. To address this issue, this paper describes an approach for applying concolic execution on attacking scripts in `Nmap` in order to automatically generate lightweight fake versions of the vulnerable services that can fool the scripts. By doing so in an automated and scalable manner, the approach can enable rapid deployment of custom honeyfarms that leverage the results of concolic execution to trick an attacker's script into returning a result chosen by the honeyfarm, making the script unreliable for the use by the attacker.

CCS CONCEPTS

• **Security and privacy** → *Software security engineering*.

KEYWORDS

Symbolic Execution, Honeyfarm Generation, Nmap, Scripting Language

ACM Reference Format:
Zhe Li, Bo Chen, Wu-chang Feng, and Fei Xie. 2021. Concolic Execution of NMap Scripts for Honeyfarm Generation. In *Proceedings of the 8th ACM Workshop on Moving Target Defense (MTD '21), November 15, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3474370.3485660

MTD '21, November 15, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8658-6/21/11…$15.00
https://doi.org/10.1145/3474370.3485660

1 INTRODUCTION

When it comes to identifying and compromising targets, automation is essential in making an adversary's workflow more efficient and productive. Network tools such as `Nmap` [22] allow attackers to efficiently perform reconnaissance while tools such as `Metasploit` and `sqlmap` allow attackers to streamline exploitation of vulnerable systems that have been identified. Many offensive tools are built using modular frameworks that support extensibility via scripts, allowing developers to continuously update the capabilities of the tool. Such updates are often published immediately after new vulnerabilities are disclosed, allowing anyone (both good and bad) to locate and exploit vulnerable systems. For example, within a month of the Eternal Blue release [2], updates to attack tools allowed adversaries to leverage the flaw with devastating effects before systems could be patched. In an even more severe case, on the day the Apache Struts vulnerability involved in the Equifax breach was disclosed, identification and attacking scripts were published for it, thus allowing adversaries to instantly scan for vulnerable systems and exploit them soon after [6].

One way to slow down automated tooling is to use fake networks and servers to either trick automated tools into believing they are interacting with a real vulnerable system (such as with honeypots and honeynets) or to selectively terminate the operation of the script by denying access (such as with web application firewalls). Unfortunately, due to the massive code bases being used and the volume of vulnerabilities that are being discovered, it is difficult to keep such approaches up to date and to scale them to the number of vulnerabilities that are being disclosed. Thus, it is important that automated defenses keep up with this arms race and attempt to make some of the most common tasks an adversary relies upon more difficult and time-consuming. In particular, as reconnaissance and targeting are critical in an attack, slowing down or degrading this capability can provide defenders valuable breathing room in protecting their networks.

Towards this end, this paper describes an approach for applying concolic execution on scripts in `Nmap` that are used for performing reconnaissance and scanning. The goal is to generate responses that can allow automated defenses to trick the script into an arbitrary state within itself. The approach is driven by the observation that most `Nmap` scripts for scanning and identifying vulnerable hosts are well structured and clean. By using concolic execution to generate

responses that can fool such scripts into its various execution states, one can slow down an adversary enough to allow for vulnerabilities to be remediated. For example, returning a response that causes a script to identify a service as vulnerable could be used to set up potemkin honeyfarms [40], while returning an input that causes a script to identify a service as not vulnerable could be used at the network edge as an application firewall to stop reconnaissance.

Section 2 provides a survey on embedded scripts, and symbolic and concolic execution, particularly CRETE [10], the concolic testing tool which we utilize. Our approach specifically targets Lua, the scripting language used in Nmap, a commonly used reconnaissance tool. Section 3 describes our approach for applying concolic execution to efficiently execute Nmap scripts in order to generate sets of responses that can reach each execution state of a script. Section 5 discusses a preliminary evaluation of our approach. Section 6 provides an overview of related work while Section 7 concludes.

2 BACKGROUND

2.1 Embedded Scripting and Nmap

Scripts are commonly used in attack tools because they allow for easy and rapid programming at a higher level of abstraction and can be easily embedded within the tool to extend its functionality without recompilation [5]. Typically, such scripts are executed by an interpreter program that is supported by a host program through *system language* API calls, often called the *glue layer*. Most existing interpreters are designed to be embedded into a glue layer written in C [17]. Examples include CPython (Python and C) [4], Ricsin (Ruby and C) [32], and Luabind (Lua and C++) [30].

Lua, in particular, is intended to be embedded into C applications and provides developers a mature C API to integrate with. As a result, it has been extensively used with C in many practical applications such as Apache2 (web server), OpenResty3 (application server), and Awesome4 (window manager for X). It is also used within Nmap, an open-source utility for network discovery and security auditing that is prevalently used by security practitioners and adversaries alike. Nmap comes with a scripting engine (Nmap scripting engine or NSE) that is based on Lua, as well as a set of commonly used NSE scripts for supporting common reconnaissance tasks. The scripts are continually updated so that scanning can include the latest vulnerabilities being discovered. Figure 2 shows an overall architecture of Nmap.

2.2 Symbolic Execution

The goal of our work is to symbolically execute NSE scripts in order to determine sets of responses that can trick each script into entering a particular state in its execution. Such a task is traditionally supported via symbolic execution tools [1, 36, 39] which can be applied on target programs to automatically find bugs and vulnerabilities along with the inputs that will trigger them [12]. Instead of running programs with concrete inputs, symbolic execution runs them with symbolic ones that can be "anything" initially [18]. When the inputs of a program are symbolic values, the program can hit any feasible branch during execution and explore all possible execution paths. Symbolic execution explores feasible paths, guiding the program to paths that have not been covered yet. Each path of an execution maintains a set of constraints, called the *path condition*. When a path terminates or hits a desired state, symbolic execution can generate a test case by solving the current path condition for concrete values using a constraint solver. Figure 1 shows the symbolic execution tree and the path conditions of a program. The constraint in the black box on the right is a path condition and symbolic execution will eventually cover all 3 branches in the tree.

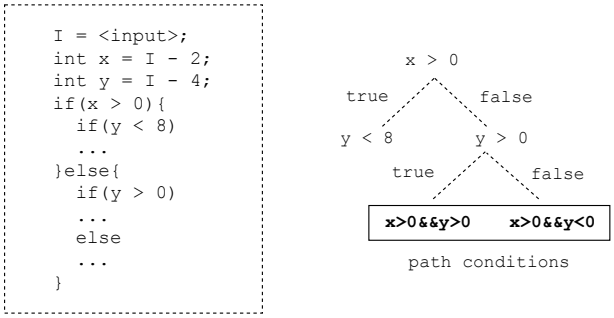

Figure 1: Symbolic Execution Tree

The main scalability challenge for symbolic execution is path explosion. Since each conditional branch can potentially fork the execution, the number of states (and thus paths) grows roughly exponentially in the size of the program [3]. Often, even small programs generate tens or even hundreds of thousands of path conditions during the first few minutes of execution. Path-condition explosion can cause a symbolic executor to fail or miss execution paths it is supposed to cover.

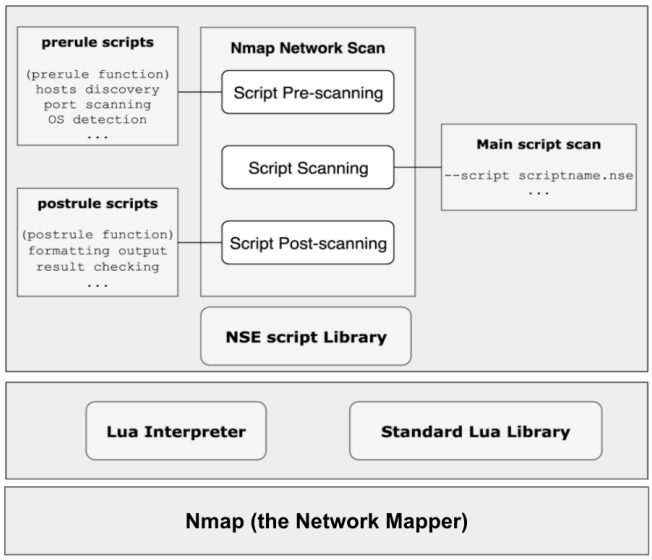

Figure 2: Nmap Architecture

Figure 2 shows the overall architecture of Nmap, its NSE and the library of NSE scripts that are used for network scanning. While NSE scripts themselves are simple, they are written in Lua and interpreted dynamically at run-time. With the complexity of interpreters, it is currently impractical to symbolically execute the Nmap program, the NSE, and even the simplest of NSE scripts as a whole without path explosion.

2.3 Concolic Execution

Several approaches exist to ease the problems caused by path explosion, such as using heuristic path-finding to increase code coverage [23], reducing execution time by parallelizing independent paths [38], or by simply merging similar paths [19]. However in general, one cannot completely avoid the problem, making exhaustive exploration unrealistic for most systems code.

One fundamental idea to cope with these issues and to make symbolic execution feasible in practice is to mix concrete execution and symbolic execution together, also referred to as *concolic execution*, where the term concolic is a combination of the words "concrete" and "symbolic". For example, as Figure 3 shows, classic symbolic execution will explore all 5 paths in the figure. Any feasible path relevant to the input value x is explored, once x is made symbolic with `make_symbolic`, which will lead to path explosion when testing complex programs. Path 4 is a concrete execution path of a target application driven down by a concrete initial value x. By forcing execution to take `br1` concretely before running the target application symbolically, a concolic approach would only execute `br3` and `br4` symbolically while avoiding `br2`, `br5`, and `br6`. Thus, concolic execution can reduce the possibility of path explosion, making it more suitable than symbolic execution for testing complex applications with an embedded interpreter.

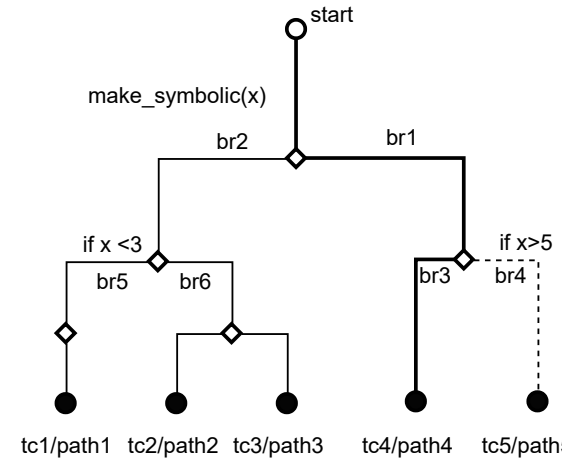

Figure 3: Symbolic execution covers all paths; Concrete execution covers path4; Concolic execution covers path4 and path5

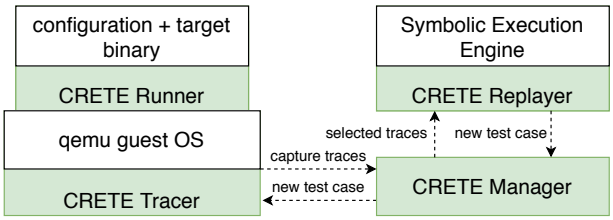

Figure 4: CRETE Architecture

2.4 Concolic Execution with CRETE

To enable concolic execution of NSE scripts, we leverage CRETE a binary-level concolic testing framework. CRETE features an open and highly extensible architecture allowing easy integration of concrete execution front-ends and symbolic execution engine backends [10].

As shown in Figure 4, CRETE uses a configuration file to mark symbolic and concrete inputs in what is referred to as the CRETE runner. As the target program is concretely executed in a modified QEMU virtual machine, the CRETE tracer, a QEMU extension, captures concrete execution traces. These traces are in the form of LLVM bytecode augmented to indicate the execution paths induced by the concrete inputs [21]. If one path contains a symbolic variable marked by the configuration file, CRETE feeds the captured trace of the path to its symbolic execution engine (in this case KLEE [8]), to run it symbolically via CRETE replayer. CRETE extends KLEE to generate test cases only for feasible branches confined by concrete traces so that KLEE will not fork unnecessary states. This results in fewer path conditions. In addition, CRETE uses a selective binary-level trace file other than a full trace file to further reduce path explosion. A full trace contains basic blocks at the assembly level which represents a concrete execution of a target application. It is often unnecessary to trace the complete execution. Because the trace file might be too big for the symbolic engine to consume and there will always be paths that are irrelevant to the symbolic values, which will lead to massive path explosions. Thus, CRETE uses a Dynamic Taint Analysis (DTA) algorithm to implement selective tracing. It only captures the execution traces relevant to the symbolic values marked by DTA. CRETE uses tainted memories to represent the relevant memories that are initially assigned to the symbolic values. For example, if variable "a" is marked as symbolic, when there is an assignment operation involving "a", such as "b=a", the memory slot that b possesses is also marked as symbolic. So CRETE will capture any execution trace involving memories slots of "a" and "b" [34]. CRETE provides a helper interface function "`crete_make_concolic`" to allow users to mark any symbolic variable. We will leverage this interface to implement our approach.

3 OUR APPROACH

3.1 Overview

Our goal is to run NSE scripts using concolic execution to generate test cases to form decoys against the attackers. Because the nature of concolic execution is to explore every possible execution state along a concrete execution trace, this will allow us to focus our symbolic execution on discovering sets of network responses that force an NSE script into transitioning into each of its execution states. This generation is key for honeyfarms as it provides them with responses that can be used to control the NSE scripts' execution behavior. For example, returning an input that leads the script into believing the host is vulnerable would allow the interaction to continue in order to further consume an attacker's time and energy. Selecting a response that leads the script into believing the host is not vulnerable or has no resource of interest would send the attacker away. Randomly selecting from calculated inputs per connection would allow defenders to actively confuse the attacker.

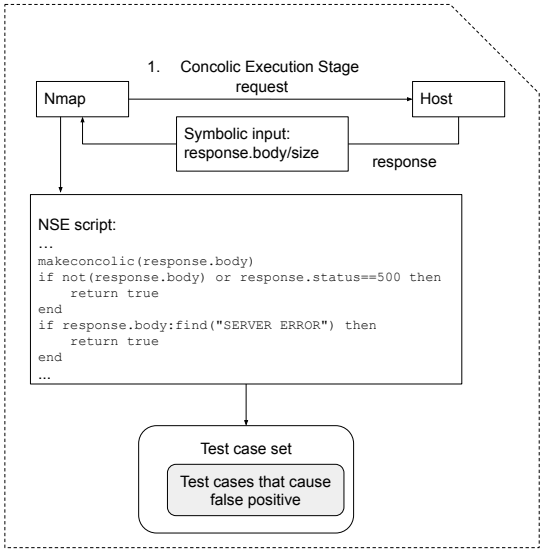

Figure 5: Concolic Execution Stage

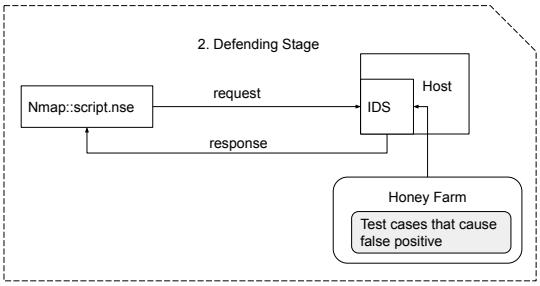

Figure 6: Defending Stage

Finally, returning inputs that may leverage bugs and errors would allow defenders to potentially crash the attacking scripts and terminate the scan altogether. All of these synthesized responses would potentially allow defenders to slow down an attacker's workflow.

In order to complete this process, the approach we take is broken into two stages:

- `Concolic Execution Stage`: As shown in Figure 5, in this stage, we perform concolic execution on NSE scripts to generate test cases for honeyfarm synthesis. For the case shown, we can set `response.body` and `response.size` to symbolic values for the engine to explore.
- `Defending Stage`: As shown in Figure 6, in this stage, we use the various test cases generated from concolic execution to synthesize honeyfarm responses with which we can then have an Intrusion Detection System (IDS) to respond upon detecting the corresponding Nmap scan.

3.2 Concolic Execution Stage

There are several challenges when considering the use of symbolic and concolic execution on an Nmap script. Most existing symbolic and concolic execution engines target low-level code compiled statically. NSE scripts use Lua as the base language and are not statically compiled, but rather interpreted by Nmap's built-in Lua interpreter. The Lua interpreter itself is extended by Nmap with a library for communication, which is responsible for providing additional information that NSE scripts need to execute. For example, `nmap.new_socket()` function supplied by the library returns a new socket wrapper object NSE scripts can use. The Nmap library also takes care of initializing the Lua context, scheduling parallel scripts and collecting the output produced by completed scripts.

Because NSE scripts can utilize both the extended libraries in Nmap and the default libraries of the Lua language, they are more complex than stand-alone Lua scripts. Compounding this complexity is that statements of interpreted languages can encapsulate complex operations that are implemented in underlying compiled libraries written in lower-level languages. For example, the Lua language supports 7 string operations that are implemented in a string library of the Lua interpreter, which contains thousands of lines of C code interpretation [20]. Symbolically executing such code can easily cause path explosion. Consequently, symbolic execution of such scripts may require manual intervention to avoid this problem. Recent work has sought to automate this task, which involves changing the interpreter and building a new symbolic execution engine. Unfortunately, the implementation of a dedicated symbolic execution engine adds a significant amount of work for each language, requiring constant maintenance if the language is updated.

Therefore, we need to apply symbolic execution to analyze arbitrary NSE scripts in a way that avoids the path explosion problem as well as continually updating our execution engine when there are update to the Lua language. To meet this goal, we adapt CRETE, our concolic execution engine, using API calls in the glue layer of the built-in interpreter. Specifically, we use the interface provided by CRETE and modify glue layer in Nmap to allow users to conveniently inject symbolic values from the scripts. Additionally, we modify the engine to provide interfaces that allow users to defer concolic execution of a program as needed in order to further limit the execution paths of the script to the minimum. The reason why we need to defer concolic execution is that we need to keep execution complete for the whole scanning process to guarantee completeness of the trace whiling ensuring that we only symbolically execute the portion of the trace that is of interest. We will show the significant reduction in execution time by deferring concolic execution in Section 5.

Figure 7 shows an example of NSE scripts involved in a Nmap network scan, which has pre-rule scripts, customized scripts and post-rule scripts running in three scan phases respectively (script pre-scan, script scan and script post-scan). In each scan phase, more than one NSE script will be executed. In the script pre-scan phase, pre-rule scripts are executed to collect information for customized scripts which will be executed in the script scan phase. In most cases, users are interested in testing customized scripts because they can be modified, allowing the library to be extended. Testing them with

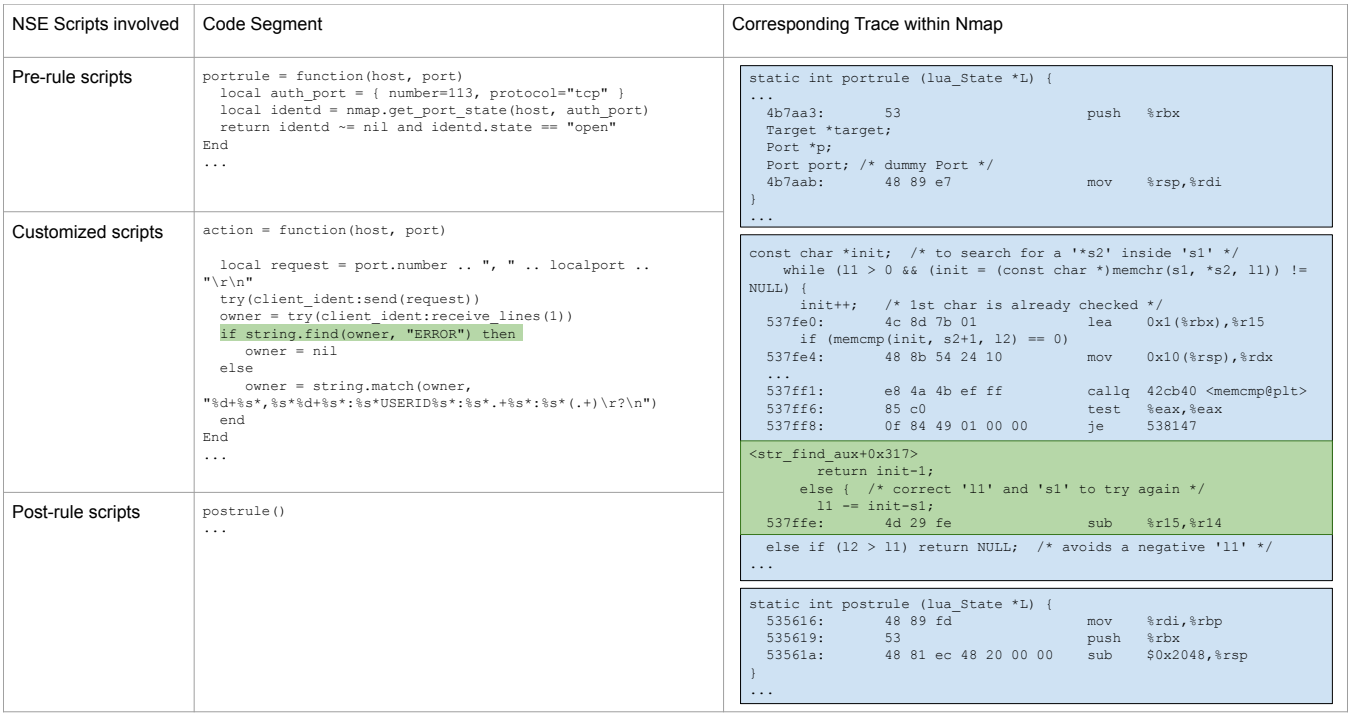

Figure 7: Explanation of Our Approach

concolic execution requires capturing the execution traces for all the NSE scripts that have been executed. The last column in Figure 7 gives an example of one such trace that shows the obstacles facing concolic execution, which is one to many code mapping from scripting language to low level code. The figure shows assembly code snippets for each phase of the scan. As concolic execution works with low-level code representation, path explosion can happen in the script pre-scan phase before the concolic engine can even reach the script scan phase for customized scripts. This situation worsens when the interpreted pre-scan script involves loops or nested pattern matching operations, which is quite common in NSE scripts for string manipulation. Therefore, being able to test the scripts users are actually interested in requires methods to defer symbolic execution to specific segments in order to prevent path explosion. As a result, our approach leverages the adapted interface of CRETE to allow user to customize concolic execution as needed.

In doing so, we make the observation that on such application, an embedded script conceptually executes both on the high level (e.g. at the script language) and the low level (e.g. at the host language). In most cases, applications use C and its interfaces for the host language. Figure 8 illustrates a typical structure for embedding scripts as of Nmap.The base layer includes the host program of the application in C. The top layer consists of embedded scripts prepared by the user. By providing various scripts, the user can customize the application as wishes without recompiling the entire program. The glue layer, which is also written in C, contains the built-in interpreter and glues the gap between C and the scripting language.

We make use of the glue layer to achieve our goal of concolically executing NSE scripts. To gain control of concolic execution, we introduce three important interfaces for symbolic execution: `start_analysis()`, `mark_symbolic()`, and `end_analysis()`. These interfaces will allow us to customize concolic execution in scripts. Modifying the glue layer to include these interfaces allows users to start symbolic execution with a function call. At the same time, starting symbolic execution from the script layer generates a massive execution trace which leads to path explosion. Hence, we have `start_analysis()` and `end_analysis()` to allow us to delay the symbolic execution till later in the execution where we want it and stop it as wish. Therefore, when the target scripts invoke additional scripts of no interest to the analysis, we can easily avoid running unwanted scripts symbolically and only execute the target scripts symbolically by properly calling the above functions. This method has a potential to be applied on other application with the similar structure. In our case (Nmap), embedded scripts and built-in interpreter refer to NSE scripts and Lua interpreter respectively. With these interfaces we can go through the entire execution for pre-rule scripts, customized scripts, and post-rule scripts but only symbolically execute the traces of customized scripts, thus reducing possible symbolic paths significantly.

3.3 Defending Stage

With the method explained above, we can apply concolic execution to any NSE script to get responses that can be leveraged by honeyfarm to control the execution state of the attacking scripts. To achieve our objectives, a range of selection rules targeting different application scenarios can be implemented. Two rules, in particular, include:

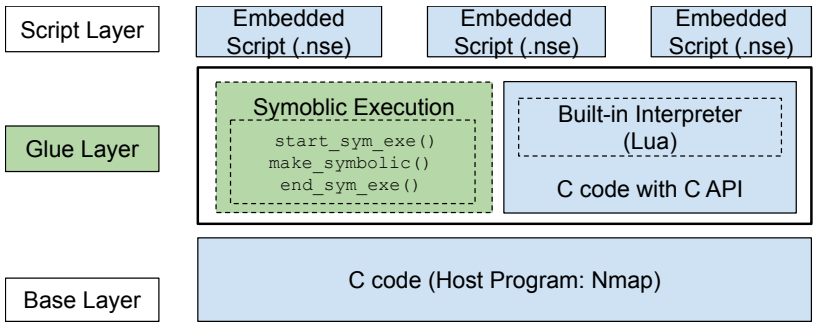

Figure 8: Structure for Applications with Embedded Scripts and Symbolic Execution Interface

- **Early Termination Rule.** With this rule, responses selected will be the ones which will cause the attacking script to stop as soon as possible. We use script coverage as an indicator. We will consider test cases that achieve lower coverage on a script with higher priorities for the synthesis of a honeyfarm.
- **False Positive Rule.** The test cases selected for honeyfarm generation will be the ones which will cause the attacking script to believe that it has find a host with certain vulnerabilities. We will consider test cases that reach certain end-points in a script. These end-points can be annotated manually or identified through templates.

Upon selecting a response, the next step of the defending stage is handling the attacking connection and delivering the response back to the script. Intrusion detection systems (IDS) combined with templating systems provide a natural mechanism for doing so. For example, consider an NSE script seeking to find a vulnerable HTML form submission. An IDS running on a honeyfarm system can provide us hooks into the request being made by the script, while an HTML-templating engine such as Mustache [41], can allow us to use templates that we fill in with the test cases from concolic execution in order to complete the defending stage response.

4 IMPLEMENTATION

4.1 Concolic Script Execution

To support concolic execution of NSE scripts our implementation focuses on the glue layer of Nmap. We use CRETE as the concolic execution back-end engine and modify the glue layer of Nmap to allow users to customize concolic execution for the target application. This includes allowing users to start concolic execution, to introduce symbolic values and to stop concolic execution as needed.

By default, CRETE performs concolic execution on the entire execution trace of a program captured by the CRETE front-end in QEMU. Because we wish to finely control the parts that are symbolically executed, we modify CRETE to decouple concolic execution with a set of interface functions, namely `sendpid()`, `mconcolic()` and `exit()`. These functions pass control of concolic execution from CRETE to NSE scripts. For clarity, the naming convention we used in our implementation of the glue layer for NSE scripts is to keep consistent with the CRETE back-end engine: `sendpid()` is the interface function to start concolic execution if a symbolic variable is present (in corresponding to `start_analysis()`). `mconcolic()` is the interface function to mark symbolic variable (in corresponding to `mark_symbolic()`). `exit()` is to stop concolic execution (in corresponding to `end_analysis()`). As a result, we can defer the concolic execution in Nmap until after the script pre-scan phase and end it before the post-scan phase. We use this control library to minimize symbolic execution on execution traces to address the path explosion problem when concolically executing an interpreted script as shown in Figure 9. The control library allows us to decide which segment of the intermediate code we want CRETE to execute symbolically.

4.2 Lua Interpreter Instrumentation

The embedded Lua Interpreter in Nmap interprets NSE scripts utilizing the string interning optimization. We disabled string interning so that CRETE can use taint analysis to make sure all the relative traces to the symbolic values are captured. Disabling string interning is relatively simple and can be done through a Lua configuration macro [20]. We also handled the Lua's two internal representations for numbers: float and integer. Specifically, we ignored numbers whose internal representation are float, as the underlying symbolic execution engine CRETEuses, namely KLEE, does not support floating point numbers. In addition, we modified Lua math library for all functions to support making internal integer representations symbolic. As an example, Listing 1 shows how we call the interface functions from a NSE script that allows for us to customize concolic execution. The script performs a form submission on a potential vulnerable site and obtains a response. It returns `true` if a null response body is received or if an error is returned. In this example, we choose to inject symbolic values and start symbolic analysis right when the relevant parts of the script are being executed to minimize path explosion.

4.3 Snort response

Once we have performed concolic execution on the script, we then use Snort [31], a network-based IDS to deliver the response. Snort can be configured to detect malicious behaviors over the network with a set of rules in `snort.conf`. We leverage one such set of rules that is maintained, validated, and updated by Proofpoint [29] to allow Snort to detect Nmap scans. Listing 2 shows the rule used to detect Nmap web application attacks in the evaluation. As part of the Snort rule, we configure the rule's `react` option to deliver specific responses that are synthesized using the generated test cases from our concolic execution when the Nmap scan is detected. For the Web

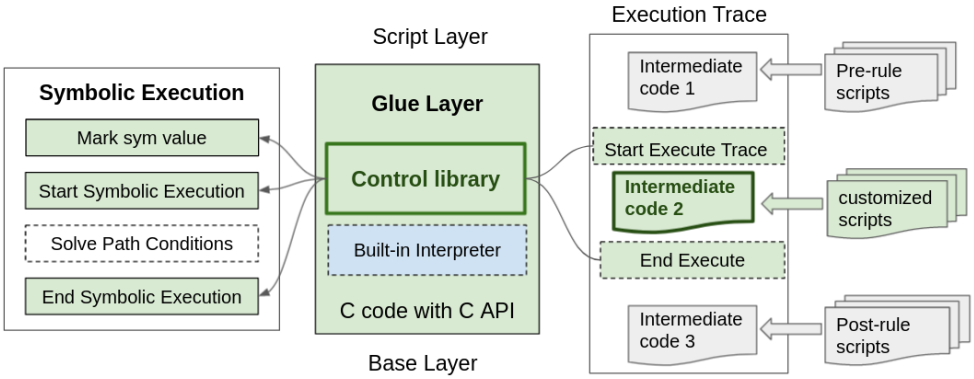

Figure 9: Control Library for Concolic Execution

```
1  local function check_response(response)
2    --crete start
3    crete.sendpid()
4    crete.mconcolic(response.body,12)
5
6    if not(response.body) or response.status==500 then
7      return true
8    end
9    if response.body:find("SERVER ERROR") then
10     return true
11   end
12
13   --exit program
14   crete.mexit(0)
15
16   return false
17 end
```

Listing 1: http-form-fuzzer.nse instrumented with CRETE.

Application Scan from Listing 1, an example of the synthesized response is shown in Listing 3. The string "SERVER ERROR" in line 3 has been generated by CRETE. Note that for this case, while the string appears in the page's title, one can place the string anywhere in the `response.body` to trick this particular script. The generation of the response HTML can be done using any automated templating system such as Mustache [41] that allows us to replace parts of the content with the test case generated from concolic execution.

Listing 2: Snort detecting rule for `Nmap` web application scan

```
alert tcp any any -> any any (msg:"ET SCAN Nmap Scripting
Engine User-Agent Detected (Nmap Scripting Engine)";
flow:to_server,established;
content:"User-Agent|3a| Mozilla/5.0 (compatible|3b| Nmap
Scripting Engine";
react; fast_pattern:38,20; http_header;
nocase; reference:url,doc.emergingthreats.net/2009358;
classtype:web-application-attack; sid:2009358; rev:5;)
```

```
1  <!doctype html>
2  <html lang="en">
3    <head><title>SERVER ERROR</title></head>
4    <body>
5    <div style="color:red">
6    </div>
7     <form name="LoginForm" method="post"
8     action="/loginclass/Login.do;jsessionid=
        D34B538055462B75E1CD6DFD18B9650E">
9      User Name:<input type="text" name="userName" value="">
10     Password:<input type="password" name="password" value="">
11     <input type="submit" value="Login">
12    </form>
13   </body>
14  </html>
```

Listing 3: An Example of Synthesized Response in Snort

5 PRELIMINARY EVALUATION

In this section, we first introduce the NSE scripts we target and the experimental setup for our approach including the CRETE settings which are used in the concolic execution stage. Then, we will summarize our preliminary results, which shows the type of test cases from running NSE scripts with our approach with a set of examples. Finally, we analyze why we are able to achieve these results.

5.1 Experimental Setup for NSE Scripts

Because a large majority of network protocols such as HTTP are string-based, string manipulation operations are some of the most frequently used in NSE scripts. As a result, our experiments mainly focus on string variables when injecting symbolic values into NSE scripts. We follow the simple heuristics below to select which variables are made symbolic:

- For host scan scripts, variables that are involved in `if-else` branches in scripts are set as symbolic values. Among string operations, substring finds and string pattern matching commonly appear in branch statements since such functions return values that are of `boolean` type.
- For web scripts, `response.body` and `response.size` are set as symbolic since they are commonly involved in branches as information they return is often of interest to NSE scripts.

To showcase our approach, we use `http-form-fuzzer.nse` as an example, which involves the `string.find` function. With the above heuristics, we set `response.body` and `response.size` as symbolic variables for the case where response is an HTML page.

5.2 Control Interface Evaluation

5.2.1 Naïve Case. Our early attempt of applying concolic execution on NSE scripts is to run the NSE script concolically using CRETE without deferring concolic execution until when it is needed. The experiment setup for this case is that we simply use the interface of `crete.mconcolic()` to mark symbolic variables then run the NSE script directly. As expected, doing so causes the pre-scan stage to

be involved in the concolic execution process, leading to excessive execution time. Across four executions of the script done in this manner, execution time averages 4519 seconds to explore each new feasible path in the script.

5.2.2 Customized Concolic Execution Case. The advantage of our approach is the support for a control interface that allows the NSE script to defer concolic execution. For example, the segment of code in Listing 1 is from `http-form-fuzzer.nse`. It is frequently used to fuzz the fields of web page that contain `<form>` tags to try to find a certain request that will cause an ERROR in the web page [26]. Listing 1 shows an example of how we use the control interface to efficiently enable concolic execution when needed. In the listing, we wish to test line 6 to line 11, which contains two "`if`" statements and the symbolic value we wish to evaluate, `response.body`, whose type is a string. We then call function `crete.sendpid` to start symbolic analysis before we mark symbolic value with `crete.mconcolic` function. In this way, we have CRETE defer symbolic execution of the code until after we inject the symbolic value, thus avoiding the symbolic execution of pre-run scripts. Finally, we terminate symbolic execution using `crete.mexit` so that the symbolic execution only targets line 6 to 11 and avoids running post-run scripts symbolically.

When testing `http-form-fuzzer.nse`, with otherwise the same experimental setup as the naïve case, execution time is reduced from 4519 seconds on average to around 179 seconds per new feasible path in the script. This indicates the effectiveness of customized concolic execution. For the rest of our experiments, we apply this method for deferring concolic execution when testing NSE scripts.

5.3 NSE Script Evaluation

5.3.1 Test case generation for honeyfarms. Our goal is to concolically execute a variety of NSE scripts in order to produce inputs that can be used to drive them to particular states. To demonstrate this, we initially select a collection of NSE scripts for HTTP shown in Table 1. For the script we have been using as an example, `http-form-fuzzer.nse`, concolic execution yields the test case with the content of "SERVER ERROR" that leads execution to go into the `if` branch in Listing 1, demonstrating that our approach can produce results at the script level despite the massive amount of interpreted code being executed. We use this test case in a Snort `react` defense rule and succeed in fooling `Nmap` into thinking it identified a vulnerability, accomplishing the *False Positive* goal for the honeyfarm.

A more interesting case is the `http-form-brute.nse` script, in which a `string.match` call tries to validate whether a certain value exists in a user information form returned by a scan. Furthermore, the script checks that the value `v` parsed from the form via `string.ma -tch(form[k], v)` has a pattern '`%d%d`'. To match this, concolic execution generates the test case with two digits in random combinations. Our concolic execution approach also uncovers invalid patterns that lead execution into an error state. For example, the value of '`username/(`' crashes the script since magic characters such as '`(`' need to be escaped in Lua in order to be taken literally or they must appear in pairs such as '`()`'. Such a crashing pattern can be used to trigger the *Early Termination* rule for the honeyfarm.

Listing 4: A code segment of http-title.nse

```
1    if display_title and display_title ~= "" then
2      display_title
3      = string.gsub(display_title , "[\n\r\t]", "")
4      if #display_title > 65 then
5        display_title
6        = string.sub(display_title, 1, 62) .. "..."
7      end
8    else ...
```

NSE scripts	Test Cases/Bugs	Defending Rules
http-form-fuzzer	SERVER ERROR	False positive
http-form-brute	Invalid patterns	Early termination
http-auth	Embedded zero	Early termination
http-grep	Type inconsistency	Early termination

Table 1: Examples of interesting test cases and bugs discovered

For `http-auth.nse`, we have test cases that have '`\0`' in the middle of the `name` variable, e.g., '`name = do\0in`'. This causes an error since '`\0`' is not considered as a terminator for a string in NSE with the Lua interpreter instead treating the character as an *embedded zero* instead. Therefore, in NSE the length of variable `name` is 6 but in C it is 2. This leads to an inconsistency in length which forces execution into the Lua error state, triggering another *Early Termination* situation.

Finally, our concolic approach exposes another similar bug in `http-grep.nse` by generating input that triggers a type inconsistency bug in the script shown in Listing 6. Detailed explanation is given later in the Anlysis section. Our patch for this crashing bug has been accepted by the `Nmap` team [1].

We use the generated test cases discussed above to form honeyfarm responses that fool the scripts. These test cases are expected to trick the scripts or stop them from running. We synthesized these test cases with templates and deliver them back to `Nmap` using Snort configured with appropriate `react` rules. The test cases successfully cause `Nmap` to reach the desired states, accomplishing the goals from Section 3: namely *Early Termination* and *False Positive* as summarized in Table 1.

5.3.2 Analysis. We use a few scripts as an example to show how we generate such test cases. When testing `http-form-fuzzer.nse`, we have the desired test case with the content of "SERVER ERROR" that leads the execution to go into the `if` branch. We get to this particular test case at the 80th iteration, and we obtain the test cases that cover both `if` and `else` branches. We disassemble the relevant part of the `Nmap` binary and show it in Listing 5. For this case, our approach only captures the basic block that has the branch ("`537ff6`") in shown in line 12, which matches the branch of `string.find` in the NSE script in Listing 1. Only this part of the execution trace is executed symbolically instead of the entire trace, thus allowing us to generate the desired test cases efficiently.

For testing of the `http-grep.nse` script shown in Listing 6, our approach enabled us to discover a bug in a local function within the script that implements `Luhn`, an algorithm that is used to validate a variety of identification numbers, such as credit card numbers.

[1] https://github.com/nmap/nmap/issues/1931

To understand how the bug works, we first describe the Luhn algorithm [15] in the following 4 steps:

(1) Starting from the rightmost digit, double the value of every second digit.
(2) If doubling of a number results in a two digit number, then add the digits of the product to get a single digit number.
(3) Take the sum of all the digits.
(4) If the total modulo 10 is equal to 0 (if the total ends in zero) then the number is valid according to the Luhn formula; otherwise it is not valid.

The two loops (in lines 5-7 and in lines 9-15) show the implementation of steps 2 and 3 in the NSE script and contain a bug. The bug is triggered by a test case which causes the value of variable `double` inside of `string.gsub` to be 14. When this happens, the returning value of the `string.gsub` call in line 12 becomes `5.0.0`, which cannot be coerced to a string by the code in line 13. Thus, our concolic approach allows us to easily reveal crashing bugs in NSE scripts that could be used to trigger the termination of the scan. In this case, however, the bug was reported and the Nmap developers changed its implementation to fix the issue.

Listing 7 shows the captured execution trace that corresponds to the loop of the reverse function in C code that triggers the issue. This trace guides concolic execution to mutate input string backwards (from the last position instead from the first position). In addition, it has the information about the two `for` loops, which increment `i` by 2 every iteration. This means only mutating the bytes in odd positions of the input string after being reversed can trigger the bug in line 12 due to the step of 2 in each iteration. With this knowledge, our approach can make changes on the proper position of the string, which is every other character in the string after being reversed. And the effective change is to flip the bits of the character to an ASCII code that can be converted to a number so it can pass line 11 to get to line 12 where the bug resides. The bug is triggered if the number (`doubled`) in an odd position is greater than 9. Our approach was able to make the right mutation after a few iterations to trigger the bug in line 12.

```
const char *init;  /* to search for a '*s2' inside 's1' */
    l2--;  /* 1st char will be checked by 'memchr' */
    l1 = l1-l2;  /* 's2' cannot be found after that */
    while (l1 > 0  (init = (const char *)memchr(s1, *s2, l1))
      != NULL) {
      init++;   /* 1st char is already checked */
  537fe0:       4c 8d 7b 01             lea    0x1(%rbx),%r15
      if (memcmp(init, s2+1, l2) == 0)
  537fe4:       48 8b 54 24 10          mov    0x10(%rsp),%rdx
  537fe9:       48 8b 74 24 18          mov    0x18(%rsp),%rsi
  537fee:       4c 89 ff                mov    %r15,%rdi
  537ff1:       e8 4a 4b ef ff          callq  42cb40 <
      memcmp@plt>
  537ff6:       85 c0                   test   %eax,%eax
  537ff8:       0f 84 49 01 00 00       je     538147 <
      str_find_aux+0x317>
        return init-1;
      else {  /* correct 'l1' and 's1' to try again */
        l1 -= init-s1;
  537ffe:       4d 29 fe                sub    %r15,%r14
  else if (l2 > l1) return NULL;  /* avoids a negative 'l1' */
```

Listing 5: Captured trace from http-form-fuzzer.nse script

```
1  function luhn (matched_ccno)
2    crete.mconcolic(matched_ccno, matched_ccno.len)
3    local n = string.reverse(matched_ccno)
4    local s1 = 0
5    for i=1, n:len(), 2 do
6      s1 = s1 + tonumber(n:sub(i,i))
7    end
8    local s2 = 0
9    for i=2, n:len(), 2 do
10     --conversion from string to double
11     local doubled = n:sub(i,i)*2
12     doubled = string.gsub(doubled,'(%d)(%d)',
13     function(a,b)return a+b end)
14     s2 = s2+doubled
15   end
16 end
```

Listing 6: A code segment of http-grep.nse script with string.reverse function: a type inconsistency bug is triggered in line 13 when trying to sum `doubled` with `s2`. This function (*luhn*) is used to validate credit card numbers

```
static int str_reverse (lua_State *L) {
  535616:       48 89 fd                mov    %rdi,%rbp
  535619:       53                      push   %rbx
  53561a:       48 81 ec 48 20 00 00    sub    $0x2048,%rsp
  size_t l, i;
  luaL_Buffer b;
  const char *s = luaL_checklstring(L, 1, 1);
  535621:       48 8d 54 24 08          lea    %rsp,%rdx
  else lua_pushliteral(L, "");
  return 1;
}
```

Listing 7: The captured trace when testing http-grep.nse script. This trace segment contributes to finding the type inconsistency bug

6 RELATED WORK

Most existing symbolic and concolic execution engines target low-level code representations. For example, symbolic execution engines such as KLEE [8], BitBlaze [37] and S2E [11] as well as concolic execution engines such as DART [13], CUTE [35] and SAGE [14] work with either machine code or LLVM intermediate representation code [21] that has been statically compiled. The NSEscripts that we are dealing with, are however, interpreted, not statically compiled. There has also been efforts in building symbolic engines targeting script languages. However, such implementation requires significant amount of work for every single language and constant maintenance if the target language is updated. NICE [9] for Python and Kudzu [33] for Javascript are early efforts to directly implement symbolic execution engines for dynamically interpreted scripts in high-level languages. Existing symbolic execution engines that can support Lua only target standalone interpreters such as CHEF [7] while NSE scripts are interpreted by an interpreter embedded in Nmap.

There are two ways for deploying honeyfarms: low-interaction honeyfarm and high-interaction honeyfarm. Low-interaction honeyfarm can monitor activities over millions of IP addresses at a time, such as KFSensor Honeypot [25] and Conpot [24]. This kind of scalability is achieved by emulating the network interface exposed by common services and requires low maintenance. However, such systems do not execute any code from applications; therefore, they may not be able to block attacks that have multiple phases of communication [40]. On the other hand, high-interaction honeyfarms run native application code, and therefore, is able to catch code behavior in its full complexity [28]. As a consequence, the

implementation cost is quite high. Systems of high interaction honeyfarms include Honeynets [28], Sebek [16], Argos [27], etc. Our method is a light way of achieving the purpose of high-interaction honeyfarms.

7 CONCLUSIONS

This paper presented an approach to test NSE scripts via concolic execution and to use the result to generate honeyfarms that can slow down attackers. Preliminary results have shown its efficiency in generating test cases that can stop Nmap scans or return false positive responses. Our approach is effective with complicated programs such as Nmap which runs embedded scripts where traditional concolic execution does not work at all. Our approach does so by avoiding path explosion by supporting customized concolic execution at specific locations in order to generate useful test cases efficiently. The implementation for our approach makes use of the glue layer that most embedded scripting languages provide to integrate the concolic execution engine and the interface functions for customizing concolic execution. In this way, the approach does not need to modify the built-in interpreter each time the language is updated. In the future, we aim to test more libraries in NSE since the effective concolic execution of more NSE scripts is the key to building diverse honeyfarms.

8 ACKNOWLEDGEMENT

This research received financial support in part from National Science Foundation (Grant #: 1908571).

REFERENCES

[1] Angr Developers. 2016. Angr, a Binary Analysis Framework. http://angr.io/.
[2] Ars Technica. 2017. NSA-leaking Shadow Brokers Just Dumped Its Most Damaging Release Yet. https://arstechnica.com/information-technology/2017/04/nsa-leaking-shadow-brokers-just-dumped-its-most-damaging-release-yet/.
[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (May 2018), 39 pages. https://doi.org/10.1145/3182657
[4] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. 2011. Cython: The Best of Both Worlds. *Computing in Science Engineering* 13, 2 (2011), 31–39. https://doi.org/10.1109/MCSE.2010.118
[5] Tegawendé F. Bissyandé, Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Réveillère. 2013. Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects. In *2013 IEEE 37th Annual Computer Software and Applications Conference*. 303–312. https://doi.org/10.1109/COMPSAC.2013.55
[6] Bloomberg Technology. 2017. Equifax Suffered a Hack Almost Five Months Earlier Than the Date It Disclosed. https://www.bloomberg.com/news/articles/2017-09-18/equifax-is-said-to-suffer-a-hack-earlier-than-the-date-disclosed.
[7] Stefan Bucur, Johannes Kinder, and George Candea. 2014. Prototyping symbolic execution engines for interpreted languages. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. 239–254.
[8] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.
[9] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. 2012. A NICE Way to Test OpenFlow Applications. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 127–140. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/canini
[10] Bo Chen, Christopher Havlicek, Zhenkun Yang, Kai Cong, Raghudeep Kannavara, and Fei Xie. 2018. CRETE: A Versatile Binary-Level Concolic Testing Framework. In *Fundamental Approaches to Software Engineering*, Alessandra Russo and Andy Schürr (Eds.). Springer International Publishing, Cham, 281–298.
[11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices* 46, 3 (2011), 265–278.
[12] DARPAtv. 2017. DARPA's Cyber Grand Challenge: Final Event Program. https://www.youtube.com/watch?v=n0kn4mDXY6I.
[13] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 213–223.
[14] Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (2012), 40–44.
[15] Hans Peter Luhn. 2021. Luhn algorithm. https://en.wikipedia.org/wiki/Luhn_algorithm.
[16] Pei-Sheng Huang, Chung-Huang Yang, and Tae-Nam Ahn. 2009. Design and implementation of a distributed early warning system combined with intrusion detection system and honeypot. In *Proceedings of the 2009 International Conference on Hybrid Information Technology*. 232–238.
[17] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes. 2011. Passing a Language through the Eye of a Needle. *Commun. ACM* 54, 7 (July 2011), 38–43. https://doi.org/10.1145/1965724.1965739
[18] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252
[19] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient State Merging in Symbolic Execution. *SIGPLAN Not.* 47, 6 (June 2012), 193–204. https://doi.org/10.1145/2345156.2254088
[20] LabLua. 2021. Lua Reference Manuals. https://www.lua.org/manual/.
[21] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665
[22] Gordon Lyon. 2021. Nmap: the Network Mapper. https://nmap.org/.
[23] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. 2011. Directed Symbolic Execution. In *Static Analysis*, Eran Yahav (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 95–111.
[24] Microsoft. 2021. CONPOT ICS/SCADA Honeypot. http://conpot.org/.
[25] Microsoft. 2021. KFsensor: Advanced Windows Honeypot System. http://www.keyfocus.net/kfsensor/.
[26] Piotr Olma and Gioacchino Mazzurco. 2021. NSE Script description. https://nmap.org/nsedoc/scripts/http-form-fuzzer.html.
[27] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. 2006. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *ACM SIGOPS Operating Systems Review* 40, 4 (2006), 15–27.
[28] Honeynet Project. 2001. *Know Your Enemy: Revealing the Security Tools, Tactics, and Motives of the Blackhat Community*. Addison-Wesley Professional.
[29] Proofpoint. 2021. Proofpoint Emerging Threats Rules. https://rules.emergingthreats.net/.
[30] Rasterbar Software. 2005. Luabind. http://www.rasterbar.com/products/luabind.html.
[31] Martin Roesch et al. 1999. Snort: Lightweight intrusion detection for networks.. In *LISA*. 229–238.
[32] K Sasada. 2009. Ricsin: A System for 'Mix-in to Ruby'. *IPSJ Transactions on Programming* 2, 2 (March 2009), 13–26.
[33] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A Symbolic Execution Framework for JavaScript. In *2010 IEEE Symposium on Security and Privacy*. 513–528.
[34] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. 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*. 317–331. https://doi.org/10.1109/SP.2010.26
[35] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 263–272.
[36] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. 138–157. https://doi.org/10.1109/SP.2016.17
[37] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*. Springer, 1–25.
[38] Matt Staats and Corina Pundefinedsundefinedreanu. 2010. Parallel Symbolic Execution for Structural Test Generation. In *Proceedings of the 19th International Symposium on Software Testing and Analysis* (Trento, Italy) *(ISSTA '10)*. Association for Computing Machinery, New York, NY, USA, 183–194. https://doi.org/10.1145/1831708.1831732
[39] Trail of Bits. 2017. Manticore: Dynamic Binary Analysis Tool. http://github.com/trailofbits/manticore.
[40] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. Snoeren, G. Voelker, and S. Savage. 2005. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *ACM SOSP*. 148–162.
[41] Chris Wanstrath. 2009. Mustache Processor. https://mustache.github.io/.