2017 IEEE Symposium on Security and Privacy

Your Exploit is Mine: Automatic Shellcode Transplant for Remote Exploits

Tiffany Bao
Carnegie Mellon University
tiffanybao@cmu.edu

Ruoyu Wang
UC Santa Barbara
fish@cs.ucsb.edu

Yan Shoshitaishvili
UC Santa Barbara
yans@cs.ucsb.edu

David Brumley
Carnegie Mellon University
dbrumley@cmu.edu

***Abstract*—Developing a remote exploit is not easy. It requires a comprehensive understanding of a vulnerability and delicate techniques to bypass defense mechanisms. As a result, attackers may prefer to reuse an existing exploit and make necessary changes over developing a new exploit from scratch. One such adaptation is the replacement of the original shellcode (i.e., the attacker-injected code that is executed as the final step of the exploit) in the original exploit with a replacement shellcode, resulting in a modified exploit that carries out the actions desired by the attacker as opposed to the original exploit author. We call this a shellcode transplant.**

Current automated shellcode placement methods are insufficient because they over-constrain the replacement shellcode, and so cannot be used to achieve shellcode transplant. For example, these systems consider the shellcode as an integrated memory chunk, and require that the execution path of the modified exploit must be same as the original one. To resolve these issues, we present ShellSwap, a system that uses symbolic tracing, with a combination of *shellcode layout remediation* and *path kneading* to achieve shellcode transplant. We evaluated the ShellSwap system on a combination of 20 exploits and 5 pieces of shellcode that are independently developed and different from the original exploit. Among the 100 test cases, our system successfully generated 88% of the exploits.

I. Introduction

Remote exploits are extremely dangerous. With the help of remote exploits against a piece of software running on a victim computer, an attacker can install backdoors and exfiltrate sensitive information without physical access to the compromised system, leading to real-world impacts on the finances and reputation of the victim.

However, developing a remote exploit is not easy. A comprehensive understanding of the vulnerability is a must, and complex techniques to bypass defenses on the remote system are necessary. When possible, rather than developing a new exploit from scratch, attackers prefer to *reuse* existing exploits in their attacks, making necessary changes to adapt these exploits to new environments. One such adaptation is the replacement of the original *shellcode* (i.e., the attacker-injected code that is executed as the final step of the exploit) in the original exploit with a replacement shellcode, resulting in a modified exploit that carries out the actions desired by the attacker as opposed to the original exploit author. We call this a *shellcode transplant*. Shellcode transplanting has many applications, including reversing command and control protocols, understanding captured exploits, and replaying attacks. Thus, this capability is very helpful in situations ranging from rapid cyber-response (i.e., quick analysis of and response to 0-day attacks) and adversarial scenarios (like cyber-security Capture-The-Flag competitions or cyber warfare in the real world). Unfortunately, current techniques to transplant shellcode generally require an analyst to have a decent understanding of how the original exploit interacts with the program, what vulnerability it triggers, and how it bypasses deployed exploit mitigations. As a result, the analyst must put a lot of effort into development and debugging, which negates much of the advantage of shellcode transplanting.

In investigating this problem, we identified three main challenges to tackling the *shellcode transplant problem*. First, it is very difficult to separate the shellcode from the rest of an exploit, as there is generally no clear boundary separating one from the other. Second, as an exploit's shellcode is commonly constructed through non-trivial data transformations, even if the bytes representing the original shellcode could be separated from the exploit, rewriting these bytes to a replacement shellcode would be non-trivial. Third, the shellcode and the remainder of the content in an exploit can be mutually dependent on each other (e.g., a field in the exploit payload may dictate the size of the embedded shellcode). Such relations can pose potentially complex constraints on any replacement shellcode that might be transplanted. When those constraints are violated by replacement shellcode, it is challenging to modify the exploit and/or the replacement shellcode in order for the modified exploit to function properly.

Previous work in the field of automated exploit generation generates exploits by constraining the memory bytes in each attacker-controlled buffer to the target shellcode. They enumerate all possible offsets in every attacker-controlled buffer until a solution is found [12, 17]. Such methods are insufficient. In the worst case, when attempting to compensate for the case of conflicting constraints on the replacement shellcode, these methods degenerate to a symbolic exploration of the program, which generally ends in a path explosion problem or is hampered by the inability of the symbolic execution engine to efficiently reverse complex data transformations. In fact, as we show in our evaluation, less than a third of the original exploits in our dataset can be modified by existing techniques.

In this paper, we present *ShellSwap*, an automated system that addresses the *shellcode transplant problem*. ShellSwap takes an existing exploit and a user-specified replacement shellcode as input and produces a modified exploit that

DOI 10.1109/SP.2017.67

targets the same vulnerability as the original exploit does but executes the replacement shellcode after exploitation. ShellSwap tackles the challenges discussed above with a mix of symbolic execution and static analysis techniques, applying novel techniques to identify the original shellcode, recover the data transformation performed on it, and resolve any conflicts introduced by the transplant of the replacement shellcode. By utilizing information obtained from the original exploit and creatively transforming the replacement shellcode, ShellSwap rarely degrades to a pure symbolic exploration, and is thus more efficient and effective compared to previous solutions.Additionally, the use of carefully-designed systematic approaches enables ShellSwap to transplant more shellcode variants. In our experiment, ShellSwap successfully generates new exploits for 88% of all cases, which is almost three times the success rate of prior techniques.

To the best of our knowledge, ShellSwap is the first automated system that modifies exploits based on shellcode provided by analysts. In terms of offense, ShellSwap greatly reduces the overhead in attack reflection, which enables prompt responses to security incidents like 0-day attacks, especially in a time-constrained, competitive scenario such as a hacking competition or cyber warfare. ShellSwap also makes it possible for entities to *stockpile* exploits in bulk, and tailor them to specific mission parameters before they are deployed at a later time. As organizations such as the National Security Agency are commonly known to be stockpiling caches of vulnerabilities, such a capability can greatly reduce the overhead in using weapons from this cache. ShellSwap is also helpful in defense, where it can be used to debug exploits discovered in the wild (i.e. by transplanting a piece of shellcode that is benign or implements monitoring and reporting functionality) and rediscover vulnerabilities being exploited.

Specifically, our paper makes the following contributions:

- We design the ShellSwap system, which is the first end-to-end system that can modify an observed exploit and replace the original shellcode in it with an arbitrary replacement shellcode. Our system shows that the automatic exploit reuse is possible: even a person who has little understandings about security vulnerabilities can retrofit an exploit for their custom use-case.
- We propose novel, systematic approaches to utilize information from the original exploit to prevent ShellSwap from degenerating to inefficient symbolic exploration, and revise the replacement shellcode without changing its semantics to fit constraints implicit to the original exploit. Those approaches are essential to the performance of ShellSwap.
- We evaluate our system on 100 cases — 20 original exploits, each with 5 different pieces of shellcode. Our system successfully generates modified exploits in 88% of our test set, and all new exploits work as expected. We also compare our system with the previous state of the art, and we find that previous methods only work for 31% of our test set. The fact that ShellSwap exhibits a success rate almost triple that of the previous solution implies that the impact of the challenges inherent in shellcode transplant were under-estimated, and that future work targeting this problem will be beneficial.

II. Overview

ShellSwap takes, as an input, a *vulnerable program*, the *original exploit* that had been observed being launched against this program, and a *replacement shellcode* that the *original shellcode* in the original exploit should be replaced with. Given these inputs, it uses a combination of symbolic execution and static analysis to produce a *modified exploit* that, when launched against the vulnerable program, causes the replacement shellcode to be executed.

Our intuition for solving the shellcode transplant problem comes from the observation that a successful control flow hijacking exploit consists of two phases: before the hijack, where the program state is carefully set up to enable the hijack, and after the hijack, when injected shellcode carries out attacker-specified actions. We call the program state after the first phase the *exploitable state*, and we call the instruction sequence that the program executes until the exploitable state the *exploit path*. An input that makes the program execute the same path as the original exploit does will lead the program to an exploitable state. Therefore, if we find an input that executes the instructions of the original exploit path in the first phase and the new shellcode in the second phase, that input represents the modified exploit.

Given these inputs, it proceeds through a number of steps, as diagrammed in Figure 1. The steps for generating the new exploit are as follows:

Symbolic Tracing. The path generator replays the exploit in an isolated environment and records the executed instructions. The output of the path generator is a sequence of instruction addresses, which we call the *dynamic exploit path*.

The path generator passes the dynamic exploit path to the symbolic tracing engine. Then tracer sets the input from the exploit as a symbolic value and starts symbolically executing the program. At every step of this execution, the tracer checks if the current program state violates a security policy. There are two reasons for this: a) we want to double check that the exploit succeeds, and b) we need to get the end of the normal execution and the start of malicious computation, where the exploit diverts the control flow of the program to the shellcode. When the tracer detects that the security policy has been violated, it considers the trace complete and the exploitable state reached.

The tracing engine records the path constraints introduced by the program on the exploit input in order to reach the exploitable state, and the memory contents of the exploitable state itself. These will be used in the next step to surmount challenges associated with shellcode transplanting.

Shellcode Transplant. Shellcode transplant is the critical step in the ShellSwap system. It takes the exploitable state, the

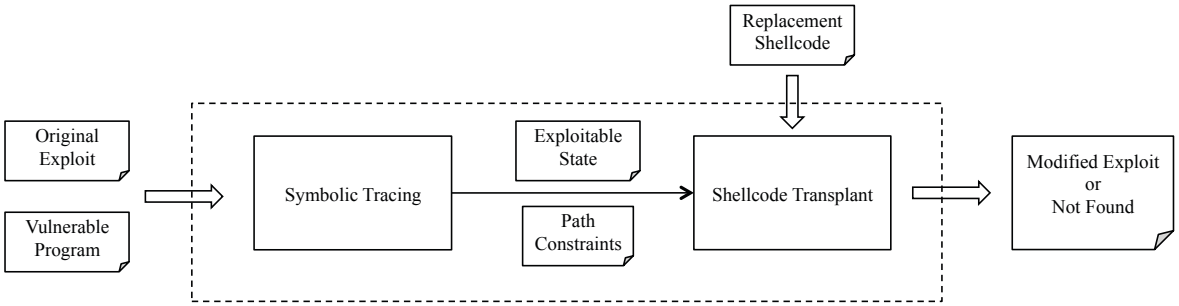


Fig. 1: The architecture of the ShellSwap system.

path constraints, and the replacement shellcode as input, and outputs a modified exploit that takes advantage of the vulnerability and executes the replacement shellcode. After this step, the system will output either the modified exploit or an error indicating that a modified exploit could not be found.

These steps are further described in Section III (for Symbolic Tracing) and Section IV (for Shellcode Transplant).

ShellSwap focuses on exploits against control-flow hijacking vulnerabilities, which are a type of software bug that allows an attacker to alter a program's control flow and execute arbitrary code (specifically, the shellcode). Control-flow hijacking vulnerabilities have been considered as the most serious vulnerabilities, since the attacker can take control of the vulnerable system. Unfortunately, control-flow hijacking vulnerabilities are the most prevalent class of vulnerabilities in the real world: over the past 18 years, 30.6% of reports in the Common Vulnerabilities and Exposures database represent control-flow hijacking vulnerabilities [15]. Thus, while the ability to reason about other types of exploits would be interesting, we leave the exploration of this to future work.

A. Motivating Example

To better communicate the concept of shellcode transplant and demonstrate the challenges inherent to it, we provide a motivating example. We first introduce a vulnerable program and an original exploit, and then discuss the challenges posed by two different instances of replacement shellcode.

1) Vulnerable program: Consider a vulnerable program with source code shown in Listing 1, where the program receives a string terminated by a newline, checks the first character and calculates the length of the string. Note that the source code is for clarity and simplicity; our system runs on binary program and does not require source code.

```
1   int example(){
2     int len = 0;
3     char string[20];
4     int i;
5     if (receive_delim(0, string, 50, '\n') != 0)
6       return -1;
7     if(string[0] == '^')
8         _terminate(0);
9     for(i = 0; string[i] != '\0'; i++)
10      len++;
11    return len;
12  }
```

Listing 1: Motivating Example.

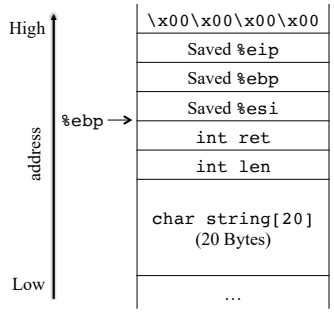


Fig. 2: The stack layout of the `example` function.

This program has a control-flow hijacking vulnerability in the processing of the received input. The `string` variable is a 20-byte buffer defined at line 3. However, the string received from user input can have up to 50 characters, which will overflow the buffer `string` and eventually overwrite the return address stored on the stack if the provided string is long enough. Figure 2 shows the stack layout of the `example` function. The saved return address (shown as saved `%eip`) is 36 bytes above the beginning of buffer `string`. This implies that if the received input has more than 36 characters, the input will overwrite the saved return address and change the control flow of the program when function `example` returns.

```
1  shellcode = "\x31\xc0\x40\x40\x89\x45\xdc"
2  exploit = shellcode + "\x90" * (36 - len(
     shellcode)) + "\x50\xaf\xaa\xba\n"
```

Listing 2: The original exploit with `shellcode`.

2) Original exploit: Listing 2 shows the original exploit for the running example. The shellcode starts at the beginning of the exploit, followed by padding and the address with which to overwrite the return address. When the vulnerable program executes with the original exploit, the return address for function `example` will be changed to `0xbaaaaf50`, which points to the beginning of buffer `string`, and when function `example` returns, the control flow will be redirected to the shellcode.

B. Challenges

To demonstrate the challenges inherent in the shellcode transplant problem, we first consider a naive approach: if we find the location of the old shellcode in the original exploit, we could generate a new exploit by replacing, byte by byte, the old shellcode with the new one. We call this the *shellcode byte-replacement approach*. However, this naive approach assumes two things, that the shellcode stays in its original form throughout execution and that the replacement shellcode is the same size as the original shellcode. As we discussed previously, both of these assumptions are too strict for real-world use cases.

For example, consider the following replacement shellcode for the original exploit in our motivating example:

```
1  xor    %esi,%esi              31 f6
2  lea    0x1(%esi),%ebx         8d 5e 01
3  lea    0x8(%esi),%edx         8d 56 08
4  push   0xaaaaaaaa             ff 35 aa aa aa aa
5  push   $0xdddddddd            68 dd dd dd dd
6  mov    %esp,%ecx              89 e1
7  lea    0x2(%esi),%eax         8d 46 02
8  int    $0x80                  cd 80
```

Listing 3: The disassembly of the replacement shellcode `shellcode1`.

If we apply the shellcode byte-replacement method, the modified exploit be:

```
1  shellcode = "\x31\xf6\x8d\x5e\x01\x8d\x56\x08\
   xff\x35\xaa\xaa\xaa\xaa\x68\xdd\xdd\xdd\xdd\
   x89\xe1\x8d\x46\x02\xcd\x80"
2  exploit = shellcode + "\x90" * (36 - len(
   shellcode)) + "\x50\xaf\xaa\xba\n"
```

Listing 4: The modified exploit for `shellcode1` using the shellcode replacement approach.

However, the modified exploit *will not work* when applied to our motivating example. Figure 3a shows the stack layout before function `example` starts. Besides saved registers, there are two variables between `string` and the saved `%eip`. When the program receives an input, the resulting stack layout is shown in Figure 3b. However, control is not immediately transferred to the shellcode. The program continues, and because variable `len` is updated before returning, the value at this address changes. By the time the function transfers control flow to the shellcode, the program changes the 20th through the 28th bytes of the replacement shellcode, as shown in Figure 3c. In our example, this represents unexpected modification to the replacement shellcode, rendering it nonfunctional.

In some cases, previous work is, using very resource-intensive techniques, capable of re-finding the vulnerability and re-creating an exploit, but these systems all suffer from extreme scalability issues because they approach vulnerability detection as a *search problem*. If we do not want to re-execute these resource-expensive systems to re-identify and re-exploit vulnerabilities, a new approach is needed. To this end, we identified two main categories of challenges in shellcode transplanting: one dealing with the layout of memory at the time the vulnerability is triggered, and the other having to do with the actions taken in the path of execution *before* the vulnerability is triggered.

1) Memory conflicts: Previous work [12, 17] places shellcode in memory by querying a constraint solver to solve the constraints generated in the Symbolic Tracing step and concretizing a region of memory to be equal to the desired shellcode. However, as is the case in our naive byte-replacement approach, this is not always possible: often, when dealing with fine-tuned exploits, there is simply not enough symbolic data in the state to concretize to shellcode [29].

For example, recall the shellcode in Listing 4 in the context of our motivating example. This piece of shellcode is 26 bytes long, which *should* have fit into the 50 bytes of user input. However, the 20th through the 28th byte are overwritten, and the 36th through 40th byte must be set to the address of the shellcode (to redirect control flow). This leaves three symbolic regions: a 20-byte one at the beginning of the buffer, an 8-byte one between the `ret` and `len` variables and the saved return address, and the 10 bytes after the saved return address. None of these regions are big enough to place this shellcode, causing a *memory conflict* for the shellcode transplanting process.

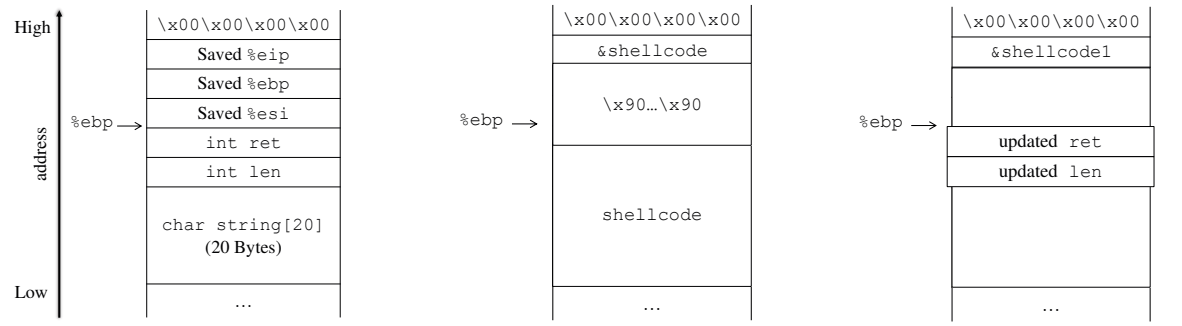
2) Path conflicts: To drive program execution to the exploited state, the content of the modified exploit must satisfy the path constraints recovered from the Symbolic Tracing step. However, by requiring the replacement shellcode to be in the memory of the exploitation state, we add new constraints ("shellcode constraints") on the exploit input. These new conditions may be conflict with those generated along the path. We call such conflict the *path conflict*. In the presence of such a conflict, if we locate the replacement shellcode in the exploitation state (and discard the path constraints that conflict with this), the exploit path will change, and the new program state resulting from the changed path may not trigger the vulnerability.

For example, consider the replacement shellcode in Listing 5 in the context of the motivating example.

```
1  push   $0x0                   6a 00
2  push   $0xa65                 68 65 0a 00 00
3  push   $0x646f636c            68 6c 63 6f 64
4  push   $0x6c656873            68 73 68 65 6c
5  mov    $0x2,%eax              b8 02 00 00 00
6  mov    $0x1,%ebx              bb 01 00 00 00
7  mov    %esp,%ecx              89 e1
8  mov    $0xa,%edx              ba 0a 00 00 00
9  lea    0x10(%esp),%esi        8d 74 24 10
10 int    $0x80                  cd 80
```

Listing 5: The disassembly of the replacement shellcode. `shellcode2`.

When the running example executes with an input string, the `for` loop body before the return increments `i` until `string[i]` is a null byte. For the original exploit, the loop will repeat for 40 times (the length of the exploit string), meaning that the path constraints will mandate that the first 40 bytes of `string` are not null. For the replacement shellcode, however, if we locate the new shellcode at the beginning of



(a) Before receiving an exploit. (b) After receiving an exploit. (c) When `example` returns.

Fig. 3: The stack layout of function `example` at runtime.

`string`, the loop will only iterate once, because the second byte of the shellcode is null. This creates a contradiction between the path constraints and the shellcode constraints.

3) Surmounting the challenges: The intelligent reader can certainly envision approaches to achieve shellcode transplanting in the motivating example. However, this example is just 12 lines of code. One can see that, with bigger examples and in the general case, these challenges can be quite complicated to surmount.

In the rest of the paper, we will discuss how to identify conflicts while transplanting the shellcode and how to satisfy both memory and path conflicts to successfully transplant shellcode in a wide variety of exploits.

III. Symbolic Tracing

Essentially, ShellSwap separates the entire execution of the original exploit into two phases: before the control-flow hijack and after the control-flow hijack. The Symbolic Tracing step analyzes the former. The goal of this step is to generate the *exploitable state* of the program and record the *path constraints* that are induced by conditional branches that are encountered on the path. This involves two main considerations.

First, we must determine when the control-flow hijack occurs. We do this by leveraging the concept of *security policies*, which has been thoroughly explored by researchers [10, 16, 18, 21, 32]. In our work, we use the well-studied taint-based enforceable security policy [26, 32]. This policy determines whether or not a program state is safe by checking the instruction being executed. If the instruction directly is tainted by remote input, then the program state is deemed unsafe and the path is terminated.

Second, we must determine how to perform the tracing, as there are several possible techniques that might be used here. For example, we could use dynamic taint analysis to identify when executed instructions are tainted by input data. While this would be relatively fast, taint analysis is not sufficient. Although it can identify violations to our security policy caused by tainted input, it cannot recover and track path constraints. Thus, in our system, we apply concolic execution to trace the path that the exploit runs on the program. We ensure tracing accuracy in two ways: we record a dynamic trace of the exploit process (and require that our symbolic trace conform to the same instructions), and we *pre-constrain* the symbolic data to be equal to the original exploit. The former avoids the path explosion inherent in concolic execution exploration (because we only care about the branch that the exploit chooses), and the latter greatly simplifies the job of the symbolic constraint solver during tracing (by providing it with a pre-determined solution). This method is similar to the pre-constraint tracing and the input pre-constraining approach proposed by Driller [30] (and, in fact, part of the implementation derives off of Driller's tracing module).

The trace-directed symbolic execution takes a program and an original exploit and produces path constraints and the exploitable state. The exploitable state includes the symbolic value of registers and memory at the moment that the program starts to execute the shellcode. After this step completes, the pre-constraints introduced in the beginning are removed, making it possible to constrain some of the memory in the exploitable state to contain values representing, for example, the replacement shellcode. The remaining path constraints guarantee that any satisfying input will make the program to execute the same execution trace and triggers the vulnerability.

IV. Shellcode Transplant

After the exploitable state and the path constraints associated with it have been recovered, ShellSwap can attempt to *re-constrain* the shellcode to be equal to the replacement shellcode by adding *shellcode constraints*. However, as discussed in Section II, the shellcode constraints may conflict with the path constraints. Previous work [12, 17] addresses this issue by trying other shellcode locations, but even the simple motivating example in Section II is too complicated for this to work.

The Shellcode Transplant steps attempts to resolve these conflicts. If it can do so, the modified exploit, containing the replacement shellcode, is produced. If it fails, it returns an error indicating that the exploit could not be found.

The step proceeds in several phases, in a loop, as shown in Figure 4. First, in the *Preprocessing* phase, ShellSwap identifies possible memory locations into which replacement shellcode

(or pieces of it) can be placed. Next, in the *Layout Remediation* phase, it attempts to remedy memory conflicts (as discussed in Section II-B) and fit the replacement shellcode into the identified memory locations, performing semantics-preserving modifications (such as code splitting) if necessary. If this fails due to a resulting conflict with the path constraints (a path conflict, as discussed in Section II-B), ShellSwap enters the *Path Kneading* phase and attempts to identify alternate paths that resolve these conflicts while still triggering the vulnerability. If such a path can be found, its constraints replace the path constraints, and the system repeats from the preprocessing phase.

If ShellSwap encounters a situation where neither the memory conflicts nor the path conflicts can be remedied, it triggers the *Two-Stage Fallback* and attempts to repeat the Shellcode Transplant stage with a fallback, two-stage shellcode.

A. Preprocessing

Before the system tries to locate the new shellcode, it scans the memory in the exploitable state to identify *symbolic buffers*. A symbolic buffer is a contiguous memory where all bytes are symbolic. To find symbolic buffers, our system iterates the bytes of the memory, marking each contiguous region. After finding all symbolic buffers, we sort the buffers by the length and the number of symbolic input variables involved in each buffer. Buffers with bigger length and more symbolic values has more varieties of concrete values, and thus are more likely to be able to hold the replacement shellcode.

B. Layout Remediation

Given symbolic buffers from the previous phase, the system attempts to fit the replacement shellcode into the exploitable program state. As an innovation over prior work, ShellSwap does not consider a piece of shellcode as an integrated memory chunk. Instead, we model the new shellcode as a sequence of instructions. It is not necessary to keep these instructions contiguous; we could insert `jmp` instructions to "hop" from one shellcode instruction in one symbolic buffer to another instruction in another buffer. Thus, we attempt to fit pieces of the shellcode (plus any necessary jump instructions) into previously-identified symbolic buffers.

Algorithm 1 and Algorithm 2 shows the algorithms for Layout Remediation. The system invokes function `Locate`, and function `Locate` calls out to function `Hop` when needed. Both functions take five arguments as input: SH, ST, I, C, i, a, where SH is the shellcode, ST is the exploitable state, I is the symbolic buffers, C is the set of constraints for ST, i is an index into the not-yet-written bytes of the replacement shellcode, and a is the memory address being currently considered by the algorithm.

We use the motivating example to demonstrate how the algorithm works. As mentioned in Section II-B, there will be three symbolic buffers in this example. Suppose the ShellSwap system tries to fit the shellcode from Listing 3 to the stack of the exploitable state of the motivating example. It calls `Locate` with $i = 0$ and $a =$ `&string`, initially trying

Input :
SH: The new shellcode
ST: The current exploitable program state. $ST.mem[j]$ means the memory at j in the state ST
I: The symbolic buffers generated by preprocessing
C: The constraints set
i: The index of the instruction of the shellcode
a: The start address that we plan to put $SH[i]$
Output :
E: A new exploit or Not Found

```
1  if i > len(SH) then
       // We have successfully put the entire piece
          of shellcode to the exploitable state.
2      E ← Solve(C);
3      return E;
4  end
5  else if i < 0 then
       // We cannot successfully put the entire piece
          of shellcode to the exploitable state if we
          put SH[i] at a.
6      return Not Found;
7  end
8  else
9      if I has enough space after a then
           // Construct the new constraint asserting
              the memory at a concretize to the i-th
              byte of the replacement shellcode.
10         c ← (ST.mem[a : a + len(SH[i])] == SH[i]);
11         C' ← C + c;
12         if Solve(C') has solution then
13             ST' ← a new state with
                 ST.mem[a : a + len(SH[i])] = SH[i];
14             a' ← Next(I, a + len(SH[i]));
15             return Locate(SH, ST', I, C', i + 1, a');
16         end
17         else
               // We cannot put SH[i] at a. Instead, we
                  need to find another location for
                  SH[i] and hop to the location.
18             if Hop(SH, ST, I, C, i, a) == Not Found then
19                 return Not Found;
20             end
21             else
22                 ST', a', C' ← Hop(SH, ST, I, C, i, a);
23                 return Locate(SH, ST', I, C', i + 1, a');
24             end
25         end
26     end
27     else
28         return Not Found
29     end
30 end
```

Algorithm 1: The algorithm of the `Locate` function.

to put the first instruction of the replacement shellcode at the beginning of the buffer `string`.

The layout remediation process is shown in Figure 5. As the first 6 instructions of the replacement shellcode satisfy the constraints in memory, the process will continue adding new instructions until the 7th instruction (Figure 5b). At this point, the system fails to add the 7th instruction (because `len` is in the way), so it calls function `Hop`, trying to jump over `len` and place the 7th instruction to into the next symbolic buffer (Figure 5c). In function `Hop`, it successfully finds a location for the 7th instruction. However, the `jmp` instruction cannot fit after the first 6 instructions (Figure 5d, so we roll back and call `Hop` to re-locate the 6th instruction (Figure 5e). Since the `jmp` instruction still covers `len`, this rollback occurs again,

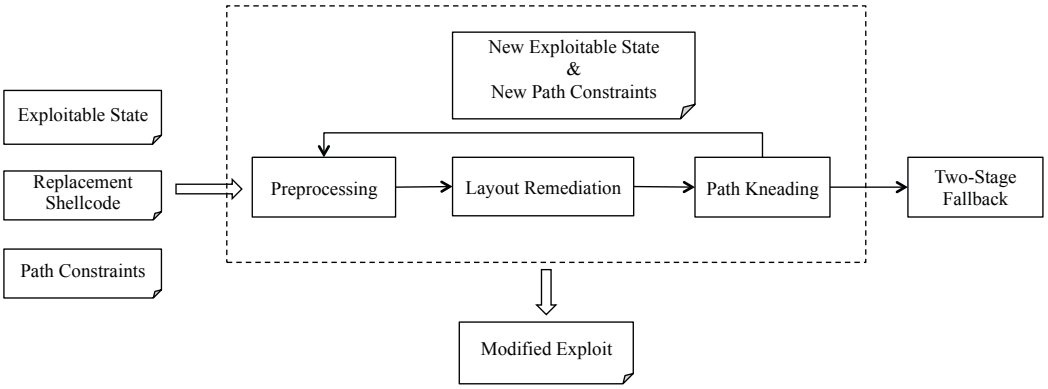


Fig. 4: The phases of the Shellcode Transplant step.

Input :
SH: The new shellcode
ST: The current exploitable program state. $ST.mem[j]$ means the memory at j in the state ST
I: The symbolic buffers generated by preprocessing
C: The constraints set
i: The index of the instruction of the shellcode. $SH[i]$ means the bytes for the i-th instruction of the shellcode SH.
a: The start address that we plan to put $SH[i]$
Output :
ST': The updated exploitable program state, with the jump instruction and SH[i] in the memory.
C': The updated constraints set
a': The start address for the next instruction

```
1  if i < 0 then
       // We cannot successfully hop SH[i].
2      return Not Found;
3  end
4  else
       // find an address to put SH[i]
5      a' ← None;
6      a_t ← Next(I, a + len_jmp));
7      while a_t is not None do
8          c ← (ST.mem[a_t : a_t + len(SH[i])] == SH[i]);
9          C' ← C + c;
10         if Solve(C') has solution then
               // SH[i] can be put at ST.mem[a_t]
11             c_jmp ← jump instruction constraint;
12             C'' ← C' + c_jmp;
13             if Solve(C'') has solution then
                   // The jump instruction can be put
                      at ST.mem[a]
14                 ST' ← a new state with SH[i] and jump instruction;
15                 a' ← Next(I, a_t + len(SH[i]));
16                 return ST', a', C';
17             end
18         end
19         else
20             a_t ← Next(I, a_t));
21         end
22     end
       // We cannot hop to an address with SH[i]
          after address a. Then we roll back and hop
          to the previous instruction.
23     ST', a', C' ← Rollback(SH, ST, C, I, a);
24     return Hop(SH, ST', I, C', i − 1, a');
25 end
```

Algorithm 2: The algorithm for the `Hop` function.

until the *5th* instruction ends up relocated, and a `jmp` inserted after the 4th instruction to the 5th instruction. In the end, this is repeated until the full shellcode is placed in memory, split into three parts as shown in Figure 5f.

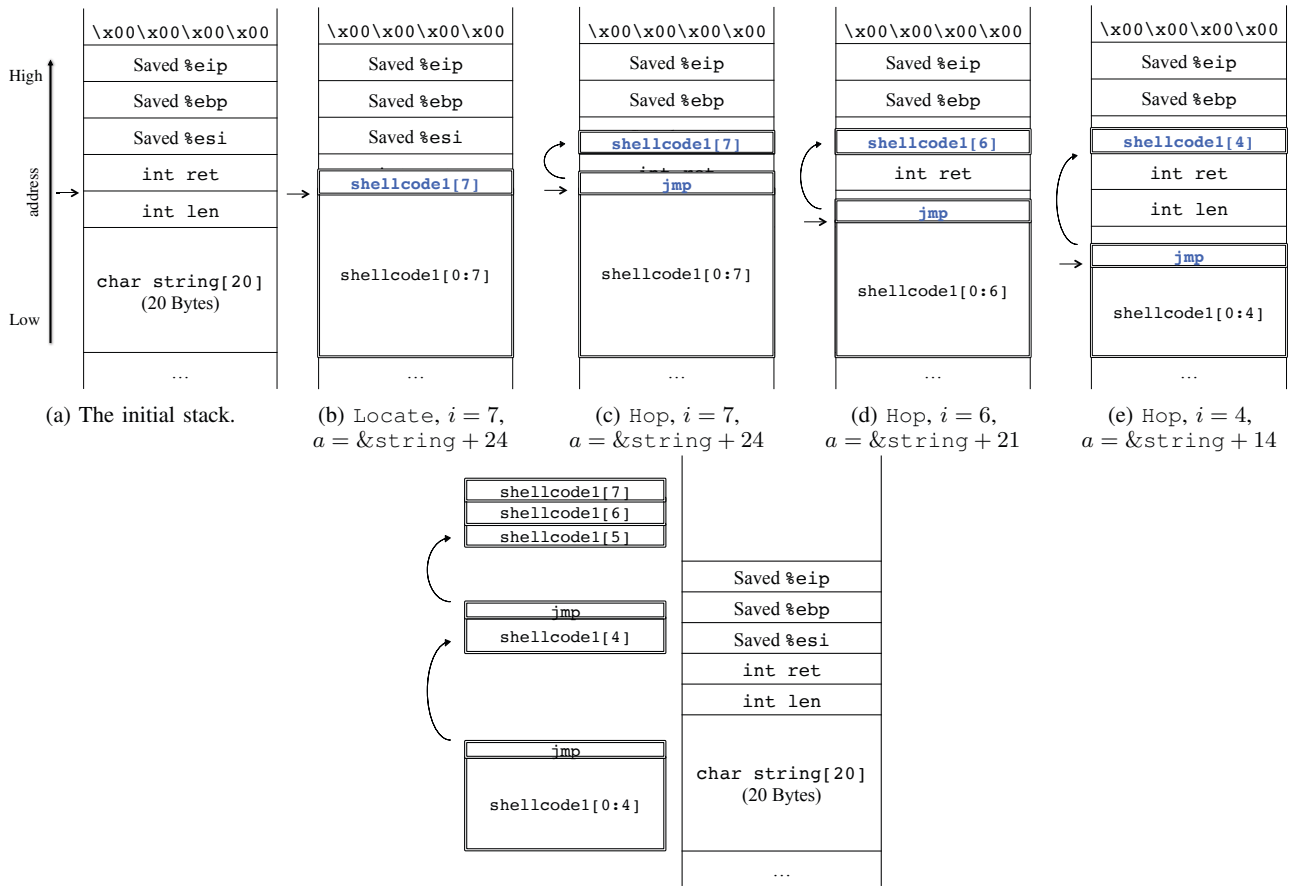
C. Path Kneading

If the system cannot find a new exploit for the new shellcode using the exploitable state of the original exploit, we need to diagnose the cause of conflict and tweak the path to generate new exploitable states and new path constraints. To diagnose the cause of conflict, we first identify the conflicting path constraints and then check which instructions generated them.

Since shellcode is placed to the exploitable state instruction by instruction, we can retrieve the smallest set of shellcode constraints that cause a path conflict as soon as `Locate` terminates unsuccessfully. Let c be the constraint for locating the current instruction, and let C be the set of path constraints set of the current state. We already know that c and C are conflicted (otherwise, a location for the last instruction would have been found), which implies that $c \wedge C = \mathit{False}$. To understand the cause of the conflict, we find the smallest set of path constraints S such that: $S \subseteq C$, such that:

$$c \wedge S = \mathit{False} \text{ and } c \wedge (S - C) = \mathit{True}.$$

After finding the conflict subset, ShellSwap identifies the source of each constraint in this subset by checking the execution history for when it was introduced. If the conflicting constraint was introduced by condition branch, ShellSwap will tweak the path to *avoid* the path constraints in the conflict subset. The intuition for this is as follows: if the shellcode constraint contradicts a path constraint, then the shellcode constraint does *not* contradict the *negation* of that path constraint. For path constraints created by conditional branches, our idea is to negate the conflict path constraints by selecting the other branch in the program. In this way, if the program executes along the path with the opposite branch, the new path constraints will contain the negation of the previously-conflicting path constraint, and the new path constraints will not conflict with the shellcode constraint c.



(a) The initial stack. (b) Locate, $i = 7$, $a = \&\texttt{string} + 24$ (c) Hop, $i = 7$, $a = \&\texttt{string} + 24$ (d) Hop, $i = 6$, $a = \&\texttt{string} + 21$ (e) Hop, $i = 4$, $a = \&\texttt{string} + 14$

(f) The final shellcode layout.

Fig. 5: The layout remediation process for the motivating example with `shellcode1`.

For example, consider the motivating example and the replacement shellcode in Listing 5. As we described in Section II-B, we encounter a path conflict because the `for` loop in our example, which runs 40 times for the original shellcode, only runs once for the replacement shellcode. Let E be the exploit, and let E_i be the i-th byte in E. The symbolic value of `string` in the exploitable state is equal to:

$$\texttt{Concatenate}(E_0,\ E_1, \ldots,\ E_{18},\ E_{40})$$

which means the string from the 0th to the 40th byte of the input. In this case, the path constraints include the following:

$$E_0 \neq \texttt{'\x00'} \wedge E_1 \neq \texttt{'\x00'} \wedge \ldots \wedge E_{40} \neq \texttt{'\x00'}$$

However, because the second character of the replacement shellcode is `'\x00'`, the shellcode constraints conflict with the path constraints[1].

Suppose that ShellSwap identifies this situation while trying to place the first instruction of the replacement shellcode at the beginning of `string`. After analyzing the conflicting constraint subset, we know that the conflict stems from the path constraint $E_1 \neq \texttt{'\x00'}$, and this constraint is created at address `0x080482F3`, shown in Figure 6. Specifically, the conflict constraint occurs at the second iteration of the for loop.

To generate a new path, we negate the conditional jump associated with the conflicting path constraint by modifying the trace to force an exit from the loop after the second iteration. However, after this change, we need to merge the diversion back to the original path. We accomplish this by leveraging static analysis. First, we find the function containing the divergence point, and build a control flow graph for the specific function. Next, we statically find the descendants of the diverted node and see if any of the descendants appear in the original path after the negated node. For each satisfying descendant, we attempt to construct a new path that is identical to the original path until the negated node, followed by the detected detour back to the descendent node that appears in the original path, and then ending with the postfix from the descendant node to the end of the original path.

Figure 7 shows the generation of a new path. Suppose node n_c is negated to $n_{c'}$, and node n_d is the descendant of node

[1]The inquisitive reader might question why the part of the shellcode with the null byte could not be written *after* the return address to bypass this loop. However, a closer look at the replacement shellcode would reveal null bytes in many other locations as well.

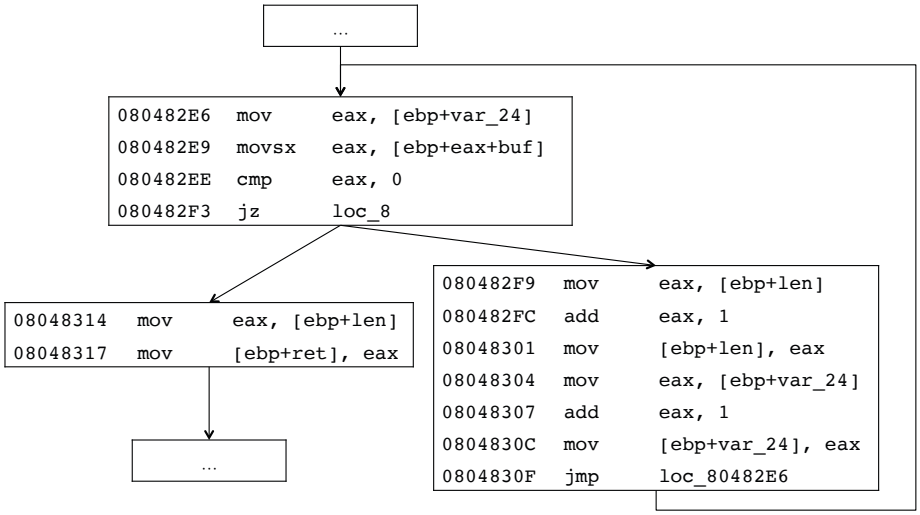


Fig. 6: Part of the control flow graph for the motivating example.

$n_{c'}$. For the new path, the basic blocks do not change before $n_{c'}$ or after n_d. In between, we insert an intraprocedural path from $n_{c'}$ to n_d G($n_{c'}$, n_d), which can be generated using the control graph of the function. In the best case, the question is equivalent to finding a path between two nodes in a directed graph. However, it is possible that there is no such path to rejoin the original path, or that the problem reduces to symbolic exploration (if the divergence is too big). In this case, ShellSwap falls back on the Two-Stage Fallback.

In the motivating example, as simply exiting the loop already rejoins the original path, the detour back to the path is trivial: it is the direct jump to the return site of the `example` function.

After constructing the new path, the ShellSwap system generates the new exploitable program state and a new set of path constraints using the Symbolic Tracing step. Meanwhile, it also checks if the new program state is still exploitable. If the new program state is exploitable, our system starts again from the preprocessing phase to fit the replacement shellcode into the new exploitable program state. Otherwise, the system will attempt to construct the other paths and generate the other program states, falling back on the Two-Stage Fallback if it is unable to do so.

D. Two-Stage Fallback

If ShellSwap is unable to overcome the memory and path conflicts and fit the replacement shellcode into the exploitable state, then it falls back on pre-defined a two-stage shellcode instead of the provided replacement shellcode. The motivation of this fallback is straightforward: if the provided shellcode cannot fit the exploitable state, even after Path Kneading, we try a smaller first-stage replacement shellcode that can then load an arbitrary second-stage shellcode.

There are several options for a first-stage shellcode. One option is a shellcode that reads shell commands from the socket and executes them. Another, to bypass modern defenses such as Data Execution Protection, could read a Return Oriented Programing payload over the stack and initiates a return. For our prototype, we implemented a stack-based shellcode-loading first-stage payload that reads a second-stage payload onto the stack and jumps into it. While this is not immune from DEP techniques, it is only meant as a proof of concept for our prototype.

Consider the motivating example. The program receives input by using the DECREE syscall `receive()` (more information on DECREE is provided in Section VI), which is a system call similar to `recv()` in Unix/Linux. If the new shellcode is longer than 50 bytes, we cannot generate a new exploit because the program is able to receive 50 bytes at most. In this case, we could consider the following template for generating a two-stage shellcode:

```
1  xor   %eax,%eax       31 c0
2  inc   %eax            40
3  inc   %eax            40
4  inc   %eax            40
5  xor   %ebx,%ebx       31 db
6  inc   %ebx            43
7  mov   %esp,%ecx       89 e1      ; %ecx: &dst
8  mov   _ ,%edx         8b _       ; %edx: len
9  mov   _,%esi          8b _       ; %esi: &ret
10 int   $0x80           cd 80
11 jmp   *%esp           ff e4
```

Listing 6: The disassembly of the template for a two-stage shellcode.

This first-stage shellcode reads a string, stores at the bottom of the stack (`%esp`) and jumps to the received string. There are two blanks in the template – we need to fill the receiving length and the address of return value for register `%edx` and `%esi`, respectively. After completing the template, our system will restart the layout remediation process with the two-stage shellcode as the replacement shellcode. If the system cannot find a modified exploit using the Two-Stage Fallback, it returns an error indicating that no modified exploit could be found.

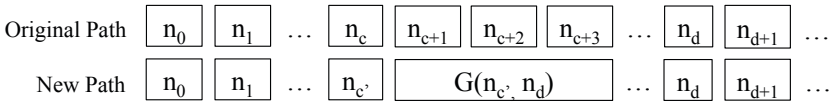


Fig. 7: The generation of a new path. G(x, y) means a path between node x and y found by static analysis.

Although the two-stage shellcode helps to solve the shellcode transplant problem by increasing the situations in which ShellSwap can function, we consider this purely as a fallback. This is because two-stage exploits may be less robust than the other exploits, as they assume that the victim machine can receive extra bytes from the attacker. This assumption does not always hold. For instance, the victim machine may be protected by other mechanisms which block the message, such as an external firewall, or the network connection over which communication happens might already be closed when the vulnerability triggers. Therefore, our system prioritizes the conflict resolution approaches, and it will not trigger the Two-Stage Fallback when the previous layout remediation process fails.

V. Implementation

ShellSwap is implemented on top of `angr` [29], a binary analysis platform. We rely on angr's symbolic tracing component [3], which also leverages the QEMU emulator [1] for exploit replay and symbolic tracing. The core of our system, consists of about 2000 lines of Python code.

A. Finding Infeasible Constraint Sets

Finding a minimal subset of infeasible constraints, which is an essential part of Path Kneading, is not a trivial problem. The underlying constraint solver Z3, which is used in angr (and thus in ShellSwap), provides an `unsat_core` function to retrieve the *smallest* subset of an unsatisfiable set of constraints. However, in our experiment, we found that `unsat_core` can be very time consuming, and sometimes even lead to crashes of Z3. Since we weren't able to pinpoint the root cause of the problem, we further implement a constraint set slimming method (as described below) to resort to in case `unsat_core` fails.

The constraint set slimming is a divide-and-conquer approach. Given a constraint set A and a constraint c that contradicts A, constraint set slimming will try to find a subset of constraints in A (but not the smallest subset) that still contradicts c. We first divide A into two subsets and check if any of them is contradictory to constraint c. If both subsets contradict c, the final infeasible constraint set will include conflicting constraints subsets from the two. If only one subset contradicts c, the other subset can be safely discarded as the result will only contain conflicting constraints from the contradictory subset. We repeat this procedure on contradictory subsets recursively until we find the very last contradictory subset, which either contains a single constraint that contradicts c, or several constraints that none of which contradicts c if considered individually. The union of all conflicting subsets of constraints represent the slimmed set of constraints.

B. Optimizations

Much of the execution in symbolic tracing does not involve symbolic data. To speed up the tracing step, ShellSwap enables code JIT'ing (through the use of Unicorn Engine [24]) by default, which allows instructions in the original exploit to be executed natively instead of being emulated. While it greatly speed up symbolic tracing, we find that this step is still the bottleneck in ShellSwap: as discussed in Section VI, an average of 95% of execution time is spent in this step.

To avoid generating an entire control-flow graph in our path kneading component, we used a fast function detection approach to pick out the exact function for which to generate the control flow [9].

In the course of the development of this system, we have upstreamed many big-fixes and some improvements to angr and its tracing module. With these fixes, we observed a *1000-times* speed improvement on some samples in our evaluation.

VI. Evaluation

In this section, we present our evaluation of ShellSwap. We first describe the data set, including all vulnerable programs and exploits, used in our evaluation (Section VI-A). Then, we show the experimental setup in Section VI-B. Next, we demonstrate the effectiveness of our approach in Section VI-C by evaluating both ShellSwap and a reference implementation of previous work on 20 original exploits and 5 pieces of replacement shellcode. There, we show the necessity of ShellSwap in effectively transplanting shellcode. In the end, we evaluate the efficiency of ShellSwap and display the results in Section VI-D.

A. Data Set

Our evaluation data set contains three parts: 11 vulnerable binaries, 20 original exploits, and 5 pieces of replacement shellcode. We present how the data set is constructed below.

1) Vulnerable binaries: We selected 11 vulnerable binaries (see Table I) from the qualifying event as well as the final event of DARPA Cyber Grand Challenge (CGC). These binaries are shipped with source code, reference exploits, and actual exploits generated by other CGC participants, making them a perfect fit for our evaluation. All of the binaries are standalone x86 binaries with a special set of system calls (DECREE syscalls), roughly analogous to the Linux system calls `recv` (as DECREE's `receive`), `send` (as DECREE's `transmit`), `mmap` (as DECREE's `allocate`), `munmap` (as DECREE's `deallocate`), `select` (as DECREE's `fdwait`), `get_random` (as DECREE's `random`), and `exit` (as DECREE's `_terminate`). Sizes of those binaries range from 83 KB to 18 MB. Those vulnerable binaries cover a wide range of subtypes of control flow hijack

vulnerabilities, including stack overflow, heap overflow, integer overflow, arbitrary memory access, improper bound checking, etc.

2) Exploits: As the CGC provides generators for reference exploits, we generated a few exploits for each vulnerable binary, for a total of 20 reference exploits (as is shown in Table I). It is worth noting that exploits (or *Proofs of Vulnerability* in CGC terminology) in CGC are special in the sense that each of them should demonstrate attacker's ability to fully control values in two registers: the instruction pointer and one other register. As a result, some generated exploits do not contain any shellcode. We manually post-processed all exploits to make sure each one of them has a piece of shellcode to execute in the end of the exploitation.

3) Shellcode: As shown in Table II, we collected five instances of replacement shellcode from three different sources, four of which are from CGC finalists (ForAllSecure and Shellphish), and one of which is manually crafted by ourselves. This range of replacement shellcode instances is important: with the shellcode coming from multiple sources, we can mimic the setting of cyber attack customization in our experiments. We refer to these instances as S_1 through S_5. Therefore, with five instances of replacement shellcode for each of the 20 original exploits in our dataset, we have a total of 100 modified exploits for ShellSwap to generate.

B. Experiment Setup

One of the applications of transplanting shellcode is to automatically reflect, or *ricochet*, an attack coming from a rival. In this scenario, the victim first detects an exploit coming from the attacker. They then automatically replace the payload (the shellcode) in the exploit and replay the modified exploit against the attacker. We try to simulate such a scenario in our experiment, where the attacker emits original exploits and the victim (or replayer/reflector) replays a modified exploit with the shellcode replaced.

1) Machines.: Our experimental setup contains two machines: one machines hosts the DARPA Experimental Cyber Research Evaluation Environment (DECREE), and the other runs ShellSwap. DECREE runs on a virtual machine built using an image provided by DARPA CGC [5, 6], which offers an isolated environment for running and testing vulnerable programs. It is assigned 1 CPU core and 1 GB of memory on a host machine with Intel Core i7 2.8 GHz. The ShellSwap machine is a standalone server with Intel Xeon E5-2630 v2 as CPU and 96 GB of memory, running Ubuntu 14.04 LTS.

2) Process.: As is shown in Figure 8, the original exploits are pre-generated for each vulnerable binary. ShellSwap takes as input each pair of original exploit and replacement shellcode and attempts to generate a modified exploit. We verify the modified exploit against the binary in DECREE box to make sure that it works and that the replacement shellcode is executed with intended results. For testing and verification, we modified the utility script `cb-replay-pov` shipped in DECREE.

3) Reference system for comparison.: To demonstrate the necessity of our approach in tackling the shellcode transplant problem, we reimplemented the shellcode placement method in the work of Cha et al. [12] in a new system on top of angr and used it as our reference system (codenamed SystemM). We simulate shellcode transplanting in SystemM by first re-triggering the exploit and then re-constraining individual symbolic blocks in memory to the replacement shellcode one by one until the modified exploit is created. If none of the symbolic memory blocks is sufficiently large to hold the replacement shellcode, or constraining every symbolic memory block to replacement shellcode leads to an unsatisfiable exploitation state (due to path conflicts), then we deem the shellcode transplanting as having failed.

C. Effectiveness

Table I presents the effectiveness comparison between SystemM and ShellSwap. There is a significant difference between the number of modified exploits the two systems successfully generated: SystemM successfully generated 31 exploits, whereas ShellSwap successfully generated 88 exploits. The success rate for SystemM and ShellSwap are 31% and 88%, respectively. Not surprisingly, our method generated more new exploits than previous work.

Statistics for all modified exploits successfully generated by SystemM and ShellSwap are shown in Table III. ShellSwap generated 57 exploits using only Layout Remediation and 31 more by leveraging Path Kneading. For comparison, we also extended SystemM with Layout Remediation, resulting in, as expected, an additional 26 more exploits over the base SystemM implementation. Only 57% of all cases are successfully replaced with new shellcode without Path Kneading, which demonstrates the importance of conflict identification and kneading of the exploit path during shellcode replacement.

In addition, we evaluate the two-stage fallback on all 20 exploits: we replace the original shellcode in each exploit with the fallback shellcode and generate new exploits[2]. In our experiment, the two-stage fallback worked on 19 out of 20 exploits. This is because the fallback shellcode is shorter (19 bytes) than any instance of the replacement shellcode, and is thus more likely to fit into buffers under attacker controls.

Meanwhile, we observe that the success rate of shellcode transplanting varies between different instances of replacement shellcode (see Table IV). There is an expected negative correlation between the success rate and the length of the replacement shellcode. For example, shellcode S_4 and S_5, which are both 37 byte long, have lower success rates than other replacement shellcode that are shorter. This fits with our intuition that the longer a piece of shellcode is, the more conflicts it might produce during the shellcode transplant step, and the more difficult it will be to generate a modified exploit.

Other results are less intuitive. For instance, S_5 has a lower success rate than S_4, which is the same size. We looked into failure cases, and we found that the failure is related to the null byte in S_5. S_4 does not contain any null bytes. This conforms

[2]We do not evaluate all five instances of shellcode since any shellcode will work in the second stage.

Binary	Size	Vulnerability Type	Original Exploits	Modified Exploits	SystemM	ShellSwap
CADET_00001	83 KB	Buffer Overflow	1	5	4	5
CROMU_00001	92 KB	Integer Overflow	1	5	3	5
EternalPass	18 MB	Untrusted Pointer	2	10	0	8
Hug_Game	3.1 MB	Improper Bounds Checking	1	5	0	2
LUNGE_00002	1.6 MB	Off-by-one Error	1	5	14	5
On_Sale	125 KB	Buffer Overflow	4	20	10	20
OTPSim	106 KB	Improper Bounds Checking	2	10	0	10
Overflow_Parking	92 KB	Integer Overflow	1	5	0	0
SQL_Slammer	102 KB	Buffer Overflow	3	15	0	15
Trust_Platform_Module	89 KB	Buffer Overflow	3	15	0	15
WhackJack	105 KB	Buffer Overflow	1	5	0	3
Total			20	100	31	88

TABLE I: This table shows the vulnerable binaries, the types of their vulnerabilities, the numbers of original exploits of each binary, the total number of attempted exploit modifications (one replacement shellcode per original exploit per binary), and the number of modified exploits successfully produced by SystemM and ShellSwap.

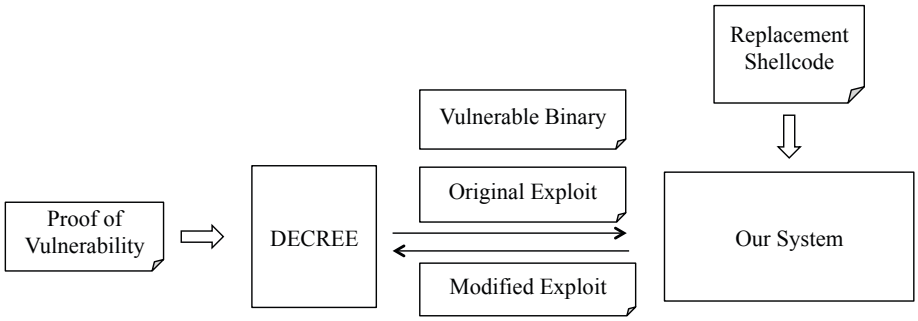


Fig. 8: Experiment setup.

Shellcode	Length	# Instruction	Source
S_1	26 Bytes	7	ForAllSecure
S_2	29 Bytes	11	ForAllSecure
S_3	22 Bytes	12	Shellphish
S_4	37 Bytes	8	Shellphish
S_5	37 Bytes	12	ShellSwap

TABLE II: The shellcode information.

Shellcode	Layout Remediation	Path Kneading
S_1	12	7
S_2	13	5
S_3	14	5
S_4	9	8
S_5	9	6
Total	57	31

TABLE III: The number of the generated exploits for each shellcode and each approach.

to the common knowledge that null bytes complicate shellcode, which is why they are generally avoided by exploit authors: since null bytes are so frequently used as string terminators,

Shellcode	Length	# Success	Success Rate
S_3	22 Bytes	19	95%
S_1	26 Bytes	19	95%
S_2	29 Bytes	18	90%
S_4	37 Bytes	17	85%
S_5	37 Bytes	15	75%

TABLE IV: Success rate for each instance of replacement shellcode, sorted by length.

the existence of null bytes may negatively impact the success of the exploit if data is moved around using something like `strcpy`.

D. Efficiency

Table V shows the time cost for each instance of replacement shellcode and each approach. The average time cost for Layout Remediation is 19.73 seconds, while the average time cost for Path Kneading is 9426.99 seconds. The dramatic difference between the two is because the latter requires one or more iterations of symbolic tracing, which, as we have previously discussed, is an extremely time consuming process. We leave further performance improvement as future work, and note that

Shellcode	Layout Remediation	Path Kneading
S_1	18.85	5638.30
S_2	21.05	10993.84
S_3	20.38	8017.11
S_4	21.01	7993.11
S_5	17.36	14492.62
Average	19.73	9426.99

TABLE V: Average time cost (in seconds) for each instance of replacement shellcode and each approach.

there are example optimizations in related work that could be applied to this problem.

VII. Discussion

ShellSwap's results open up new possibilities for the fast adaptation and analysis of software exploits. In this section, we explore the implications of these results, the limitations of the system, and the direction of our future work.

A. Ethical Concerns

ShellSwap raises the concern that it enables malicious attackers to quickly adapt exploits against unwitting victims on the internet. Unfortunately, such criticism can be applied to almost all security research. Similar to known techniques such as automatic exploit generation [7, 12] or automatic patch-based exploit generation [11], the merit of the ShellSwap system and its solution of the shellcode transplant problem is to show the potential abilities of attackers and to highlight the possibility that one can automatically modify exploits to tailor attacks to custom requirements. Our hope is that, by showing that this is possible, ShellSwap will motivate new research into defenses against customized exploits.

B. Limitation

While ShellSwap makes fundamental contributions toward the solution of the shellcode transplant problem, there is still work left to be done. Here, we discuss specific weaknesses of the system that could be addressed by future work.

1) Other types of vulnerabilities: Our system focuses solely on control-flow hijack vulnerabilities, and we do not address other vulnerability types, such as Information Leakage and Denial of Service (DoS). To consider these types of vulnerabilities, as well as other popular types, the shellcode transplant problem would need to be redefined, as shellcode is not utilized in exploits targeting these vulnerabilities. Thus, to generalize ShellSwap, we must first define the analogous problem in the context of a different vulnerabilities, and then discuss possible designs to solve it.

We define the analogous problem for information leakage vulnerabilities as the generation of a modified exploit that leaks a *different* piece of data (whether a memory location, a file, a variable in the program, etc.) than the original exploit does. This is a complex task to accomplish: information leakage exploits are hard to detect in the first place because monitoring the information flow through a program is not EM-enforceable in general. However, weaker variants such as taint tracking can find a smaller set of information leakage vulnerabilities. For example, evidence shows that Valgrind can detect information leakage exploits such as the Heartbleed attack [31], given test cases that trigger it (i.e., an exploit). Since, by definition, ShellSwap receives such an exploit as input, a possible method for ShellSwap to function on information leakage is to use symbolic execution to find the correlation between the exploit and the leaked information or its reference, and modify it accordingly. In this case, the memory conflicts will likely not come into play (since they are specific to placing replacement shellcode in memory), but path conflicts will still occur, and will need to be kneaded away, due to the modifications required to re-target the leak. After identifying the relation, one can come up with an exploit by solving the constraints.

We define the ricochet problem for Denial of Service vulnerabilities as the generation of a modified exploit that causes the same effect to the vulnerable program. Of course, there is little modification required – if the original exploit makes the program crash or hang at a given point, the modified exploit should have the same effect. In this case, ShellSwap is used purely as an exploit replaying system.

2) Exploit Replayabilty: ShellSwap assumes that the original exploit is deterministically replayable, in the sense that the exploit always succeeds when re-launched against the target. However, this assumption does not always hold. For instance, a vulnerable server may implement a challenge-response protocol that requires the client to send messages with a nonce that the two sides negotiated at the beginning of the session. This nonce would change when we replay the exploit, and the exploit would fail. Asymmetric encryption and sources of randomness from the environment can also manifest in such failures. To generate the modified exploit for such case, ShellSwap would have to consider an exploit as a state machine rather than a series static bytes, which would require fundamental extensions of the design.

This being said, our experiments showed that most of the exploits in our dataset *are* replayable, and our system is applicable for this majority. We intend to investigate the replaying of non-deterministic exploits in future work.

3) Modern Defense Mechanisms: Modern systems have memory protection mechanisms such as Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP). However, such protection mechanisms can be bypassed by properly-crafted exploits.

Our solution to the shellcode transplant problem is based on an functional original exploit, which implies that this exploit has already bypassed the required defense mechanisms. When this is the case, ShellSwap's replacement exploit often bypasses these mitigation techniques as well. For example, DEP is often bypassed through the use of Return Oriented Programming that chains pieces of code (termed *gadgets*) in a program to map an executable page (using the Linux `mmap` or DECREE `allocate` syscalls), insert shellcode into it, and execute it. Alternatively, the page containing the shellcode (for example, the program stack) can simply be marked executable

by `mprotect`. For such exploits, ShellSwap bypasses DEP by reusing the original exploit's DEP bypass and replacing the final mapped shellcode with the replacement shellcode. If the replacement shellcode cannot be located at the same location as the original shellcode, the final control flow transfer of the mitigation bypass stage must be modified to point at the new location. This can be done with the constraint solver as an adaptation of the Path Kneading phase discussed in Section IV.

However, in the more general case of DEP bypass (for example, when a pure ROP payload is used, with no mapped shellcode), future work is required to solve the *ROP chain transplant problem*.

Bypassing ASLR is similar. One way to bypass ASLR, in the absence of DEP, is to overwrite the instruction pointer to point to `jmp *%reg` with a register `%reg` referring to a register location that currently points to the shellcode. A typical instance of the instruction is `jmp *%esp`. For the ShellSwap system, the modified exploit is able to bypass the ASLR protection if 1) the original exploit is able to bypass ASLR and 2) the beginning of the replacement shellcode is placed at the same start address as the shellcode in the original exploit (to which control flow is transferred after DEP bypass, for example). In this way, when the program dereferences a function pointer or returns a function, it will jump to the address of the start of the original shellcode, which is also the start of the replacement shellcode, and the modified exploit will succeed. Again, the final shellcode location can be modified through an adaptation of the Path Kneading phase.

More complex cases, including exploits that require an information disclosure step (to break ASLR), are currently not supported by ShellSwap. We plan to explore these in future work, and would welcome collaboration in this area.

C. Future Work

We plan to explore, and hope to see other researchers investigate, four main areas of future work.

First, ShellSwap can be extended to deal with encrypted, packed, or obfuscated traffic. In theory, our approach can handle these cases, because we assume knowledge the encryption key and because the decryption/decoding/deobfuscation functionality is in the original binary. However, the exploration of cases that do *not* assume knowledge of the encryption key would be interesting (albeit probably impossible in cryptographically-secure cases). A further generalization of this is the ability to successfully transplant shellcode in the presence of nondeterminism. Currently, ShellSwap cannot handle nondeterministic behavior, and some fundamental problems would need to be addressed to enable its operation on this.

Second, it would be interesting to make ShellSwap usable in an on-line capacity, where instead of modifying exploits and launching them at a later date, ShellSwap could perform the exploit live against the remote system, modifying it as appropriate based on that system's operation. Symbolic tracing is the current bottleneck of achieving this capability, but it can likely be improved by leveraging optimizations from related work [8, 25, 28]. Interestingly, the ability to function on-line would allow ShellSwap to reason about information disclosure in the course of an exploit to defeat ASLR, which is something that is not currently possible.

Third, the extension of ShellSwap to the *ROP chain transplant problem* would be an interesting future direction. Related work in the field of automatic ROP payload generation can be leveraged toward this end [27, 29].

Finally, ShellSwapcan be expanded to support the generation of shellcode that is semantically equivalent to the replacement shellcode while having different contents to satisfy path constraints. Such shellcode polymorphism would increase the cases in which ShellSwap can resolve path conflicts. For example, we could consider building up a dictionary of "instruction synonyms", or creating templates to interchange instructions without changing the semantics.

VIII. Related Work

A. Automatic Exploit Generation

An exploit is valuable to attackers only when it suits attackers' specific goal. The technique of automatically generating an exploit with a piece of shellcode is called automatic exploit generation (AEG) [7, 11, 20, 27]. Those work are mostly based on dynamic symbolic execution. AEG is closely related to ShellSwap in the sense that they both take a vulnerable program and a piece of shellcode and generate a viable exploit.

Helaan et al. [17] proposed how to place shellcode in memory: scan through the memory and find symbolic memory gaps that are big enough to hold the entire piece of shellcode. For each gap, they try to put shellcode at different offsets by constraining symbolic memory bytes beginning at that offset to the actual bytes of the shellcode. This procedure continues until the shellcode is put in a memory gap or all gaps have been tried.

As we have demonstrated in our evaluation, AEG techniques are not suitable for shellcode transplanting, as they lack principled approach to diagnose and resolve conflicts imposed by replacement shellcode, and must resort to symbolic exploration. Our system makes it possible to *adapt* and *retrofit* an existing exploit to different instances of shellcode efficiently.

B. Intrusion Detection

In ShellSwap we detect attacks triggering software vulnerabilities and capture exploits by enforcing a set of taint-based security policies during dynamic symbolic tracing. Traditionally, taint tracking implemented on dynamic binary instrumentation frameworks (e.g. Pin [22] and Valgrind [23]) is used to detect attacks during runtime, Xu et al. [32], Autograph[19], Vigilante [14], and Bouncer [13] are all reasonable choices. While those solutions are more performant than symbolic tracing, ShellSwap cannot use them as they do not record path constraints, which are vital to our approach.

C. Manual Ricochet Attacks in the Wild

Ricochet attacks are widely adopted in competitive attack-defense contests today. The CTF team Shellphish has stated at DEF CON:

"Stealing and replaying exploits has become very popular; basically, it is the main way in which most teams attack others these days. I think that, during the last DEF CON, a majority of our flags (aka points) were coming from running 'stolen' exploits."

The CTF team PPP has also stated they inspected network traffic to find new vulnerabilities, which helped them score points and win DEF CON CTF in 2013 and 2014.

However, while the concept of ricochet attacks is well-known within the hacking-competition community [2, 4], it does not appear to have received much direct attention elsewhere. To the best of the knowledge, our system is the first end-to-end automatic ricochet attack generation system.

IX. Conclusion

In this paper, we introduce the automatic shellcode transplanting problem. Given a program, an exploit and a piece of shellcode, this problem asks how to automatically generate a new exploit that targets the potentially unknown vulnerability present in the program and executes the given shellcode.

We also propose ShellSwap, which is the system for automatic shellcode transplant for remote exploits. To our best knowledge, the ShellSwap system is the first automatic system that generally apply different shellcode on the exploits for unknown vulnerabilities. In our experiment, we evaluated the ShellSwap system on a combination of 20 exploits and 5 pieces of shellcode that are independently developed and different from the original exploit. Among the 100 test cases, our ShellSwap system successfully generated 88% of the exploits. Our results imply that exploit generation no longer requires delicate exploit skills. For those victims who are not familiar with exploit knowledge, they can also generate their exploits.

X. Acknowledgements

The authors would like to thank our shepherd, Gang Tan, and the reviewers for the valuable suggestions. We would also like to thank Alexandre Rebert, Matthew Maurer and Vyas Sekar for their comments on the project.

References

[1] QEMU. http://www.qemu-project.org/.

[2] The NOPSRUS Team @ DEF CON CTF 2007. http://nopsr.us/ctf2007/.

[3] Tracer. https://github.com/angr/tracer.

[4] We are Samurai CTF and we won Defcon CTF this year. AMA! http://www.reddit.com/r/netsec/comments/y0nnu/we_are_samurai_ctf_and_we_won_defcon_ctf_this/c5r9osm, 2013.

[5] CGC Final Event File Archive. https://repo.cybergrandchallenge.com/CFE/, 2016.

[6] The CGC Repository. https://github.com/CyberGrandChallenge, 2016.

[7] T. Avgerinos, S. K. Cha, B. T. H. Lim, and D. Brumley. AEG: Automatic Exploit Generation. In *Proceedings of 18th Annual Network and Distributed System Security Symposium*. Internet Society, 2011.

[8] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the International Conference on Software Engineering*, pages 1083–1094, New York, New York, USA, 2014. ACM Press.

[9] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. ByteWeight: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Security Symposium*, pages 845–860. USENIX, 2014.

[10] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Theory and Techniques for Automatic Generation of Vulnerability-Based Signatures. *IEEE Transactions on Dependable and Secure Computing*, 5(4):224–241, 2008.

[11] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 143–157. IEEE, 2008.

[12] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 380–394. IEEE, 2012.

[13] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing Software by Blocking Bad Input. In *Proceedings of 21st Symposium on Operating Systems Principles*, pages 117–130. ACM, 2007.

[14] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 133–147. ACM, 2005.

[15] CVE Details. Vulnerabilities By Type. https://www.cvedetails.com/vulnerabilities-by-types.php, 2017.

[16] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, N. Stefan, and A.-R. Sadeghi. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium*. The Internet Society, 2012.

[17] S. Heelan. Automatic generation of control flow hijacking exploits for software vulnerabilities, 2009.

[18] W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong, and J. Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. *Proceedings of the Virtual Execution Environments*, pages 2–12, 2006.

[19] H.-A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proceedings of the 13th USENIX Security Symposium*, pages 271–286. USENIX, 2004.

[20] Z. Lin, X. Zhang, and D. Xu. Convicting Exploitable Software Vulnerabilities: An Efficient Input Provenance Based Approach. In *IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 247–256. IEEE, 2008.

[21] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman. Protecting Against

Unexpected System Calls. In *Proceedings of the 14th USENIX Security Symposium*, pages 239–254. USENIX, 2005.

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Programming Language Design and Implementation*, pages 190–200. ACM, 2005.

[23] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM Conference on Programming Language Design and Implementation*, pages 89–100. ACM, 2007.

[24] N. A. Quynh. Unicorn - The ultimate CPU emulator. http://www.unicorn-engine.org/.

[25] A. Romano and D. Engler. Expression Reduction from Programs in a Symbolic Binary Executor. In *Proceedings of the 20th International Symposium Model Checking Software*, pages 301–319. Springer, 2013.

[26] F. B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[27] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit Hardening Made Easy. In *Proceedings of the USENIX Security Symposium*, pages 379–394, 2011.

[28] K. Sen, G. Necula, L. Gong, and W. Choi. MultiSE: Multi-Path Symbolic Execution using Value Summaries. In *Joint Meeting on Foundations of Software Engineering*, pages 842–853, New York, New York, USA, 2015. ACM Press.

[29] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, pages 138–157. IEEE, 2016.

[30] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[31] D. Wheeler. How to detect the next Heartbleed. http://www.dwheeler.com/essays/heartbleed.html#valgrind-confirmed, 2014.

[32] W. Xu, S. Bhatkar, and S. Brook. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *15th USENIX Security Symposium*, pages 121–136, 2004.