IRE

A Framework For Inductive Reverse Engineering

by

Connor Nelson

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

Approved April 2019 by the
Graduate Supervisory Committee:

Adam Doupé, Chair
Yan Shoshitaishvili
Ruoyu Wang

ARIZONA STATE UNIVERSITY

May 2019

ABSTRACT

Reverse engineering is critical to reasoning about how a system behaves. While complete access to a system inherently allows for perfect analysis, partial access is inherently uncertain. This is the case for an individual agent in a distributed system. Inductive Reverse Engineering (IRE) enables analysis under such circumstances.

IRE does this by producing program spaces consistent with individual input-output examples for a given domain-specific language. Then, IRE intersects those program spaces to produce a generalized program consistent with all examples. IRE, an easy to use framework, allows this domain-specific language to be specified in the form of *Theorist*s, which produce *Theory*s, a succinct way of representing the program space.

Programs are often much more complex than simple string transformations. One of the ways in which they are more complex is in the way that they follow a conversation-like behavior, potentially following some underlying protocol. As a result, IRE represents program interactions as *Conversation*s in order to more correctly model a distributed system. This, for instance, enables IRE to model dynamically captured inputs received from other agents in the distributed system.

While domain-specific knowledge provided by a user is extremely valuable, such information is not always possible. IRE mitigates this by automatically inferring program grammars, allowing it to still perform efficient searches of the program space. It does this by intersecting conversations prior to synthesis in order to understand what portions of conversations are constant.

IRE exists to be a tool that can aid in automatic reverse engineering across numerous domains. Further, IRE aspires to be a centralized location and interface for implementing program synthesis and automatic black box analysis techniques.

DEDICATION

Thank you to Adam, for not expelling me when I reported a potential security vulnerability in his principles of programming languages' project submission server, but instead inviting me to take his course on software security and inviting me to join ASU's hacking team, the pwndevils. His course immediately hooked me into the "cybersecurity" world. It was more game-like than it was course-like, with an actual scoreboard on several of the assignments, which ultimately created the passion I have today for solving security challenges. From there, Adam gave me the amazing opportunity to do security research with him and serve as a teaching assistant in his software security course. These experiences have been truly invaluable and absolutely amazing to be a part of.

Thank you to Yan, for showing me that anything can be done the day before it is absolutely necessary, as we worked through the night and early morning at a coffee shop–just before his first class on systems security–to create a challenge infrastructure (which would be hacked by one of the students just hours later). I remember listening to Yan's talk on angr at DEF CON 23, a couple of years before he joined our lab–and I remember how excited I was when I learned that this legendary hacker might join our lab. The level of energy and humor that Yan has brought to the lab is truly inspiring.

It has been an absolute blast working with both Adam and Yan, as well as Fish, Tiffany, and the rest of the lab. Our amazing lab is the reason that I continue to do research.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

Reverse engineering is a methodology for precisely analyzing the internal workings and substructure of a process or system in order to better understand how it works. In practice, it is done in the absence of high-level specifications and can be thought of as working backward through the standard engineering process—design towards implementation—and instead, implementation towards design. In theory, reverse engineering is relatively straightforward: simply observe how the internals are operating and how the subcomponents are connected. Of course, it is more nuanced than this; but even so, a complete working system that can be observed is, by its very nature, perfectly descriptive of what it does and how it works. This is an underlying requirement of standard reverse engineering. In hardware, a physical object is disassembled and examined. In software, its source code is read through, or in its absence, binary disassembled and machine code analyzed.

Consider, however, the task of reverse engineering without complete access to observing the system. This is the case in distributed systems, where some agent has only partial access to the overall system. Take for instance web applications, where a client makes a request to a server and is only made aware of its response. From the client's perspective, the server is merely a black-box—an oracle—that takes some input and returns some output. What happens in between is left unknown to the client. In such cases, reverse engineering becomes inherently uncertain.

Further consider the problem of systems in which interactions take place between persons and computers; for instance, a human interacting with some computer pro-

gram in a repetitive way. This constitutes a distributed system, where part of the program takes place in the computer, but also part of it takes place in the person's intentions towards interacting with the computer. In such cases, the program occurring in the person's intentions—in the person's mind—is but a black-box to the computer. It is in this way that reverse engineering may be applied not only towards analyzing a computer, but also a person. Reverse engineering may be useful here in order to profile or improve upon the user experience of the person.

Although reverse engineering in these situations becomes an inherently uncertain process, this does not stop human-efforts in reverse engineering. In practice, humans build up an entire model of the black-box system under analysis. They make assumptions based on past systems, attempt to rule out and confirm these assumptions, and use intuition as a means of guiding this process. This inductive reasoning forms the basis for inductive reverse engineering: IRE.

IRE serves to solve this problem of reverse engineering in a black-box environment. IRE is an easy to use, open source, Python 3 framework, that enables users to transform input-output examples into executable programs consistent with those examples. This effectively allows for programming by example. Users can easily introduce domain-specific knowledge about the problem they are working on to further enhance this process.

Chapter 2

BACKGROUND

This sort of reverse engineering without access to internals has become a massively important skill, and in particular, critical to cybersecurity. Take for instance phone phreaking, where early hackers mapped out the phone network and how it worked simply by interacting with it using various tones and observing the results [1]. In more recent times, penetration testing has become an important profession that often relies on reverse engineering in order to audit the security of companies from the perspective of an outside attacker.

One of the common tasks of a penetration tester is in utilizing tools such as black-box web vulnerability scanners. These scanners naively scan a web application, blindly sending exploits at input points and trying to detect if they worked. A survey of these tools shows that there is much work to be done in improving them [2]. Furthermore, efforts to provide more semantic information about an application's state has proven to be effective, despite still fundamentally only having black-box access [3].

In response to the demand for reverse engineering, recent efforts have been made to push towards Cyber Reasoning Systems which aid in this effort, and in some cases entirely automate it. The Defense Advanced Research Projects Agency (DARPA) led an initiative to develop fully autonomous systems capable of reverse engineering and exploiting challenge binaries in their Cyber Grand Challenge [4]. This has led to significant advances in program analysis and various techniques surrounding state-of-the-art reverse engineering. Many of these techniques can be seen in open source

frameworks for performing program analysis including *angr* and *Manticore* [5]–[8]. These frameworks provide users with tools for precisely reasoning about a program by analyzing their internals.

Chapter 3

THEORISTS AND THEORIES

IRE works around two central primitives: theorists and theories. Theorists use input-output examples to produce theories. They act as a sort of domain-specific language. Theories use input to produce output. They model the synthesized program.

3.1 Grammar Theorists

IRE fundamentally solves a parsing problem. It must parse input-output examples to produce a program, not entirely dissimilar to a programming language parser which parses source code to produce a program. For this reason, it is useful to think in terms of context-free grammars. While in the programming language realm

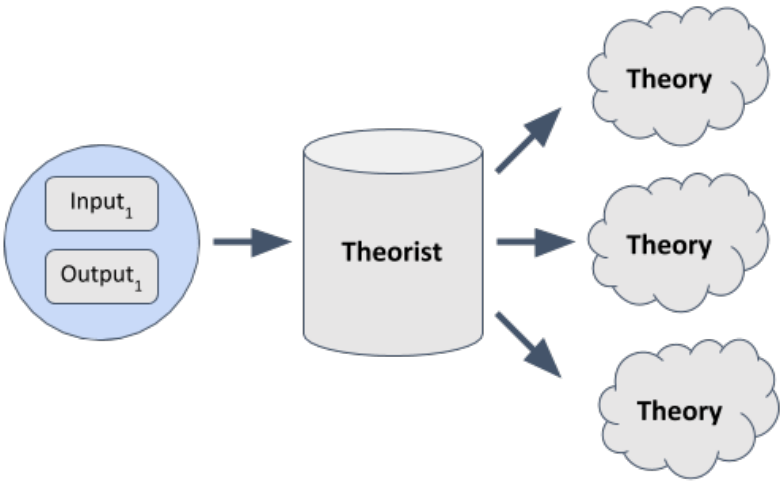

Figure 1. Theorist

Figure 2. Theory

```
import ire

class HTML(ire.GrammarTheorist, entry='html'):
    html = ire.RepeatTheorist(html_element)
    html_element = (open_tag & close_tag & self_tag) | data

    open_tag = '<' + tag_content + '>'
    close_tag = '</' + tag_name + '>'
    self_tag = '<' + tag_content + '/>'

    data = '.|\n' & ire.FunctionTheorist(None) & ire.FunctionTheorist(b64)

    tag_content = (tag_name + whitespace + attributes) | tag_name
    tag_name = '[a-zA-Z]+'

    attributes = (attribute + whitespace + attributes) | attribute
    attribute = '[a-zA-Z]+' & ('[a-zA-Z]+="' + attribute_value + '"')
    attribute_value = '[a-zA-Z0-9]'

    whitespace = '\s+'
```

Figure 3. HTML Grammar Theorist

context-free grammars strive to be unambiguous, IRE leverages ambiguity to produce the program space. Consider, for instance, the HTML grammar theorist (Figure 3) written using the IRE framework.

Here, a *GrammarTheorist* is defined, with *html* being the entry point, or start symbol in context-free grammar terminology. Variables that appear on the left side of

an = indicate a nonterminal symbol, while all other expressions are terminal symbols. Each of these assignments forms the basis for a production rule.

Strings are implicitly *RegexTheorist*s, which as the name suggests, perform regex matching. Addition expressions (using the + operator) are implicitly *ConcatTheorist*s, which serve to concatenate theorists, and their produced theories, together.

Rather than defining several production rules for the same nonterminal, operators & and | both respectively serve to implicitly create *AndTheorist*s and *OrTheorist*s. All theorists within an *AndTheorist* will attempt to produce theories. After some theorist within an *OrTheorist* has produced some number of theories, any remaining theorists will not be given a chance to produce theories. This evaluation takes place from left to right and allows for a conditionally restricted search space, and consequently more efficient parsing. On line 5, the *OrTheorist* indicates that an *html_element* can be an *open_tag*, *close_tag* and $self\_tag$, or if that is not the case then it must be *data*.

On line 4, the *RepeatTheorist* indicates that parsing should repeatedly consume (one or more times) *html_element*s until it no longer can. Implementation wise, this is more efficient than how looping is traditionally performed in context-free grammars by having a self-referential production rule.

*FunctionTheorist*s wrap functions to make them behave as theorists. Line 11 showcases two of these theorists. $FunctionTheorist(None)$ indicates the identity function ($f(x) = x$). $FunctionTheorist(b64)$ on the other hand assumes the existence of a $b64$ function (defined elsewhere), which base64 encodes its input.

```
def echo(name, msg):
    return \
f"""
<html>
    <head>
        <title>Echo</title>
    </head>
    <body>
        <p>Hello {name}!</p>
        <p>"{msg}" is {b64(msg)}</p>
    </body>
</html>
"""
```

Figure 4. Simple HTML Echo Program

```
<html>
   <head>
      <title>Echo</title>
   </head>
   <body>
      <p>Hello Paul!</p>
      <p>“Hello World” is SGVsbG8gV29ybGQ=</p>
   </body>
</html>
```

Figure 5. Output of Simple HTML Echo Program

3.2 Theorists to Theories

Consider the simple HTML echo program (Figure 4) in order to understand how IRE is able to utilize the prior HTML grammar theorist (Figure 3) to reason about this program and produce theories.

This simple program (Figure 4) transforms the inputs into an HTML output. Inputs `name='Paul'` and `msg='Hello World'` result in the output shown in Figure 5.

Running this input-output example (Figure 5) through the prior HTML grammar

```
{
<html>
    <head>
       <title>Echo</title>
    </head>
    <body>
       <p>Hello {{{Input[0]}}, Paul}!</p>
       <p>“{{{Input[1]}}, Hello World}” is {{{b64(Input[1])}}, SGVsbG8gV29ybGQ=}</p>
    </body>
</html>
}
```

Figure 6. Simple HTML Echo Theory

theorist (Figure 3) results in the theory shown in Figure 6, displayed as a simple program summary using the IRE framework.

In this simple program summary (Figure 6), the gray curly braces and commas represent *UnionTheory*s, and the blue text represents individual theories within those *UnionTheory*s. *UnionTheory*s are a way of succinctly representing the program space, allowing common theories among candidate program traces to not be repeated, effectively forming a directed acyclic graph. Double curly braces indicate theories that depend on the input (e.g. *FunctionTheory*s).

This shows an interesting result common to running a theorist against only one input-output example: the complete original constant output by itself appears as a possible program trace within the program space. These constant theories can be ruled out only with more input-output examples.

3.3 Theory Intersection

In order to collapse the program space, theories resulting from different input-output examples must be intersected together.

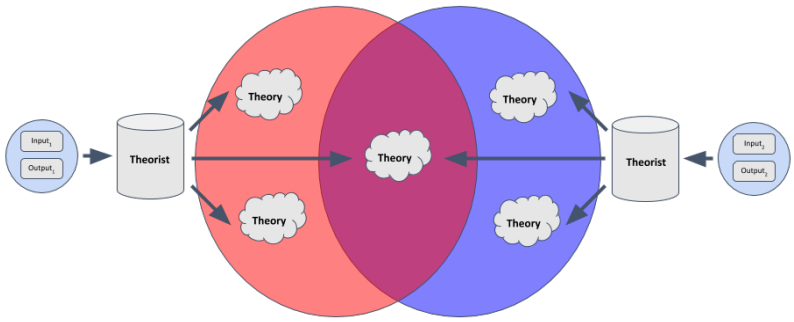

Figure 7. Theory Intersection

```
<html>
   <head>
      <title>Echo</title>
   </head>
   <body>
      <p>Hello Pablo!</p>
      <p>“Hola Mundo” is SG9sYSBNdW5kbw==</p>
   </body>
</html>
```

Figure 8. Another Output of Simple HTML Echo Program

```
{
<html>
   <head>
      <title>Echo</title>
   </head>
   <body>
      <p>Hello {{{Input[0]}}, Pablo}!</p>
      <p>“{{{Input[1]}}, Hola Mundo}” is {{{b64(Input[1])}}, SG9sYSBNdW5kbw==}</p>
   </body>
</html>
}
```

Figure 9. Another Simple HTML Echo Theory

Inputs `name='Pablo'` and `msg='Hola Mundo'` result in the output shown in Figure 8.

Running this input-output example (Figure 8) through the prior HTML grammar theorist (Figure 3) results in the theory shown in Figure 9, displayed as a simple program summary using the IRE framework.

```
{
<html>
    <head>
       <title>Echo</title>
    </head>
    <body>
       <p>Hello {{{Input[0]}}}!</p>
       <p>“{{{Input[1]}}}” is {{{b64(Input[1])}}}</p>
    </body>
</html>
}
```

Figure 10. Simple HTML Echo Theory Intersected

Intersecting the first theory (Figure 6) with the second theory (Figure 9), results in just one program trace in the program space: our original program (Figure 4).

Chapter 4

CONVERSATIONS

Often times, distributed programs follow a conversation-like behavior, potentially following some underlying protocol. In such cases, program synthesis cannot merely take place over only simple input-output examples. Instead, input-output examples must be generalized to conversation examples. Here, discerning between input and output doesn't necessarily make sense, as it depends on an agent's perspective. Instead, a conversation takes place over a series of messages, where a message has some source agent and destination agent. Further, these agents have context–a generalization of input–representing what an agent individually brings as their input to a particular conversation. This allows for some notion of state. Nevertheless, in a deterministic program, identical contexts will produce identical conversations.

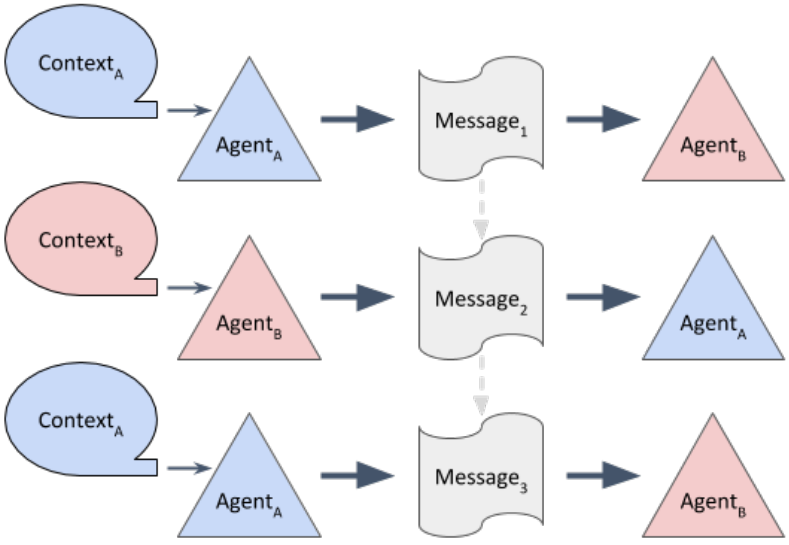

Figure 11. Conversation

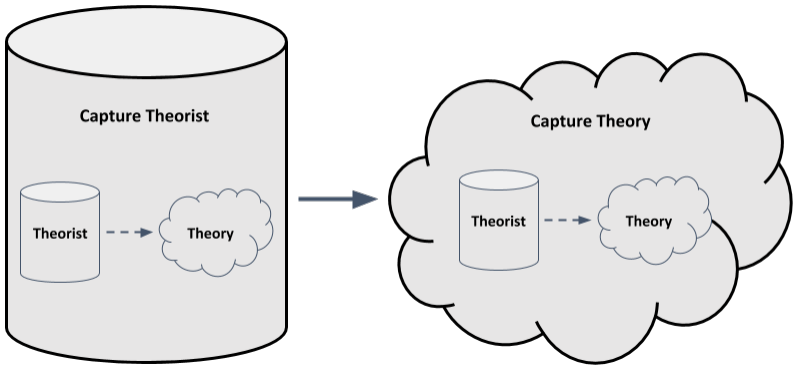

Figure 12. Capture Theorist and Capture Theory

4.1 Capturing Theorists and Theories

An agent is not aware of other agents' contexts; only its own context and any received messages. This is a fundamental problem in reverse engineering distributed systems. Therefore, IRE must use an agent's received messages in order to recover the sender's context. It does so using *CaptureTheorist*s and *CaptureTheory*s.

*CaptureTheorist*s wrap other theorists and effectively leverage their ability to parse. Those *CaptureTheorist*s go on to create *CaptureTheory*s, which also wrap those same theorists. During synthesis, the underlying theorists attempt to produce theories, and anything consumed during this is captured. This captured data may then be used in later theorists, much like a traditional input. Those produced theories do not actually become a part of the program space, but instead the *CaptureTheory*s storing the underlying theorist does. This allows this parsing done during the synthesis to be performed again during execution of the theory.

Consider the HTTP theorists shown in Figure 13, written using the IRE framework, in order to understand how these *CaptureTheorist*s and *CaptureTheory*s may be applied.

In the case of web applications, it is common for an HTTP server to negotiate some token, commonly known as a cookie, for keeping track of state with its clients [9]. Cookies are a way of enabling statefulness in the otherwise stateless HTTP protocol. An HTTP client will include the cookies associated with a particular server with all web requests made to that server. By introducing a *CaptureTheorist* on line 28, this behavior is effectively conveyed and enables IRE to capture the cookie.

Now consider the HTTP conversation (Figure 14) that results from the client's context of `username='Paul'`, `password='p455w0rd'`, and `msg='Hello_World'`; and unknown to the client, server's context of `cookie='sessionid=12345'`.

Running this conversation example (Figure 14) through the prior HTTP theorist (Figure 13) results in the theory shown in Figure 15, displayed as a simple program summary using the IRE framework.

With another example conversation, this program space could be collapsed as discussed in Section 3.3.

```
class HTTPRequest(ire.OutputMessageTheorist, entry='request'):
    request = request_prolog + headers + '\n' + request_contents
    request_prolog = '(GET|POST) ' + request_url + ' HTTP/1.1\n'
    request_url = ire.RepeatTheorist(request_url_data)
    request_url_data = '[^ ]' & inputs

    headers = ire.RepeatTheorist(header)
    header = '[a-zA-Z-]+: ' + ire.RepeatTheorist(header_data) + '\n'
    header_data = '.' & inputs

    request_contents = ire.RepeatTheorist(request_contents_data) | ''
    request_contents_data = '.|\n' & inputs

    inputs = ire.FunctionTheorist(None) & ire.FunctionTheorist(b64)


class HTTPResponse(ire.InputMessageTheorist, entry='response'):
    response = response_prolog + headers + '\n' + response_contents
    response_prolog = 'HTTP/1.1 [0-9]+ [a-zA-Z ]+\n'

    headers = ire.RepeatTheorist(cookie_header | header)
    cookie_header = 'Set-Cookie: ' + cookie + '\n'
    header = '[a-zA-Z-]+: ' + ire.RepeatTheorist(header_data) + '\n'
    header_data = '.' & inputs

    response_contents = HTML() | ''

    cookie = ire.CaptureTheorist(cookie_data, 'cookie')
    cookie_data = '[a-zA-Z]+=[a-zA-Z0-9]+'

    inputs = ire.FunctionTheorist(None) & ire.FunctionTheorist(b64)


http = HTTPRequest() & HTTPResponse()
```

Figure 13. HTTP Theorist

```
POST /login HTTP/1.1
Host: example.com
Content-Type: application/json

{“username”: “Paul”, “password”: “p455w0rd”}
```

```
HTTP/1.1 200 OK
Set-Cookie: sessionid=12345
Connection: close
```

```
GET /echo?msg=Hello_World HTTP/1.1
Host: example.com
Cookie: sessionid=12345
```

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Connection: close

<html>
    <head>
        <title>Echo base64</title>
    </head>
    <body>
        <p>Hello Paul!</p>
        <p>“Hello_World” is SGVsbG9fV29ybGQ=</p>
    </body>
</html>
```

Figure 14. Simple HTTP Conversation

```
{
POST /login HTTP/1.1
Host: example.com
Content-Type: application/json

{“username”: “{{{Input[username]}}, Paul}”, “password”: “{{{Input[password]}}, p455w0rd}”}
}
```

```
{
HTTP/1.1 200 OK
Set-Cookie: {{{Capture[cookie]}}}
Connection: close

}
```

```
{
GET /echo?msg={{{Input[msg]}}, Hello_World} HTTP/1.1
Host: example.com
Cookie: {{{Input[cookie]}}, sessionid=12345}

}
```

```
{
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Connection: close

<html>
   <head>
      <title>Echo base64</title>
   </head>
   <body>
      <p>Hello {{{Input[username]}}, Paul}!</p>
      <p>“{{{Input[msg]}}, Hello_World}” is {{{b64(Input[username])}}},
SGVsbG9fV29ybGQ=}</p>
   </body>
</html>
}
```

Figure 15. Simple HTTP Theory

Chapter 5

GRAMMARLESS SYNTHESIS

While it is useful to be able to provide domain-specific knowledge to an analysis, doing so is not always possible. Providing a grammar to IRE allows it to be more efficient in its search, by reducing the possible program space. Consider, for instance, the grammar shown in Figure 16 which is highly generic.

This grammar (Figure 16) repeatedly matches against a single byte and derivations of the input. Repeatedly matching against a single byte is what effectively drives the parsing forward.

Inputs `name='Paul'` and `msg='body'` run against the prior simple program (Figure 4) and through the generic theorist (Figure 16) result in the theory shown in Figure 17, displayed as a simple program summary using the IRE framework.

```
1  class Generic(ire.GrammarTheorist, entry='entry'):
2      entry = ire.RepeatTheorist(data)
3      data = '.|\n' & ire.FunctionTheorist(None) & ire.FunctionTheorist(b64)
```

Figure 16. Generic Theorist

```
{
<html>
    <head>
        <title>Echo</title>
    </head>
    <{{{Input[1]}}}, body}>
        <p>Hello {{{Input[0]}}}, Paul}!</p>
        <p>“{{{Input[1]}}}, body}” is {{{b64(Input[1])}}}, Ym9keQ==}</p>
    </{{{Input[1]}}}, body}>
</html>
}
```

Figure 17. Generic Theory

Note that in this case, the HTML body tags are considered to have been potentially derived from the input. Of course, with another input-output example, this program space could be collapsed to resolve this as discussed in Section 3.3. Nevertheless, it is useful for demonstrating how much extra work must be needlessly done in the absence of domain-specific knowledge about the HTML structure of this program's output.

5.1 Conversation Intersection

The prior structure of the generic theorist (Figure 16) does not scale very well with message sizes and the number of core theorists (e.g. $FunctionTheorist$s). Consider for instance a message of 100,000 bytes, with hundreds of potential $FunctionTheorist$s and $CaptureTheorist$s defined to allow for all sorts of context derivations. This would likely be the case in applying IRE to real world web applications, where HTML pages get quite large, and context derivations quite complex. Under such circumstances, applying this sort of generic theorist would become infeasible.

Fortunately, IRE can do much better by analyzing multiple conversations at once, prior to producing their theories and intersecting them. With this early access, IRE can perform a sort of conversation intersection before it begins its traditional analysis. In doing so, IRE analyzes the supplied conversations to determine their structure. In particular, it looks for portions of messages which remain constant throughout all of the conversations. IRE uses the insight that if portions of a message don't change across any conversation examples, then it must be the case that a consistent program trace keeps it constant. Boldly, IRE assumes that it must be constant.

```
<html>
   <head>
      <title>Echo</title>
   </head>
   <body>
      <p>Hello Pa{u, b}l{, o}</p>
      <p>“H{ell, }o{, la}” is SG{V, 9}s{bG8, VQ=}=</p>
   </body>
</html>
```

Figure 18. Conversation Intersection

Effectively, IRE automatically generates a grammar, where constant sections are parsed by constant theorists, and nonconstant sections are parsed by all supplied core theorists. This can dramatically decrease the program search space. However, IRE must first determine which sections of messages are constant, and which aren't.

This is done by taking all corresponding messages across all conversations, and finding the longest common substring across them. The longest common substring is marked as being constant in the grammar, and this is recursively applied to all the prefixes and all the suffixes. This continues on until there no longer is a common substring. This ultimately results in marking the maximum amount of constant data across all conversations.

Observe what happens (Figure 18) when the output from input `name='Paul'` and `msg='Hello'` and output from input `name='Pablo'` and `msg='Hola'` run against the prior simple program (Figure 4) are intersected.

This shows an interesting result common to intersecting only a few examples: too much of the conversation is marked constant. This can potentially cause problems with inputs being fragmented, though this is not necessarily a major issue. Fortunately, though, with more examples this problem is resolved. Figure 19 shows the result of introducing another output due to input `name='Joe'` and `msg='ABC'`.

This discovered structure (Figure 19), in turn, is used by IRE for automatically

```
<html>
   <head>
      <title>Echo</title>
   </head>
   <body>
      <p>Hello {Paul, Pablo, Joe}!</p>
      <p>“{Hello, Hola, ABC}” is {SGVsbG8=, SG9sYQ==, QUJD}</p>
   </body>
</html>
```

Figure 19. Another Conversation Intersection

generating a grammar. This dramatically reduces the possible program space, enabling IRE to be much more efficient.

Chapter 6

RELATED WORK

Early work in this field has formalized inductive learning as a search problem [10]. Past work has shown just how useful version space algebras can be in representing a program space [11]–[13]. Several search techniques including enumerative, stochastic, and constraint-based algorithms have been developed in order to search the program space in a number of domains [14]–[17].

Microsoft has made major contributions in this area. Much of the research discussed here builds upon their work in Inductive Program Synthesis. Their *PROSE SDK* enables program synthesis development in a way akin to what this research explores [18]. Their *FlashFill* program demonstrates some of the potential applications that this research has for millions of users [19], [20].

The specific problem of protocol reverse engineering is a very closely related area that has been researched in order to quickly understand custom botnet protocols in *Prospex* and *Dispatcher* [21], [22].

Because this topic can be applied to so many areas, research on it is scattered in several fields [23]. For this reason, IRE aspires to be a centralized location and interface for implementing program synthesis and automatic black box analysis techniques.

REFERENCES

[1] K. D. Mitnick and W. L. Simon, *The art of deception: Controlling the human element of security*. John Wiley & Sons, 2011.

[2] A. Doupé, M. Cova, and G. Vigna, "Why johnny can't pentest: An analysis of black-box web vulnerability scanners," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2010, pp. 111–131.

[3] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A state-aware black-box web vulnerability scanner," in *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, 2012, pp. 523–538.

[4] T. Shellphish, "Cyber grand shellphish," *Phrack Papers*, 2017.

[5] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," 2016.

[6] 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 *NDSS*, vol. 16, 2016, pp. 1–16.

[7] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," 2015.

[8] T. of Bits, *Manticore: Symbolic execution for humans*, https://github.com/trailofbits/manticore.

[9] A. Barth, "Rfc 6265-http state management mechanism," *Internet Engineering Task Force (IETF)*, pp. 2070–1721, 2011.

[10] T. M. Mitchell, "Generalization as search," *Artificial intelligence*, vol. 18, no. 2, pp. 203–226, 1982.

[11] H. Hirsh, "Theoretical underpinnings of version spaces.," in *IJCAI*, 1991, pp. 665–670.

[12] T. A. Lau, P. M. Domingos, and D. S. Weld, "Version space algebra and its application to programming by demonstration.," in *ICML*, 2000, pp. 527–534.

[13] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld, "Programming by demonstration using version space algebra," *Machine Learning*, vol. 53, no. 1-2, pp. 111–156, 2003.

[14] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur, "Transit: Specifying protocols with concolic snippets," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 287–296, 2013.

[15] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *ACM SIGPLAN Notices*, ACM, vol. 48, 2013, pp. 305–316.

[16] A. Solar-Lezama and R. Bodik, *Program synthesis by sketching*. Citeseer, 2008.

[17] J. K. Feser, S. Chaudhuri, and I. Dillig, "Synthesizing data structure transformations from input-output examples," in *ACM SIGPLAN Notices*, ACM, vol. 50, 2015, pp. 229–239.

[18] S. Gulwani, "Programming by examples: Applications, algorithms, and ambiguity resolution," in *International Joint Conference on Automated Reasoning*, Springer, 2016, pp. 9–14.

[19] ——, "Automating string processing in spreadsheets using input-output examples," in *ACM SIGPLAN Notices*, ACM, vol. 46, 2011, pp. 317–330.

[20] S. Gulwani, J. Hernández-Orallo, E. Kitzelmann, S. H. Muggleton, U. Schmid, and B. Zorn, "Inductive programming meets the real world," *Communications of the ACM*, vol. 58, no. 11, pp. 90–99, 2015.

[21] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *Security and Privacy, 2009 30th IEEE Symposium on*, IEEE, 2009, pp. 110–125.

[22] J. Caballero, P. Poosankam, C. Kreibich, and D. Song, "Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering," in *Proceedings of the 16th ACM conference on Computer and communications security*, ACM, 2009, pp. 621–634.

[23] E. Kitzelmann, "Inductive programming: A survey of program synthesis techniques," in *International workshop on approaches and applications of inductive programming*, Springer, 2009, pp. 50–73.