Automated Reflection of CTF Hostile Exploits (ARCHES)

Inductive Programming Techniques for Network Traffic Comprehension and Reflection

by

Zackary Crosley

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Masters of Science

Approved June 2019 by the
Graduate Supervisory Committee:

Adam Doupe, Chair
Yan Shoshitaishvili
Ruoyu Wang

ARIZONA STATE UNIVERSITY

August 2019

ABSTRACT

As the gap widens between the number of security threats and the number of security professionals, the need for automated security tools becomes increasingly important. These automated systems assist security professionals by identifying and/or fixing potential vulnerabilities before they can be exploited. One such category of tools is exploit generators, which craft exploits to demonstrate a vulnerability and provide guidance on how to repair it. Existing exploit generators largely use the application code, either through static or dynamic analysis, to locate crashes and craft a payload.

This thesis proposes the Automated Reflection of CTF Hostile Exploits (ARCHES), an exploit generator that learns by example. ARCHES uses an inductive programming library named IRE to generate exploits from exploit examples. In doing so, ARCHES can create an exploit only from example exploit payloads without interacting with the service. By representing each component of the exploit interaction as a collection of theories for how that component occurs, ARCHES can identify critical state information and replicate an executable exploit. This methodology learns rapidly and works with only a few examples. The ARCHES exploit generator is targeted towards Capture the Flag (CTF) events as a suitable environment for initial research.

The effectiveness of this methodology was evaluated on four exploits with features that demonstrate the capabilities and limitations of this methodology. ARCHES is capable of reproducing exploits that require an understanding of state dependent input, such as a flag id. Additionally, ARCHES can handle basic utilization of state information that is revealed through service output. However, limitations in this methodology result in failure to replicate exploits that require a loop, intricate mathematics, or multiple TCP connections.

Inductive programming has potential as a security tool to augment existing automated security tools. Future research into these techniques will provide more capabilities for security professionals in academia and in industry.

ACKNOWLEDGMENTS

I would like to thank Adam, Yan, Fish, and Tiffany for creating an environment that pushed me to escape my comfort zone and learn as much as I could with my time at ASU. It has been one of the most challenging and rewarding experiences of my life, and I will miss it dearly. Thank you to everyone in SEFCOM for their assistance with numerous coding issues, fascinating discussions, and general support. I am grateful to have met each of you.

Special thanks to my family and friends outside of school for listening patiently to my unintelligible babblings on my research for the last year. Hopefully this document will serve to put all those conversations in context, should you ever subject yourself to reading it.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

PROBLEM OVERVIEW

1.1 Introduction

The creation of automated systems capable of performing laborious tasks have become a major focus for the security field. This push is the result of many factors, but possibly the largest is the growing number of cyber security threats and a lack of trained professionals. Two thirds of global businesses and government departments indicated that they had insufficient trained security personnel for their needs [21]. About half of these groups in the global report indicated that an inability to find qualified employees in the field is a cause of the deficit. The same report found that the deficit in US Federal departments was attributed seventy percent to a shortage of qualified workers [22]. By 2020, the global shortage of security professionals is estimated to be 1.8 million. Meanwhile the number of cyber crimes commited continues to rise steadily. In 2017 over three hundred thousand cyber crimes were reported to the FBI, totalling one and a half billion dollars in damages [9]. One reason for this increase in attacks is simply opportunity. The introduction of web connected systems greatly widens the risk of exploitation by malicious parties. An air-gapped system only has a few potential locations from which it can be exploited. An attacker could always directly access the machine, but this risk is insignificant compared to the billions of devices an internet device can communicate with at any moment. An attacker can, from the comfort of his or her own home and through the obfuscation of an anonymizing network, execute attacks with limited risk and on any vulnerable

target. As the gap between security professionals and the number of attacks continues to widen, automated systems that make security available to more people at a lower cost will be one of the core ways to hold of attackers.

One such area of research revolves around the automatic comprehension of vulnerabilities. Such a system could turn a flagged interaction into an actionable security report. In an advanced system, a patch for this vulnerability could be produced with minimal or no human involvement. For large systems like Facebook or Google that handle billions of requests per day [20], this would augment the human workforce's ability to fix vulnerabilities faster, giving less opportunities to attackers. There are many leaps in technology that must be made to enable such a future. In mountains of data, how do we efficiently locate traffic of interest? While Denial of Service and other loud offenses can be easily identified due to the volume of traffic, other vulnerabilities such as Cross-Site Scripting (XSS) are far quieter. Once we have traffic of interest, how do we use that to find a vulnerability? How can we consistently fix an identified vulnerability? These questions are too broad for one paper or researcher, but incremental steps to answer these questions help us secure the devices that run critical systems around the world.

One of the other challenges with research in this area is testing and proving system competence. No large company is going to try security tools that have not been proven on their large scale operations. Thus, researchers have to develop proxies by which we can experiment without the risks and barriers that real world deployment carries. One venue which enables fast experimentation of new security research is a Capture the Flag (CTF) event. Capture the Flag is a competition between cyber security professions as an educational tool for practicing security techniques [13]. CTFs can be organized in many different ways, but all variations provide challenges which simulate

some facet of real world cybersecurity. Players solving the challenge get to practice security techniques or try new methodologies that are still in development and not ready for widespread use. In a CTF game, we can test new ways to answer questions about identifying traffic; fixing vulnerabilities; and, critically for this paper, finding and exploiting vulnerabilities.

In this thesis, we present a novel methodology for reflecting exploits in a Capture the Flag environment. We introduce the ARCHES system which employs reflection to play a CTF without human intervention. ARCHES interfaces with the CTF system to identify services, sniffs traffic on each service, and applies inductive programming techniques to learn exploits from the observed traffic. These exploits are then launched at opposing teams and the resulting flags submitted to the CTF manager. We additionally introduce a heuristic for grouping traffic by exploit that is source agnostic to improve the learning rate. In the short term, ARCHES is a tool which CTF players can use to score extra points. In the long term, ARCHES introduces a way to recreate and represent exploits that can be used by automated systems for comprehension and further processing.

1.2 Related Work

Recently there has been a plethora of research on automatic security systems, including automatic exploitation systems. While most of these papers focus on exploit detection, there is a growing body of work on exploit generation due to its potential for penetration testing. Programs like BurpSuite already use exploit generation techniques to identify bugs for web application administrators [17]. Recent research has advanced these techniques using strategies like artificial intelligence and program synthesis.

One such example of this research was born out of the Cyber Grand Challenge (CGC) organized by the Defense Advanced Research Project Agency (DARPA). The CGC challenged researchers to build systems that could identify vulnerabilities in binaries and develop them into exploits and patches. Two major systems that were developed by competing teams as part of this challenge were Mayhem, produced by For All Secure, and MechanicalPhish, by Shellphish. Mayhem earned For All Secure the first place prize, while MechanicalPhish won third. Both teams combined fuzzing and symbolic execution to locate crashes for possible exploits [1]. The systems use the crash and a constraint solver to discover the input properties that lead to the crash. The constraint solver used both random mutation of bytes and symbolic analysis techniques with a Satisfiability Modulo Theories (SMT) solver to develop a theory of how the input affects the program to cause the crash. With this knowledge the program can create a malicious payload from the patch, possibly allowing control over the Instruction Pointer (IP) or performing arbitrary reads or writes to memory. The system can then use these payloads to prove to the server that a valid exploit has been obtained. Mechanical Phish additionally used network packets as seeds for the fuzzer to speed up the process of finding crashes [19]. In this way, the system uses opposing teams' exploits to try to enhance its own analysis. [1]. Other implementations of these techniques supported arbitrary reads with greater range and showed that similar constraint solving methods, along with some efficiency improvements, could yield exploits under more conditions [16]. The symbolic execution techniques used by MechanicalPhish and Mayhem were previously used to locate vulnerabilities in several real-world applications, including Microsoft Word and Unrar [16, 15]. The exploits

[1]Unfortunately this component failed, preventing the Shellphish team from reaping any reward from it.

discovered included stack/heap overflow, buffer overflow, use of uninitialized variable, and format string vulnerabilities. Similar approaches were used by researchers at CMU to create buffer overflow exploits [2] [2].

Other research in this area focused on the generation of exploits not from other exploits or the source code itself, but from the patches to the binary [5]. This technique first uses a standard file diff to discover the changes made in a binary from a patch. The system then uses constraint satisfaction to try to determine what inputs were permitted in the old binary but are stopped by the check in the new binary. All inputs which fit this description are determined to be possible vulnerabilities. This constraint is then used to create an exploit which could potentially hit any vulnerabilities present in the old version of the system. The researchers suggest patches be released in as short of a time window as possible to prevent individuals from developing exploits for the older version. Additionally, they recommend obfuscation techniques to make these attacks less successful.

Another technique, called data flow stitching, enables programs to develop exploits based on the data flow [14]. This technique produces a two dimensional data flow graph (2D-DFG) which tracks variables and how the data moves between them. The program then identifies values of interest such as private keys and identifies inputs that cause the values to become accessible in some form. This allows the attacker to obtain private data. Using this technique on real world software, the researchers were able to find several new vulnerabilities in addition to replicating known vulnerabilities. Another piece of dataflow research was used to identify insufficient sanitization that resulted in XSS or SQL injection vulnerabilities in Java web apps [8].

[2]This program is also named Mayhem, though it appears unrelated to the program produced by ForAllSecure

Return Oriented Programming (ROP) is an exploit technique that has become more popular due to its ability to bypass traditional security measures. Recent research has found automatic ways to generate ROP chains that also defeat Address Space Layout Randomization (ASLR) [18]. The resulting program automatically generated ROP payloads for a given binary and modify those payloads to beat W$\oplus$X and ASLR. This program was able to exploit nine real world programs with these defenses enabled.

Machine learning has been applied to identify novel SQL injections and XSS attacks [24]. Researchers used machine learning on several properties of the exploit, such as string length and character distribution (among others), and create profiles based on these properties. Inputs that do not match the profiles of normal traffic are then inspected as potential exploits. This strategy yielded a low false positive rate and a high detection rate. Researchers have also leveraged machine learning to detect abnormal traffic using neural nets, which identified all normal traffic and correctly identified seventy five percent of malicious traffic [7].

Intelligent analysis techniques have also been implemented to try to fingerprint exploits automatically by vulnerability, permitting faster responses to zero-day exploits. An ideal signature describes the exploit abstractly enough to apply to all derivative exploits, identifying when the exploit is slightly modified and reused. This vulnerability signature acts as a conditional that is satisfied only for payloads which are exploits for the associated vulnerability [6]. These signatures can be specified as turing machines, symbolic constraints, or regular expressions.

Assisted exploit generation, either through interesting inputs or vulnerabilities, is another interesting topic of research. These systems are often able to generate more complex exploits, or generate exploits faster, because of the additional data. PolyAEG uses a semi-automated approach to find vulnerabilities and generate exploits. Given

one input set identified as producing abnormal output, the system can use a variety of different trampoline instructions to generate many different exploits [25]. In this way the program can produce polymorphic exploits. The researchers were able to find thousands of different possible exploits in a popular image viewer. Other research uses an existing shellcode exploit and intelligently swaps out the shellcode for a new payload, without the restriction of contiguous memory space [4].

Finally, some interesting research has been done on the implications of automatic exploit generation. One such paper analyzed this topic using game theory to identify optimal strategies in cyber warfare. In her paper, Bao found that an environment of perfect exploit reflection results in more vulnerabilities being disclosed [3]. hen exploits can be reflected, the incentive to use them goes down as some of the would-be attacker's systems are likely vulnerable. This additionally serves as a motivation for this research, as our methodology improves this exploit reflection capability and hopefully leads to a corresponding increase in vulnerability disclosures.

Chapter 2

BACKGROUND

2.1 Attack and Defense CTF

Capture the Flag is a cyber security competition and educational tool for professionals and students. The environment allows for the safe practice of security techniques away from any critical systems. Multiple teams compete against one another to complete challenges put forth by the competition organizers. Completing a challenge earns that team a flag, which can be submitted for points. There are a variety of setups for cyber security competitions, including Jeopardy (Red Team), Defense (Blue Team), and Attack and Defense. All of these competitions help security professionals practice their security skills so they can patch more vulnerabilities in real world systems.

Jeopardy style CTFs focus exclusively on the attack side of cyber security. Competitors are given vulnerable software which they need to exploit. Exploiting this software gives a team a one time flag which they redeem for points. Often the number of points a challenge yields is a function of the number of solves it has and the challenge difficulty, though this varies from competition to competition. Conversely, defense style competitions have teams defend vulnerable services against a single attacking red team. Participating teams only gain points from preventing their services from being exploited, with no point incentive to produce exploits.

Attack and Defense CTFs feature elements of both jeopardy and defense. Each team has a server with identical, vulnerable services. Players identify vulnerabilities

in these services and have the opportunity to patch their own service and/or exploit other teams services. Each flag in an Attack and Defense CTF have an associated id, which are used in the exploit to grab the flag. In a real world context, a flag id could map onto a username of interest on a particular system with the flag representing the user's password. The flag id allows the system to give different teams unique flags and more easily manage the game. The game consists of many rounds, during each of which a team is awarded points based on their performance. The scoring of this varies drastically, but usually teams are rewarded points by exploiting opponents and lose points by being exploited. The exact scoring system of the game and the architecture of the setup create significant strategy opportunities. In some games, patches to a service are visible to all teams and thus provide information about the vulnerability they fix. In others, network traffic coming into each service is visible for teams to identify exploits from. This latter case is the scenario in which ARCHES can provide an advantage.

In an Attack and Defense CTF with visible network traffic, teams can see incoming exploits to their services. Since each team is originally running identical services, an exploit for a service is likely to be successful against multiple opponents. This provides a huge incentive to reverse engineer exploits from incoming traffic. ARCHES can automatically observe this incoming traffic to each service and attempt to craft exploits from the traffic. If successful, these exploits will then be used to obtain flags. While the original source of the observed exploit may have been a team which already patched their service, the fact they are still sending the exploit implies it is likely still effective against some teams. Consequently, reflecting this derived exploit against all teams is likely to exploit some team which has not yet patched that vulnerability, earning points even without specific knowledge of the explot used.

2.2 Inductive Programming

Inductive programming is a method of contraint-based learning which learns program specifications based on incomplete information, such as input/output pairs [10]. For each part of the output, the program generates a set of all possible ways the output can be formed from the input. The system is provided with a collection of valid transformations on the data, henceforth referred to as *theories*. A developer can specify any transformation of interest to the problem domain, but common theories include constant strings that insert some fixed sequence or substrings on relevant domain data. For instance, a single byte may always equal some value (Constant theory) or may be the first character of an input variable (Substring theory). This creates a directed acyclic (multi)graph (DAG) where the edges between two nodes describe all the possible ways for this segment to be derived. The program performs an intersection on many such DAGs together to obtain the common DAG that could fit all the data. With sufficient examples, this reduces to a single edge for each segment that could produce this behavior. In this way, the program is able to replicate the source program.

One of the main advantages of inductive programming over other techniques of learning is the small number of examples required. By constraining the learning to only the theories provided at run time, the system only has to consider a small number of transformations. Simple applications can be learned in just a few examples, as opposed to the millions of examples often given to big data classifiers. This makes inductive programming an ideal candidate for program synthesis, where the rules that are permitted are clear and input/output pairs are often the only data available.

An example of an implementation using inductive programming in popular software

can be found in the latest version of Microsoft Excel. Since Excel 2013, the program has included a FlashFill feature that writes macros for users based on entered examples [11]. FlashFill uses a DAG to represent individual characters of the output and, with each example, performs a graph intersection to include only those values for each character which are possible in every example. The system integrates rules for constant characters, substrings of inputs, and concatenation of other rules. This system operates on every column of data with respect to every other column. If after some number of examples the DAG reaches a single rule for every character in the output, the system will prompt users with the automated macro entries and fill the rest of the rows. Gulwani has suggested several other applications for this technology, such as developing new math questions for education from existing examples [12]

Chapter 3

DESIGN OVERVIEW

3.1 Inductive Programming Exploits

To understand how this technique can be used for an exploit, one first must understand what an exploit is. Fundamentally, an exploit is just a series of bytes sent in a specific interaction with the server. This means there are two essential components for an exploit: the program state and the input to the program. That is, we must send specific bytes to the service based on the program state. Our input, being a series of bytes, is just a string. A dumb reflection of an exploit could be easily performed by sending an opposing services the exact inputs that one receives, byte for byte. However this will very rarely produce meaningful results, as this methodology lacks awareness of the program state. The bytes sent to the service may not produce the desired results for the state the service is in. Additionally, the exploit input will often depend on the program state, meaning the input will have to be modified for each exploit. In order to successfully reflect an exploit, then, a program will have to perform two functions. First, the program must capture input and identify which bytes are constant in the interaction and which change. Secondly, the program must understand the underlying purpose behind the changing values in the exploit. These two items can both be solved using the inductive programming techniques discussed in section 2.2.

Suppose our program has received two different interactions we know are for the same exploit. We will discuss later how interactions are identified as belonging

to a specific exploit, but for now this capability will be assumed. The inductive programming technique will represent each byte of the inputs as a series of possible theories, or methods by which that byte could be obtained. Specifically, we will represent the input as a DAG with each byte being a set of edges for possible theories. The simplest of these theories is a constant. That is, this byte will always equal this value. Note that with just one example, a constant theory will apply to every byte. By performing a intersection of the DAGs for our two interactions, we can locate the areas where a constant byte explanation doesn't work. With many more such examples, we can find all the bytes that are not some constant value. We can then identify if these values are determined from the state in a meaningful way. The possible theories for these non-constant values are domain dependent. A program may have a gate using the sum of two numbers (i.e. 4 + 12) to which the program must reply with the answer to continue. Practices like this are common in CTFs to add complexity to the exploit script. If we read these values into our state, however, then the system can learn how those two variables are used to produce the response. To accomplish this, the learning agent must not only consider the known state of the target program but also its output values. These outputs provide new information regarding the target program state, which are necessary to create a generalized exploit.

For our purposes, we are primarily interesting in locating the flag id and flag in this exploit sequence. If we know where the flag id is located, then we can place our own flag ids in this location to repeat the exploit and retrieve the flag. Note that the flag id will always be provided as an input to the system, and the flag will always be an output. Having access to the game server, we can directly observe information on the state of the game. Unfortunately, we do not know the flag ids for the other team, preventing us from identifying flag ids directly. In general, the series of input

bytes for which no theory seems to fit will either be the flag id or determined though theories on the state information. This methodology will work so long as sufficient theories are included to find all other relationships with the state, which would leave only a single series of unknown input bytes. A series of variable output bytes will either be the flag or some other state information. With a flag format, we can identify the items which are a flag. Without a flag format, we can simply attempt to submit all such state values to determine which is the flag.

Consider the interactions from one of our test programs, Tweety Bird, in figure 1. Provided the flag id and a sufficient amount of filler characters (60 to be exact, though it is cut off after 10 in the example for space reasons) one receives the flag. This exploit presents a few basic challenges for reflection that a simple system wouldn't decipher. The arbitrary number printed out indicating the user count means there is an additional non-constant string. The exploit requires a sequence of interactions with the system, rather than a single string to send. Finally, the flag and flag id are contained in a message with other characters, requiring the system to identify them from the surrounding bytes. Figure 2 shows how the inductive programming techniques will turn these interactions into a set of DAGs, at a high level. Note that the real representation would not have so few edges with so many bytes per edge, but it makes for an easier visual. Their intersection forms the structure seen in Figure 3. By analyzing two and intersecting, inductive programming has identified three fields of interest. Only one of these fields was sent to the system, so it must be our flag id. However we have two random sequences being returned to us during the exploit. If the flag conforms to a format, we can determine the last item to be the flag. Without, we can identify the first item cannot be the flag since it occurs before the flag id is sent. The first can be added to our state, but with more examples we find it isn't used

```
OUTPUT Twee! Welcome to the
   ↪ secret Twitter, Tweety Bird.
   ↪  You are the 4293997680'th
   ↪ user today, only 4 more to
   ↪ overtake Facebook. Want to
   ↪ (R)ead or (W)rite a twit?
INPUT R
OUTPUT Please type: twit_id
   ↪ password
INPUT 1893037211 AAAAAAAAAA...
OUTPUT Note content: dLzNUQwPPG
```

(a) Tweety Bird Interaction One

```
OUTPUT Twee! Welcome to the
   ↪ secret Twitter, Tweety Bird.
   ↪  You are the 4289672832'th
   ↪ user today, only 4 more to
   ↪ overtake Facebook. Want to
   ↪ (R)ead or (W)rite a twit?
INPUT R
OUTPUT Please type: twit_id
   ↪ password
INPUT 2607290646 AAAAAAAAAAA...
OUTPUT Note content: XFjONDvPDG
```

(b) Tweety Bird Interaction Two

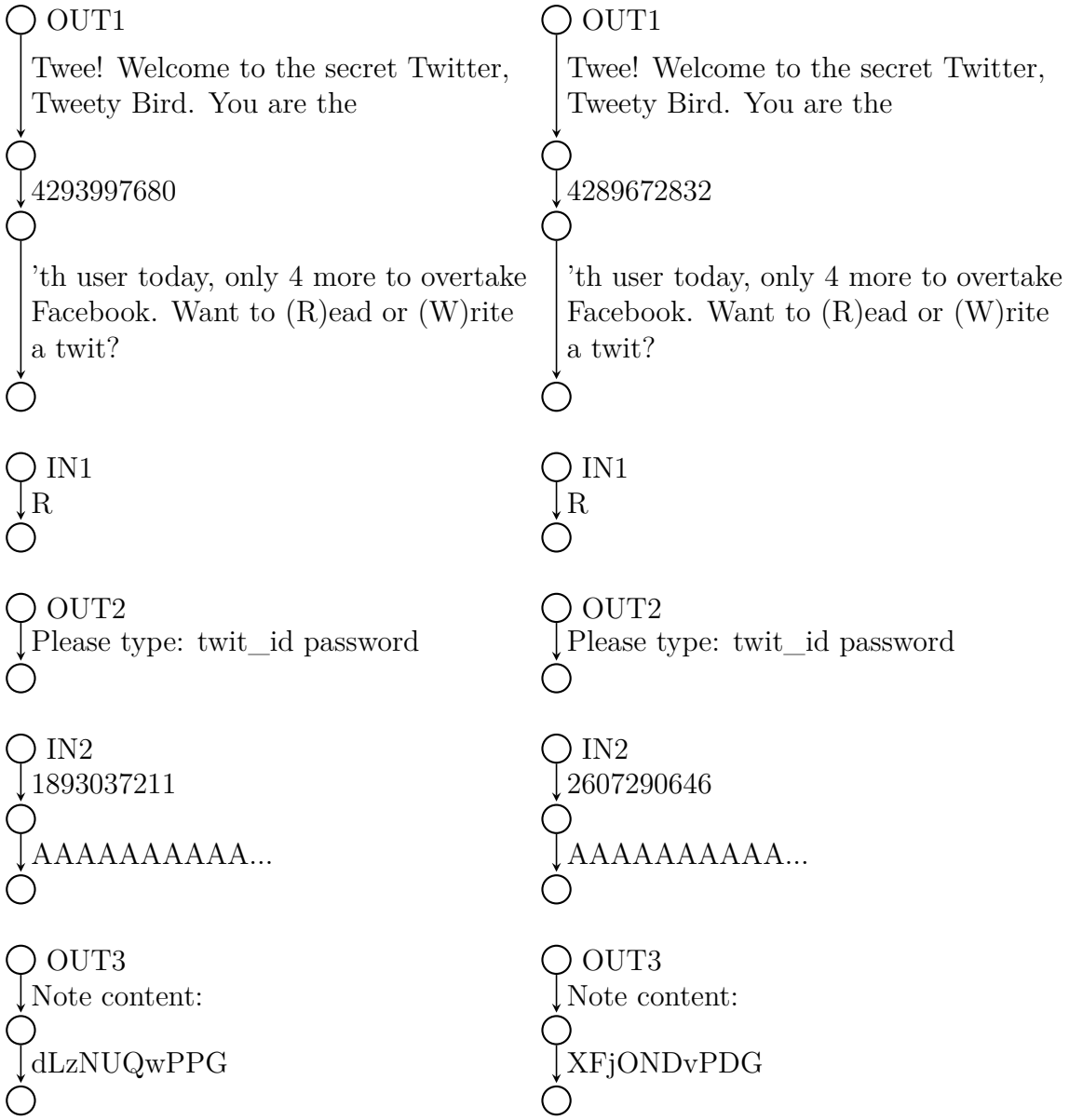
Figure 1: Tweety Bird Interaction Examples

anywhere else in the exploit. The system will learn this first item is to be consumed and ignored. Thus we have learned the structure of this interaction.

This learned DAG can be used to create a new exploit. Provided a flag id, we begin an interaction consuming the first message as constants and a variable with no further uses. The system then sends the constant R it has learned and receives back the constant request for a password. The system then sends the flag id and the sequence of constant As it has learned. Finally, the system receives the final output and pulls from it the variable corresponding to the flag. This flag can now be submitted for points.

3.2 Reflection Process

To enable the reflection of exploits there are more components that the learning module depends on. Our system is designed to be a complete exploit reflector, and



(a) Tweety Bird DAG Interaction One

(b) Tweety Bird Interaction Two

Figure 2: Tweety Bird Interaction DAGs

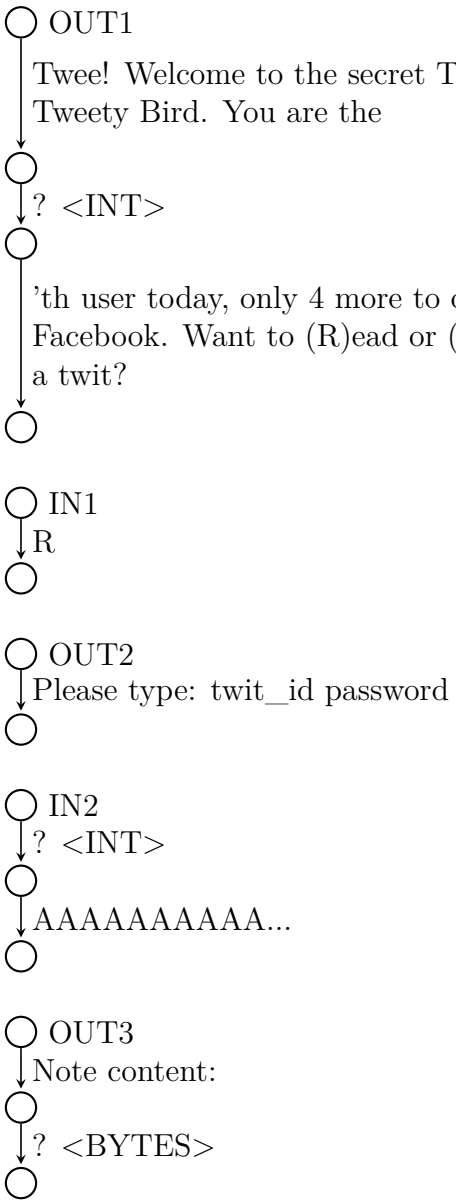


Figure 3: Tweety Bird Intersected DAGS

thus requires significant infrastucture on top of the inductive programming techniques discussed. While the next section discusses implementation details, this section will simply discuss these components and their purpose.

1. Interfacing with CTF Game Engine

   In order for the exploit reflection to be useful in a CTF, it must be able to score points. This requires interfacing with the game engine to get flag ids and submit flags. Interfacing with the game engine also informs the program of target services and onese own services, so they can be observed for traffic. As such, this is the very first part of the reflection process.

2. Sniffing CTF Host Traffic

   To intercept exploits, the system has to observe the incoming traffic to a copy of each of the services. Each participant in an attack and defense has a copy of the services and access to the network traffic, either through direct access to the host or through packets being provided through another interface. Whatever method is used, sniffing traffic is the data collection phase of the system exploit learning.

3. Deriving Conversations from Traffic

   Individual packets, in general, have no meaning to the system or to an exploit itself. Exploits, as described earlier, depend on the program state in addition to the bytes being sent. The packets only make sense in the sequence they occured in a given conversation with another host. The system recreates the sequential interactions with another host for each conversation that occured, which will then be used for learning.

4. Fingerprinting Conversations as Exploits

   It is not sufficient to simply group by conversations and learn from these alone.

The inductive programming methodology wants labeled data, in the sense that we need to know what exploit a conversation belongs to. Once these conversations are grouped, they can be learned on together to derive an exploit.

5. IRE Learning

   With the data collected and categorized, our inductive programming library IRE can learn from the exploits to derive functional copies. The system will attempt to learn from each exploit, throwing out those which it can't and creating runnable exploits for those it can. This is the exploit generation phase.

6. Launching Exploits

   With a valid exploit learned, the system uses the game engine to gather flag ids and launch exploits at all targets.

Chapter 4

IMPLEMENTATION

The Automatic Replication of CTF Hostile Exploits (ARCHES) system is organized as a collection of components. These components interact with a scheduler program, SLACRS, to handle communications and process flow[3]. ARCHES has greater modularity and interoperability as a result of this configuration; updates to components are independent of other components and the addition of new components into the chain straight forward. The integration with SLACRS also enables ARCHES to be easily integrated with other services running on the same SLACRS instance, permitting a future where ARCHES is just one of several automated systems working in concert.

ARCHES is made up of six[4] individual components that data flows between in a linear fashion. These components correspond, roughly, to the process components laid out in section 3.2. The following subsections will discuss these components at length. At a high level, ARCHES facilitates data flow from the Service Reader, Sniffing Component, Conversation Component, IRE Component, Exploit Launcher, and finally to the Flag Submitter. This architecture can be seen clearly in Figure 4.

[3]SLACRS is developed at the Security Engineering for Future Computing (SEFCOM) lab at ASU.

[4]The extra component in Figure 4, Input Component, is only used to kick off the SLACRS scheduler and is not important to the overall functionality.

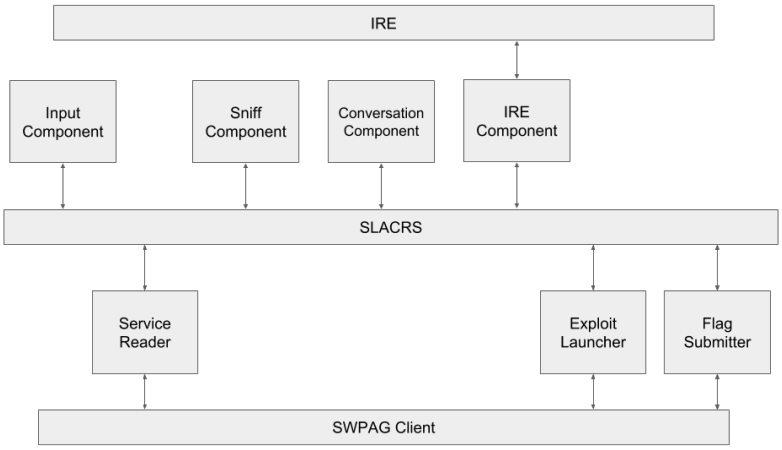


Figure 4: System Architecture

4.1 IRE

The Inductive Reverse Engineering (IRE) library is the core of the exploit reflection capabilities of ARCHES. IRE implements in Python a generalized inductive programming capabilities to learn from input-output pairs. Exploits, as discussed in section 3.1, are themselves just a series of byte sequences and thus can be learned using the same methodology as strings. IRE has several features that made it convenient for this usecase. Most importantly, differential program analysis allows IRE to use input-output samples to infer a grammar to improve its learning efficiency. Many inductive programming tools require a grammar specifying valid orders of theories to be defined. For generic exploit learning there is no single grammar we can hard code to improve our state searches. Traditional inductive techniques without a grammar could result in a state explosion that would make ARCHES prohibitively inefficient. By using IRE, the differences between the exploit instances can be analyzed to form a pseudogrammar and identify the areas which change to focus learning. Additionally, the fact that IRE is to be open sourced made it a dependable base on which to build.

Furthermore, IRE is written in Python3 and is thereby natively compatible with ARCHES. Finally, IRE provides a simple framework for adding on additional theories to improve ARCHES learning capabilities over time.

IRE is currently in its infancy, but already includes many theories that are critical to ARCHES functionality. ARCHES primarily uses just four theories to learn: constant, capture, function, and concat. The constant theory learns a constant string to consume (data from target program) or output (data to target program). This is incredibly important to ARCHES, since most exploits involve interacting with interfaces in some consistent way. For instance, in one of our sample programs the exploit must send a letter R to get to the read state with the vulnerability. The constant theory is how IRE is able to learn this consistent output. The capture theory identifies changing values being sent from the service and identifies them as potentially important to understanding the program state. These values are captured into IRE's representation of the program state. These values may be critical to the exploit itself, such as a basic math query $X + Y$ that must be solved to continue, or may be the flag being emitted. Note that while IRE does not currently provide the math operations for solving a query like $X + Y$, this capability could be added like any other theory. The function theory allows the variables in IRE's representation of the state to be used in the program. This allows IRE to place developer-provided values or the values from the capture theory to be identified. This theory is used in ARCHES to place the flag id in the appropriate location of the payload. Finally, the concat theory allows the output of several theories to be strung together. This allows for IRE to learn how to compose theories together to produce a line of output. A buffer overflow, for instance, may consist of a long string of constant bytes (such as A characters) followed by the flag id.

The concat theory can identify the constant and function theories this requires and then combine them together with a concat theory to produce the complete output.

As IRE continues to undergo development, with more theories or new functionality, ARCHES will be able to learn increasingly complicated exploits.

4.2 Service Reader

The Service Reader interfaces with the game client to identify the services that are available and their network locations. To make the program as versatile as possible, the goal was to isolate interactions with the game client itself to as few locations as possible. This will allow the game client to be easily exchanged for different CTF competitions. By performing this action in its own component, the sniffing, conversation, and IRE components are all independent of the game client, preventing them from unnecessary modifications for compatibility. Initially, ARCHES has been built to interact with the Shall We Play A Game (SWPAG) CTF Client [23]. This is a popular client used in CTFs such as iCTF, and thus is a good candidate for initial support.

4.3 Sniffing Component

To recreate the exploits for a service we must intercept and learn from the incoming traffic to that service. It was important to do this in as generic a way as possible. This implementation assumes only the presence of *tcpdump* on the host, as that is standard on most systems. As a consequence, this component requires the user for which the program has host access has *tcpdump* permissions. Futhermore, it was important to

enable ARCHES to handle traffic sourced from locations other than direct sniffing. Some CTFs do not provide access to the host box, but provide traffic via some other interface. ARCHES should handle this with minimal modifcation. Thus this sniffing component gathers traffic into PCAP files and emits those to the next component. Since PCAP files are generic and can be easily made out of any traffic format, it enables the conversation components to take a constant item that will be simple for a component to create. This way, only the sniffing component will have to be swapped out for a different traffic source. The sniffing component tracks the PCAP files it has seen and pulls down new files that exist after a certain wait period. This does introduce the opportunity for conversations to be cut off, potentially corrupting some data. In a real scenario the wait time between these queries would be several CTF rounds, resulting in a small period, relatively speaking, during which conversations will be cut off. In the event this does occur, the fragmented conversations will be classified independently and will produce no valid exploit, ending any processing on them.

4.4 Conversation Component

Creating a conversation from the provided PCAP files is necessary formatting for the exploit learning process. To accomplish this, ARCHES uses the TCP data to identify messages that were involved in an interaction and their ordering. ARCHES does this by filtering out empty packets used only for TCP control, sorting by increasing SYN numbers, and grabbing the next packet in the sequence while they exist. Note that this intoduces a potential issue with learning exploits that require packet manipulations beyond the body of the packet, the resolution of which is left

to future research. Additionally, ARCHES removes duplicate conversations that do not enhance learning but merely slow down the process. For compatibility purposes, ARCHES also separates conversation by lines. When data is sent to the service over SSH this will be done by line, and line-based separation ensures consistency throughout the ARCHES program.

4.5 IRE Component

This component receives a sequence of conversations from which to learn the exploit. IRE, however, requires the conversations to be grouped by exploit to learn from their differences. This introduces the problem of identifying what exploit a conversation belongs to without knowing what possible exploits look like. For this, ARCHES uses a heuristic to fingerprint exploits and group them together. Exploits are grouped using several features including the number of messages, the direction of the conversation, and strictest category of message content (numbers, ASCII, etc). This allows the same exploit from two different sources to be identified as related while minimizing the possible collision of two unrelated exploits. This is an imperfect heuristic, and future research will integrate other techniques to improve this [7]. The sorted conversations are then passed to the IRE module, which generates a symbolic representation of the conversation. This is encapsulated in a runnable exploit object that can be called with a target and flag id to perform the exploit on an arbitrary target.

4.5.1 Exploit Launcher

The exploits emitted from the learning component are callables that take the target and flag id. ARCHES queries the game client to identify the possible targets and calls the exploit on each of them. If the exploit fails, this exploit will be halted to make processing time available to another possible exploit. Once a valid exploit is identified, it will be run every tick of the CTF challenge with the new targets. Flags retrived from the exploit are emitted to the flag submitter.

4.5.2 Flag Submitter

Finally, ARCHES interfaces with the game engine to submit the flags it receives from the exploit component. The result of these submissions are also tracked for future use, such as strategy changes.

Chapter 5

RESULTS

5.1 Experiment Details

5.1.1 Experiment Setup

The testing framework developed for ARCHES performs end-to-end testing to ensure all components work as expected. The necessity of interfacing with the CTF game client and test services required significant testing architecture. First, the services which are to be exploited must be launched with an accessible interface. This host must also be able to be accessed via SSH and capable of running *tcpdump*. Second, the SWPAG client must be set up to locate these services as it would in a normal context. Exploit and flag setting scripts must be written for these services and run automatically to verify the output of the learned exploit is correct. Additionally, it was desirable that tests to run on any machine with limited setup.

To accomodate all these requirements, the testing framework uses Docker with a script to automatically generate containers from a template Dockerfile. A fixed directory is scanned for all service folders it contains. Each of these service folders has the same relative location for its exploit and set flag scripts, among other requirements. The test script creates a docker network to ensure control over container locations and enable interaction. Each service is copied into a container supporting *xinetd* services with an assigned IP. An *xinetd* configuration for the service is automatically generated and an unique port assigned to it. This port is opened for interacting with the service

as a standard remote host. One advantage of this setup is complete isolation of the services from one another, removing potential compatibility issues. However, the separation of these services prevents a single host from sniffing traffic to each service. The SWPAG client only allows for one IP address for all services, so without modifying the interface ARCHES cannot access each container separately. Futhermore, introducing SSH and *tcpdump* to each container makes them unnecessarily complex for their puposes. To resolve this, the test framework adds one more intermediate container to act as the host. This container supports SSH so it can be accessed like a host in a CTF. To sniff traffic, this intermediate container uses *socat* to forward traffic to corresponding services, so all exploits can be directed to this intermediate container to be observed. This has the side effect of viewing each interaction twice: one from the host to the intermediate container and once from the intermediate container to the service; however, this is filtered out in the conversation component and will not affect learning.

Once these containers are setup with ARCHES sniffing traffic, the exploits can be launched at each service automatically to begin the learning process. A mock CTF client emulating SWPAG was produced for testing purposes that uses the docker interface to retrieve server information and validate flags retrieved in tests. This copy has the same interface as the original client and so does not require any modifications to ARCHES except to the import statement.

5.1.2 Exploit Data Set

The exploit dataset consists of four services that test a range of capabilities. These exploits were selected to showcase the capabilities and limitations of ARCHES. Each

Binary	Success?	Reason for Failure
Tweety Bird	Yes	N/A
Dungeon	No	Reserved Byte for IRE
RASF	No	Looping, Math
No-RSA	No	Multiple TCP Sessions

Table 1: Summary of Results

of these experiments was run with just sixteen unique examples. As discussed in section 2.2, this small quantity is more than sufficient for this method of learning and the services tested here. The learning process itself only took a matter of seconds for each experiment, highlighting the efficiency of this methodology. A summary of these results can be found in Table 1.

5.1.2.1 Tweety Bird

The first service is a fairly simple exploit to showcase basic ARCHES capabilities. A complete interaction with this service can be seen in Figure 5. The variable values in the interaction have been placed in angle brackets. The interaction is pretty straightforward; the exploiter sending an R to get to the submenu where the flag id can be provided with filler bytes to produce the flag. However, this interaction outputs a variable value indicating the number of visits the service has received. For many learning systems, this would be challenging as it requires understanding this variable is not important—it does not influence the exploit in any way. The program learned from an interaction with IRE can be seen in Figure 6. We see on the second line that ARCHES has identified the variable number of visits as a capture theory, adding this information to its environment data on the program. Note that the *42* that the program has learned as a constant is because all the visit numbers this program

```
OUTPUT Twee! Welcome to the secret Twitter, Tweety Bird. You are the <
   ↪ ARBITRARY NUMBER>'th user today, only 4 more to overtake
   ↪ Facebook. Want to (R)ead or (W)rite a twit?
INPUT R
OUTPUT Please type: twit_id password
INPUT <FLAG ID>
   ↪ AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
OUTPUT Note content: <FLAG>
```

Figure 5: Tweety Bird Interaction

generates begin with 42. On line six of Figure 6 we observe that the program has learned to place the flag id followed by the constant number of A characters. Finally, on line seven we observe that ARCHES has learned to capture the output after *Note content:* into its environment. Note that ARCHES saves the flag under a random variable name as it doesn't have any way of knowing this is the flag, only that it is a value of interest. When launching the exploit on our test target, ARCHES successfully sends the payload and retrieves all service output, including the flag. ARCHES can then iterate over the service output to send the values to the CTF interface. This will submit both the flag and the random number indicating visits. Future work can use this feedback to further hone the learned program by identifying the flag and reduce erroneous submissions.

5.1.2.2 Dungeon

Dungeon is an exploit with one interesting feature to show some of the power of ARCHES. The service displays one of three ASCII planes which the script must name to continue. An example of one such output can be seen in Figure 7. We have seen

```
INBOUND <ConstantTheory constant='Twee! Welcome to the secret Twitter,
   ↪  Tweety Bird.'>
INBOUND <ConstantTheory constant='You are the 42'>, <CaptureTheory
   ↪ name=fpyqtbem>, <ConstantTheory constant="'th user today, only
   ↪ 4 more to overtake Facebook">
INBOUND <ConstantTheory constant='Want to (R)ead or (W)rite a twit?'>
OUTBOUND <ConstantTheory constant='R'>
INBOUND <ConstantTheory constant='Please type: twit_id password'>
OUTBOUND <FunctionTheory input='flag_id'> <ConstantTheory constant='
   ↪ AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'>
INBOUND <ConstantTheory constant='Note content: '>, <CaptureTheory
   ↪ name=ijkaapev>
```

Figure 6: Tweety Bird Learned

in the Tweety Bird exploit how ARCHES can identify changing values by capturing them and using them in future output. However, the names require a response which is not visibly predetermined by the previous value. So it would seem like this exploit shouldn't be able to be learned. Fortunately ARCHES is able to pick up on another important variance to create separate exploits: the length. Each ASCII picture uses a different number of lines in their representation. As a result, each one is learned as a unique exploit. Since each call of the exploit uses a random plane, no exploit execution is guaranteed to work. However, there is a probability for any exploit to work. Thus we can get flags, theoretically, by calling each exploit multiple times. Interesting as this example is, learning this exploit unfortunately fails for a different reason. In the payload on line 59 of Figure 7, the byte \xad is sent. IRE uses a longest common substring algorithm in its learning process that requires reserve bytes for tagging. IRE uses bytes \xa0 through \xaf for this tagging, which were selected as uncommon bytes to encounter. Unfortunately, this specific exploit requires one of these bytes and thus cannot be learned by IRE. Future IRE developments will hopefully remove this limitation, making this exploit learnable.

5.1.2.3 RASF

RASF is a much more complicated exploit. The interaction with this service can be seen, with some simplication, in Figure 8. This is an example of an exploit that ARCHES fails to learn, but not because of the large numbers of variables like one would expect. These ARCHES can learn to consume and add to the environment. This will slow down the learning a little bit due to the increased number of variables, but not significantly. The areas where this fails can be found on lines one and seventeen of Figure 8.

The first line RASF emits is a number. This indicates the number of times lines two through five of the exploit must be repeated. IRE currently has no concept of loops, and consequently neither does ARCHES. It does not realize that this repetition can be described as a single theory. Thus it views each exploit with unique length as independent, and thus fails to learn.

On line seventeen, the exploit requires a decision using the previously emitted values. The mathematical operation on these values is dependent on whether or not a specific value is zero. As discussed in the previous exploit, IRE has no concept of decision statements. Additionally, the IRE version ARCHES was built upon does not come with the mathematical theories needed for this. Unlike the previous two limitations discussed, math theories can be added and will be present in future IRE versions. Thus, the exploit cannot be learned at this time.

```
OUTPUT Welcome to my dungeon!
OUTPUT You can explore the dungeon with east,west,north,south
OUTPUT looking for treasures. But beware of the dangers!!
OUTPUT You have entered the first room!
OUTPUT I will now test your intellect
OUTPUT What is 2000+16
INPUT 2016
OUTPUT You guessed 2016
OUTPUT There is a passageway to the east
INPUT west
OUTPUT <ASCII PLANE>
OUTPUT Welcome to the Mitch\'s room!
OUTPUT You must identify the Aircraft flying overhead
INPUT f16
OUTPUT Right!
OUTPUT You have been given the golden saber
OUTPUT Where would you like to go next? East?
INPUT east
OUTPUT You have entered the first room!
OUTPUT Where would you like to go next? East?
OUTPUT Where would you like to go next? East?
OUTPUT You have entered the first room!
OUTPUT I will now test your intellect
OUTPUT What is 2000+16
INPUT 2016
OUTPUT You guessed 2016
OUTPUT There is a passageway to the east
INPUT west
OUTPUT You see a dwarf with a magic shield here.
OUTPUT I will give you my shield if you guess my number!
OUTPUT What is your guess?
INPUT 1
OUTPUT Sorry! Find your own shield
OUTPUT There is a passageway to the east and the west
INPUT east
OUTPUT Welcome to the storage room!
OUTPUT You may store a treasure by saying 'store flagid flag password'
OUTPUT e.g. store flag23 ThisIsMySecretFlag MySuperSecurePassword
OUTPUT You may retrieve a treasure by saying 'get flagid'
OUTPUT e.g. get flag23
OUTPUT will prompt for the password. If you get it right,
OUTPUT then print out ThisIsMySecretFlag
```

```
INPUT get flag
OUTPUT You said get
OUTPUT There is a passageway to the east and the west
INPUT east
OUTPUT You find a friendly gnome.
OUTPUT What is your name?
INPUT x%n
OUTPUT Hi!
OUTPUT x
OUTPUT There is a passageway to the east and the west
INPUT east
OUTPUT There's a dragon here!
OUTPUT There is a passageway to the west
INPUT kill dragon
OUTPUT You win!
OUTPUT What is your name?
INPUT \x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
   ↪ \x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
   ↪ x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
   ↪ x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
   ↪ x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
   ↪ x90\x90\x90\x90\x90\x90\x90\x90\x90\xef\xbe\xad\xde\xbd\x96\x04\
   ↪ x08\xbd\x96\x04\x08\xbd\x96\x04\x08\xbd\x96\x04\x08
OUTPUT Welcome to the special storage room!
OUTPUT You may retrieve a treasure by saying 'get flagid'
OUTPUT e.g. get flag23
OUTPUT It will then print out ThisIsMySecretFlag
INPUT get <FLAG ID>
OUTPUT The flag is: <FLAG>
```

Figure 7: Dungeon Interaction

5.1.2.4 No-RSA

No-RSA is a cyptography exploit based on bad encryption. Figure 9 shows the conversations that occur during this exploit. Note that unlike previous exploits, this one involves two separate TCP connections. ARCHES currently classifies exploits based on the signature as discussed in section 4.5. Using this methodology, ARCHES classifies

```
OUTPUT <NUMBER OF GAMES>
OUTPUT <ARBITRARY NUMBER 1>, <ARBITRARY NUMBER 2>, <ARBITRARY NUMBER
   ↪ 3>, <ARBITRARY NUMBER 4>, <ARBITRARY NUMBER 5>
INPUT <ARBITRARY NUMBER 6>
INPUT <ARBITRARY NUMBER 7>
INPUT <ARBITRARY NUMBER 8>
... Repeat lines 2 - 5
OUTPUT Entering Interactive Mode (2-core system)
OUTPUT 1: New Command\n2: Fetch Command\n3: Play Command\n4: Jack
   ↪ Command\n5: Exit\nOK, What do you want?:\n
INPUT 2
INPUT 3
INPUT <FLAG ID>
INPUT <ARBITRARY NUMBER 9>
OUTPUT 1
INPUT 1
OUTPUT <ARBITRARY NUMBER 10>
OUTPUT <ARBITRARY NUMBER 11>
INPUT <FUNCTION ON ARBITRARY NUMBER 10 AND 11>
INPUT <FUNCTION ON ARBITRARY NUMBER 10 AND 11>
OUTPUT <FLAG>
```

Figure 8: RASF Interaction

the two TCP connections as conversations for two separate exploits. On their own, neither of these are able to retrieve a flag. There is no system in place for ARCHES to affiliate these two as a single exploit. Additionally, the math involved with modifying the signature returned on line thirteen of Figure 9 is extensive, involving several operators chained on multiple state variables. Even once mathematical operators are added to IRE, learning this would be challenging. Further research will need to be done for ARCHES to learn exploits such as this.

```
OUTPUT Hi! Welcome to our note storage system. It's based on RSA!
OUTPUT What do you want to do?
OUTPUT 1. Write a Note -> Type W
OUTPUT 2. Reading a specific note? -> Type R
OUTPUT 3. Request for a signature of your integer? -> Type S
INPUT S
OUTPUT Here is our public key (you need them in order to verify the
    ↪ signature): N E
OUTPUT <RSA N> <RSA E>
OUTPUT Please type: number!
OUTPUT We don't sign integers starting with for consecutive ones!
INPUT <ARBITRARY NUMBER>
OUTPUT The signature:
OUTPUT <SIGNATURE>


OUTPUT Hi! Welcome to our note storage system. It's based on RSA!
OUTPUT What do you want to do?\
OUTPUT 1. Write a Note -> Type W
OUTPUT 2. Reading a specific note? -> Type R
OUTPUT 3. Request for a signature of your integer? -> Type S
INPUT R
OUTPUT Please type: note_id token
INPUT <FLAGID> <FUNCTION OF SIGNATURE, N, E>
OUTPUT <FLAG>
```

Figure 9: No-RSA Interaction

Chapter 6

DISCUSSION

6.1 Interpretation of Results

ARCHES introduces a new methodology for generating exploits by example. Previous research focused on generating exploits through identifying and replacing shellcode [4], various binary analysis techniques [19, 1], or from patches [5]; ARCHES uses exploit examples to determine the relationship between state variables and the exploit. Focusing exclusively on the exploit itself ARCHES is uniquely ambivalent to the service in question. This technique is still in its early stages, and far more research is needed to develop a meaningful tool for security researchers and professionals in real-world scenarios. Currently ARCHES is very focused on applications to CTF exploits and, as we showed in section 5, it is still very limited in this area. Some of these limitations, such as support of loops and mathematical operations, are not unsolved problems in this field. ARCHES presents a novel methodology that can be used in concert these other strategies to enhance research into exploit generation.

For exploits which the service must learn how to use known variables and variables provided from the system, this technique shows great promise. ARCHES succeeds at identifying these relationships and doing so faster than existing methods. Where fuzzing and symbolic execution are more feature rich but time intensive, ARCHES stands out in its ability to produce a quick result. The addition of more theories, such as mathematical operations, will add to the duration of exploit learning. While mathematical and certain other useful operations are not currently supported, their

addition to the IRE library is expected. One of the other key advantages of this system will be the ability for users to add domain specific knowledge via the theories they select. If the user knows the transformations that may be required for the exploit, then ARCHES can be setup to test only those theories to minimize the number of examples and time required. Especially in CTFs, some domain knowledge on the types of transformations is common. For instance, a cryptography challenge with allusions to RSA will provide the user with some concept of the math that may be required in this exploit. Similarly, a service with a SQL database could be fed the names of the tables for that database to identify them in SQL Injections (SQLI).

6.2 Implications

As this technology matures, it will lead to drastic changes in the way Capture the Flag games are played. In particular, there is great opportunity for this tool in automated CTF games such as those inspired by the DARPA Cyber Grand Challenge. These methods will be used by CTF teams, service writers, and game hosts alike, all for different end goals. The successful application of this technology to both regular and automatic CTFs will likely result in research into obfuscation tactics to beat this family of learning systems. The information from these systems may also be applied to other security applications, such as patch generation.

As this technology matures, CTF teams will be wary of automated systems deployed by opponents stealing their exploits to reflect them at other teams. Teams have an incentive to prevent this, as opponents scoring from their exploit reduces their individual benefit in common CTF scoring. At the highest level of professional CTFs, such the annual DefCon CTF, teams will likely have methods prepared for their

exploits to prevent easy learning and reflection. Tools used for crafting exploits will not only produce valid exploits, but ones that are hard to reproduce. A simple obfuscation would be changing irrelevant filler bytes' length and content with each run to make it hard for the system to identify a consistent pattern. More complicated obfuscation may discern multiple ways a vulnerability can be hit and exploited and switching between them to prevent the system from even identifying the traffic as a single exploit. Existing systems to make these sorts of vulnerability adjustments have already been built but are not heavily used in CTFs [25]. Another obfuscation technique may send out a similar interaction with each exploit that will appear identical to the system and taken together prevent learning. Finally, with systems like this running it may be in a teams' interest to flood the system with traffic. Since this system will chew through anything that it comes across on the network, the system itself may be manipulated to cause the process to hang. If a team could find a way to make a learning process take a much longer period of time or loop indefinitely, they could take one of their opponents' critical tools out of play.

Additionally, service writers or game hosts that are wary of games being played entirely through reflection systems may implement counter measures. While CTFs want to encourage the use of new technology, it is not unrealistic to assume this tool could, as it advances, become a significant point source for teams. As a completely automated system, it would mean any team could walk in and perform admirably. Since these competitions want to identify and reward the teams with exceptional skill, it is likely these technologies will be hindered to encourage more involved methods of solving challenges. Service writers for these CTFs may develop services with techniques to prevent systems like ARCHES from working effectively. One simple method of doing this is to require a program response that is not easily discernable from the

state. We see one example of this in section 5.1.2.2, where a string response is made to an ASCII picture. Currently it would be very difficult to make a theory by which the program can determine how to transform that image into a string, and the time investment would probably not be worth it. Service developers can also introduce loops such as in section 5.1.2.3. A meaningless loop at the beginning of the program, as this example implemented, would make it difficult for the program to identify a consistent exploit. CTF hosts that want to discourage the use of this technology may also implement restrictions on traffic access, cutting off the examples needed for ARCHES to learn. A CTF host would not have to cut off all traffic, but could hide traffic for a few select services.

On the offensive side of this technology, an ARCHES-like tool will become standard for all advanced CTF teams. This will continue to push CTF challenges towards an automated competition, where team tooling will become increasingly important. Teams with the skills to develop custom theories and tools to utilize ARCHES data will have a distinct advantage. One obvious modification would be the application of ARCHES learned exploits to other uses. Learned exploits may be fed into a symbolic analysis engine to help identify precisely where the vulnerability is located and develop a patch. This semi-automated or fully-automated system would give teams a more rapid patching capability. Additionally, ARCHES generated exploits may be handed off to a single exploit launching system for a team. Here, the teams can distinguish themselves through the strategies they assign to these systems. While all teams may have access to several exploits, teams may implement intelligent strategizers to select how to use the exploits they have for maximum effect. Finally, teams may allow ARCHES to generate partial exploits, things it learned partially but couldn't understand completely, to feed a fuzzer or symbolic execution engine. This would

allow the team to take advantage of the rapid learning from ARCHES to narrow the search space while leveraging existing tools with more features for the "last mile" of exploit generation.

Of course, anything that is useful to CTF challenges will eventually become useful to industry. One of the key technologies that is missing for this would be reliable identification of exploits. Most companies that would benefit from automatic security analysis tools like this deal with such heavy traffic loads that analyzing it all would be impossible. However, ARCHES or a system like it could take traffic that produced unexpected behavior and attempt to learn from that. In this case the goal would not be to reflect the exploit, but to reproduce it and use it to fix the vulnerability. A system with even a small success rate in this regard would be incredibly useful to a company, as with a large code base an attacker may be able to abuse a vulnerability to terrible effect before a human noticed the damage. Existing tools to do this exist, but have not been integrated with ARCHES [7].

6.3 Future Work

While many possible futures for this work have been outlined throughout the paper, this section will coalesce them for easy referencing and elaboration. The future research section can be thought of as four separate avenues for progress on this technology. The first is enhancements to the underlying IRE engine used for the learning. The second is exploit identification and classification. The third is taking advantage of feedback to improve exploits. Finally, the fourth is enchancing the strategies underpinning ARCHES with existing tools.

There are several enhancements that can be made to the IRE engine which will

improve the capabilities of ARCHES. First, IRE can implement a representation for loops and decision statements. In section 5.1.2.2 and section 5.1.2.3 we see two exploits where these capabilities would be useful. Such constructs exist in many exploits, and for other exploits they will be among the main ways for service writers to prevent the use of this technology. Second, and perhaps more importantly, is the addition of many new theories. ARCHES cannot learn anything which IRE does not have a theory to represent. There are many common theories which are not in IRE now that would provide far greater functionality. Such examples include hash algorithms and math operations. Critically with math operations, research must be done into how to learn complex functions, such as $f(x, y) = \frac{2 \times x}{y+1}$. We see in section 5.1.2.4 an exploit example where nested mathematical operations are key to learning the exploit. The challenges with this are to uncover hidden constants, like values one and two, and to learn how to combine everything to produce the output without a search space explosion of trying all possible combinations. This admittedly may not be solvable, however there may be ways theories can be written to identify when mathematical relationships may exist and prompt the user for more precise theories. Finally, refinements to the underlying IRE algorithm which uses reserve bytes which break our exploit in section 5.1.2.2 would permit more general use of this tool.

The most important job of ARCHES is the identification and classification of exploits. This labeling process is what enables the system to provide useful data to the IRE library. Key research into this area revolves around more advanced methods of fingerprinting exploits. That is, how do we know when two conversations are both part of the same exploit? This paper presented a simple heuristic that works well for these examples. However, it is not infallible. Further research into this question will greatly enhance the reliability of ARCHES. Furthermore, in the tests for this

paper we used a noiseless environment where the only traffic was the exploits. In theory, a noisy environment with a reliable fingerprinting algorithm for exploits will categorize the exploits and chaos separately, allowing the system to learn. In reality, noise always introduces errors and even a robust fingerprinting algorithm may error with sufficient random traffic. To prevent this, ARCHES would need to isolate traffic containing exploits from traffic containing noise. This may be done by identifying flags in the service database and only using conversations containing one of those flags to learn. The integration of existing research on identification may improve these results [7]. Additionally, ARCHES could only flag traffic that leads to a crash or other unexpected behavior. Research into how such a feature could be introduced would make ARCHES more useful real-world, high traffic environments. Finally, ARCHES currently has no way of identifying exploits that occur across multiple TCP conversations. We see an example in section 5.1.2.4 of an exploit which requires two conversations to occur in order to retrieve the flag. Currently, there is no method by which ARCHES can correlate two conversations and realize that a relationship exists between them. This may be accomplishable by identifying state variables which are constant across both interactions, and thus realizing they are likely dependent in some fashion. However, this may have a high risk of a false positive rate for exploits which have some constant value in their state, which would be common between all exploit attempts. More research is needed to identify a method by which multiple conversations can be correlated into a single exploit and how to form an exploit from more than one conversation.

One of the greatest limitations with ARCHES that can certainly be improved is the use of all available data to create exploits. Currently, ARCHES only uses the conversations to produce a collection of exploits. If an exploit is tried and doesn't work,

it is discarded. This is to prevent a bad exploit from wasting computing resources. However if that exploit is very close to correct, the previous computing resources already invested will have been wasted. ARCHES should implement a system by which exploits can be enhanced as more data comes in to produce an increasingly correct exploit. Such a system should also track exploits which are known to be ineffective and prevent the same exploit from being tried in the future from a different round of learning. Currently, IRE does not support this type of iterative learning. However, this could be simulated by simply keeping all important data and learning on an increasingly large set of examples. While this is not hard to implement, it was out of scope for the initial iteration of ARCHES. Future work should include this significant improvement.

There are many opportunities for ARCHES to integrate with existing tools. An obvious one would be the use of learned exploits to guide vulnerability identification. This could then be used for generating patches automatically. Such a tool would be incredibly useful to both CTF players and industry. Research into how to feed the IRE exploit representation to a patching tool such as those used in Mayhem or MechanicalPhish would be an exciting future research project with important applications. Additionally, as discussed in earlier sections, partially learned exploits could be used to guide a symbolic execution or fuzzer. This would leverage both the rapid learning of IRE and the more feature rich learning of a symbolic execution engine.

Chapter 7

CONCLUSION

ARCHES introduces a novel method of exploit generation by example, rather than through interaction with the vulnerable program. It leverages an inductive programming library called IRE that learns from a DAG representation of the conversation. Each edge of the DAG is a theory explaining how this observed interaction was created. By combining examples, we constrain the possible theories until only a single theory for each part of the interaction remains. By using these theories we can recreate the interaction to exploit other vulnerable services. This system is optimized for an attack-and-defense CTF into a popular game interface for immediate use. ARCHES uses this game interface to gain information about available services and access the host server to sniff incoming traffic. This traffic is then sorted and learned from to generate an exploit.

We find that ARCHES is capable of successfully reflecting an exploit from as few as just sixteen examples and in only a matter of seconds. ARCHES is still in its infancy and many features required for broader exploit reflection are still missing. We showed that exploits which inherently involve loops or decision statements, multiple TCP connections, and certain bytes used internally as flags are among current limitations which these missing features could address. Most of these problems can be solved with further research.

Inductive programming applied to security has the potential to augment existing automated systems with new capabilities. Given the ability to identify potential exploits, the system could learn the exploit and utilize this to automatically patch or

assist security professionals. ARCHES could also be enhanced with more IRE theories to be able to handle domain specific vulnerabilities and exploits. Finally, the system can be further improved through a database of traffic and improved classification techniques to produce more reliable performance.

Inductive programming is not yet a common technique in the security field. ARCHES presents a promising example of how it may be used to augment existing research and tools. Further improvements will continue to show that this technology is a useful tool to researchers and industry professionals alike.

REFERENCES

[1] T. Avgerinos et al. "The Mayhem Cyber Reasoning System". In: *IEEE Security Privacy* 16.2 (Mar. 2018), pp. 52–60. DOI: 10.1109/MSP.2018.1870873.

[2] Thanassis Avgerinos et al. "AEG: Automatic Exploit Generation". In: (Feb. 2011). DOI: 10.1184/R1/6468296.v1. URL: https://kilthub.cmu.edu/articles/AEG_Automatic_Exploit_Generation/6468296.

[3] T. Bao et al. "How Shall We Play a Game?: A Game-theoretical Model for Cyber-warfare Games". In: *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. Aug. 2017, pp. 7–21. DOI: 10.1109/CSF.2017.34.

[4] Tiffany Bao. "Your Exploit is Mine: Automatic Shellcode Transplant for Remote Exploits". In: (2017). URL: https://www.ieee-security.org/TC/SP2017/papers/579.pdf.

[5] D. Brumley et al. "Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications". In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. May 2008, pp. 143–157. DOI: 10.1109/SP.2008.17.

[6] D. Brumley et al. "Towards automatic generation of vulnerability-based signatures". In: *2006 IEEE Symposium on Security and Privacy (S P'06)*. May 2006, 15 pp.–16. DOI: 10.1109/SP.2006.41.

[7] A. L. Buczak and E. Guven. "A Survey of Data Mining and Machine Learning Methods for Cyber Security Intrusion Detection". In: *IEEE Communications Surveys Tutorials* 18.2 (Secondquarter 2016), pp. 1153–1176. DOI: 10.1109/COMST.2015.2494502.

[8] Michael C. Martin and Monica S. Lam. "Automatic Generation of XSS and SQL Injection Attacks with Goal-Directed Model Checking." In: Jan. 2008, pp. 31–44.

[9] Internet Crime Complaint Center. *2017 Internet Crime Report*. Tech. rep. 2017. URL: https://pdf.ic3.gov/2017_IC3Report.pdf.

[10] Pierre Flener and Ute Schmid. "An introduction to inductive programming". In: *Artificial Intelligence Review* 29.1 (Mar. 2008), pp. 45–62. DOI: 10.1007/s10462-009-9108-7. URL: https://doi.org/10.1007/s10462-009-9108-7.

[11] Sumit Gulwani. "Automatic String Processing Using Input-Output Examples". In: (2011). URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/popl11-synthesis.pdf.

[12] Sumit Gulwani. “Inductive Programming Meets Real World”. In: (2015). URL: https://www.doc.ic.ac.uk/~shm/Papers/indprogreal.pdf.

[13] Timothy D. Harmon. *Cyber Security Capture the Flag (CTF): What Is It?* Sept. 14, 2016. URL: https://blogs.cisco.com/perspectives/cyber-security-capture-the-flag-ctf-what-is-it.

[14] Hong Hu et al. “Automatic Generation of Data-Oriented Exploits”. In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 177–192. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/hu.

[15] S. Huang et al. “CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations”. In: *2012 IEEE Sixth International Conference on Software Security and Reliability*. June 2012, pp. 78–87. DOI: 10.1109/SERE.2012.20.

[16] S. Huang et al. “Software Crash Analysis for Automatic Exploit Generation on Binary Programs”. In: *IEEE Transactions on Reliability* 63.1 (Mar. 2014), pp. 270–289. DOI: 10.1109/TR.2014.2299198.

[17] PortSwigger. “Using Burp Intruder”. In: (2019). URL: https://portswigger.net/burp/documentation/desktop/tools/intruder/using.

[18] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. “Q: Exploit Hardening Made Easy”. In: *Proceedings of the 20th USENIX Conference on Security*. SEC’11. San Francisco, CA: USENIX Association, 2011, pp. 25–25. URL: http://dl.acm.org/citation.cfm?id=2028067.2028092.

[19] UCSB Shellphish. “Cyber Grand Shellphish”. In: (Jan. 25, 2017). URL: http://www.phrack.org/papers/cyber_grand_shellphish.html.

[20] Danny Sullivan. *Google now handles at least 2 trillion searches per year*. May 24, 2016. URL: https://searchengineland.com/google-now-handles-2-999-trillion-searches-per-year-250247.

[21] Frost & Sullivan. *2017 Global Information Security Workforce Study: Benchmarking Workforce Capacity and Response to Cyber Risk*. Tech. rep. 2017. URL: https://iamcybersafe.org/wp-content/uploads/2017/06/Europe-GISWS-Report.pdf.

[22] Frost & Sullivan. *2017 Global Information Security Workforce Study: U.S. Federal Government Results*. Tech. rep. 2017. URL: https://iamcybersafe.org/wp-content/uploads/2017/05/2017-US-Govt-GISWS-Report.pdf.

[23] Erik Trickel et al. "Shell We Play A Game? CTF-as-a-service for Security Education". In: *2017 USENIX Workshop on Advances in Security Education (ASE 17)*. Vancouver, BC: USENIX Association, 2017. URL: https://www.usenix.org/conference/ase17/workshop-program/presentation/trickel.

[24] Fredrik Valeur. "A Learning-Based Approach to the Detection of SQL Attacks". In: (2005). URL: https://link.springer.com/chapter/10.1007/11506881_8.

[25] Minghua Wang et al. "Automatic Polymorphic Exploit Generation for Software Vulnerabilities". In: *Security and Privacy in Communication Networks*. Ed. by Tanveer Zia et al. Cham: Springer International Publishing, 2013, pp. 216–233.