University of California
Santa Barbara

Building a Base for Cyber-autonomy

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Yan Shoshitaishvili

Committee in charge:

Professor Giovanni Vigna, Co-Chair
Professor Christopher Kruegel, Co-Chair
Professor Timothy Sherwood

September 2017

The Dissertation of Yan Shoshitaishvili is approved.

Professor Timothy Sherwood

Professor Christopher Kruegel, Committee Co-Chair

Professor Giovanni Vigna, Committee Co-Chair

August 2017

Building a Base for Cyber-autonomy

This dissertation is dedicated to my grandmothers, one of whom started me on this path, and one of whom wasn't able to see me complete this stage of it.

Acknowledgements

I always see elegant and impactful acknowledgement sections in dissertations and theses, but I don't know how to write one myself. So many people in my life had such fundamental impact that it seems impossible to distill such information into a single coherent section. Moreover, many of these people had impact through many phases in my life, making proper organization of such a section quite a daunting task.

So, let's start at the beginning. My journey, of which this dissertation was an exciting step, would never have started without my family. They initiated, supported, and maintained the passion for computing that drove me to where I am today. My grandmothers (Klara and Anna) sparked my early interest in technical pursuits. My mother (Irina) let me (ab)use our sole family computer in my exploration of the internals of Windows 3.1 and DOS, despite relying on that computer for her career. My father (Alex) constantly tried to open my eyes to new areas of exploration in computing (in fact, he was the first to suggest that I explore security as a career). My siblings (Igor, Elena, and Boris) supported me unconditionally through the many stages of this journey, offering advice, encouragement, and motivation.

My childhood friends (Nathan, Sean, Jordan, Derek, Max, Adiv, and my brother Boris) provided opportunities for expanding my technical knowledge (because of our proclivity to LAN classic games, for example, I could practically talk IPX by hand). In a later phase of life, they provided the competitive motivation to not be left behind, as they all went off to graduate school. In fact, Sean provided the lifeline to UCSB, of which I would have never known if he had not blazed that path.

During the PhD, I found myself in an unrivalled environment of intellectual challenge and support. I am eternally thankful to fellow students, postdocs, interns, and random hackers at the UC Santa Barbara SecLab and the wider security research community.

There are too many names to list here, and any partial list will be woefully incomplete, but I will note just a few. Alexandros and Manuel, who helped me understand what it meant to do research and provided support (whether they realized it or not) when it was most needed. Fish, who showed up for an internship in the lab, bought into my crazy vision for the future of binary analysis, and stayed to make it a reality. Dick, who laid the ground work for the lab's existence, and who evidently invented everything that I've published over the course of my PhD, but 30 years earlier. And, of course, my advisors Chris and Giovanni, who saw and guided the research potential in me even when I did not consider it to be there myself, and who somehow knew when to give me free reign and when to reign me in. Their example is something that I will strive to emulate throughout my academic career, and it is my hope that a student of mine will one day be as thankful to me as I am to them.

Finally, we come to Brooke, who started this journey as a stranger, became my girlfriend for the bulk of it, my fiance for the final push, and starts the next step with me as my wife. Few people could have put up with both me and the filtered insanity of the PhD process, and I could not have asked for a better partner over the years. Maybe one day, I will somehow make up for all the missed evenings, weekends, and (during the insanity of the Cyber Grand Challenge) entire months. From what I understand of the Professor route, I doubt she's holding her breath...

Curriculum Vitæ
Yan Shoshitaishvili

Education

2017	Ph.D. in Computer Science (Expected), University of California, Santa Barbara.
2006	B.Sc. in Computer Science, Rensselaer Polytechnic Institute.

Publications

1. Antonio Bianchi, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna. *Blacksheep: Detecting Compromised Hosts in Homogeneous Crowds.* ACM CCS 2012.
2. Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Christopher Kruegel, Giovanni Vigna. *Revolver: An Automated Approach to the Detection of Evasive Web-based Malware.* Usenix Security 2013.
3. Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna. *Steal this Movie - Automatically Bypassing DRM Protection in Streaming Media Services.* Usenix Security 2013.
4. Giancarlo De Mayo, Alexandros Kapravelos, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna. *PExy: The other side of Exploit Kits.* DIMVA 2014.
5. Yinzhi Cao, Yan Shoshitaishvili, Kevin Borgolte, Christopher Kruegel, Giovanni Vigna, Yan Chen. *Protecting Web-based Single Sign-on Protocols against Relying Party Impersonation Attacks through a Dedicated Bi-directional Authenticated Secure Channel.* RAID 2014.
6. Yan Shoshitaishvili, Luca Invernizzi, Adam Doupe, Christopher Kruegel, Giovanni Vigna. *Do You Feel Lucky? A Large-Scale Analysis of Risk-Reward Trade-Offs in Cyber Security.* ACM SAC 2014.
7. Giovanni Vigna, Kevin Borgolte, Jacopo Corbetta, Adam Doupe, Yanick Fratantonio, Luca Invernizzi, Dhilung Kirat, Yan Shoshitaishvili. *Ten Years of iCTF: The Good, The Bad, and The Ugly.* Usenix 3GSE 2014.
8. Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, Giovanni Vigna. *Firmalice: Detecting Authentication Bypass Vulnerabilities in Embedded Devices.* NDSS 2015.
9. Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna. *Portrait of a Privacy Invasion - Detecting Relationships Through Large-scale Photo Analysis.* PETS 2015.
10. Alessandro Di Federico, Amat Cama, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna. *How the ELF Ruined Christmas.* Usenix Security 2015.

11. Nick Stephens, John Grosen, Chris Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna. *Driller: Augmenting Fuzzing Through Symbolic Execution.* NDSS 2016.

12. Yan Shoshitaishvili, Ruoyu Wang, Chris Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, Giovanni Vigna. *SoK: (State of the) Art of War: Offensive Techniques in Binary Analysis.* IEEE Security and Privacy 2016.

13. Marius Muench, Fabio Pagani, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna, Davide Balzarotti. *Taming Transactions: Towards Hardware-Assisted Control Flow Integrity using Transactional Memory.* RAID 2016.

14. Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, Giovanni Vigna. *Ramblr: Making Reassembly Great Again.* NDSS 2017 - Distinguished Paper Award.

15. Yan Shoshitaishvili, et al. *Cyber Grand Shellphish.* Phrack 2017.

16. Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, David Brumley. *Your Exploit is Mine: Automatic Shellcode Transplant for Remote Exploits.* IEEE Security and Privacy 2017.

17. Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. *BootStomp: On the Security of Bootloaders in Mobile Devices.* Usenix Security 2017.

18. Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, Giovanni Vigna. *Rise of the HaCRS: Augmenting Autonomous Cyber Reasoning Systems with Human Assistance.* ACM CCS 2017.

19. Jacob Corina, Aravind Machiry, Christopher Salls, Shuang Hao, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna. *DIFUZE: Interface Aware Fuzzing for Kernel Drivers.* ACM CCS 2017.

Abstract

Building a Base for Cyber-autonomy

by

Yan Shoshitaishvili

As software becomes increasingly embedded in our daily lives, it becomes more and more critical to find the vulnerabilities in this software. Worse, since the amount and variety of this software is rapidly proliferating, manual analysis by rare, talented hackers cannot scale to keep this software safe.

My first foray into what became my dissertation work tried to address a specific class of this general problem: backdoors inserted (either due to malice or for later support and maintenance convenience) into internet-connected embedded devices, such as smart power-meters. To address the requirement of having to reason about logical bugs and to analyze enormous amounts of binary code, I created a novel combination of static and dynamic-symbolic analysis techniques. Combining this with an insight into a new way to define a backdoor, I was able to build a system that analyzed firmware of real-world devices to identify such vulnerabilities in them.

This first foray led into my main contribution: the generalization of this analysis composition in the form of a principled binary analysis framework built to enable the seamless combination of diverse program analysis techniques. This framework, angr, provides a powerful base future research from myself, my lab-mates, and researchers around the world (as the framework is fully open source). One of the early applications of the system was the identification of authentication bypass vulnerabilities in binary firmware using a combination of static analysis and dynamic symbolic execution.

Using angr, we built an autonomous program analysis system that was able to analyze, exploit, and protect binary code without any human intervention. This system, the Mechanical Phish, won third place in the Cyber Grand Challenge, a competition created by DARPA to bootstrap the development of autonomous Cyber Reasoning Systems. While the system did well, its performance in the Cyber Grand Challenge provided an insight that shaped the conclusion of my dissertation: even with the current program analysis techniques combined into a coherent Cyber Reasoning System, serious limitations still exist. In the final work of my graduate studies, I explored the careful reintegration of human assistance into our analysis automation, in a way that addresses its limitations without compromising its scalability advantage over manual analysis.

Contents

Chapter 1

Introduction

When I was 5 years old, my grandmother gifted me a book called "Professor Fortran's Encyclopedia" [1]. It was an illustrated book for children, following the adventures of a cat (named X), a caterpillar (named Caterpillar), and a sparrow (named Sparrow), led by their professor (named Fortran), as they learn about computing. This book revealed a whole new world to me, in which all the basic rules were understood, but where incredible, awe-inspiring, reason-defying things could nonetheless exist. I have been fascinated by computers ever since. In my fascination, I aimed to understand them. I ditched many a class in first grade, hid in the stairwell, and read and reread my book. I learned BASIC, then dug deeper to C and C++ (which I gleaned from the help files of the Borland Turbo C++ Compiler), and, in my undergraduate Computer Organization class, x86 assembly and all the way back up the stream of computational complexity to logic gates.

During this journey, something strange happened. When we crossed the threshold from assembly to logic gates, somewhere just a bit downstream of adders and multiplexers, I realized that the magic that defined computing for me was suddenly gone. Rather than the expected fascination at the underpinnings of computation, I instead found dry, boring, predictable logic gates. For any logic gate or, seemingly, collection of logic gates, the full range of inputs could simply be mapped to the full range of outputs. There seemed to be no mystery or intelligence. However, just a bit downstream, where the term software gains meaning, there was magic, complexity, and a thrilling feeling of uncertainty. Here, computing felt alive. [1]

I am a strong believer in the concept that understanding a mysterious phenomena makes it all the more magical, and I strove to understand this layer of our digital world. When you understand the bits comprising the operators and operands of binary software better than the authors of that software, you make an interesting discovery. You realize that, by carefully taking advantage corner cases and situations that the authors had not considered, you can (metaphorically) take a program that was designed to walk and, through careful manipulation of what it sees, hears, and is prompted to do, you can make it dance.

For a while, this was enough: I would set up my digital dollhouse and I would play with my toys. However, as any overseer of a vast and artificial landscape, I eventually grew lonely. I needed to go epistemologically deeper. I didn't just want to just understand what was going on – I wanted to understand how to understand it. I wanted to be able to create programs that could understand other programs and, in turn, exploit and manipulate these programs to their own ends.

Of course, I was not the first to have this idea. The field of program analysis (and the specific subfield of binary analysis, where I wanted to operate) has been an active research area for decades. Researchers had created the building blocks of binary analysis. I wanted to build on this existing foundation of program analysis and provide that final push, where I could watch one program look at another and show it how to do something that it never before thought possible.

I eventually achieved this, leading the development of an autonomous system that was capable of automatically analyzing, exploiting, and defending previously unknown

[1]I should mention, here, that going too far downstream also results in boredom – of course it makes sense that a complex programming language can express anything your mind desires. It is specifically the border where machines became alive, right there on the binary level, that fascinates me.

binary software. This system, the Mechanical Phish, competed in the DARPA Cyber Grand Challenge, the first ever hacking contest from which human beings were banned, fighting six other autonomous “Cyber Reasoning Systems” and winning third place and a spot in history. In many ways, this dissertation is the story of this system.

This journey started, like many a research direction, with an attempt to reuse as much existing work in the field as possible. That is, rather than undertaking the creation of a binary analysis framework, I tried to identify frameworks that could server as a base for my research. I had a number of requirements: the framework had to support multiple architectures (since the proliferation of the Internet of Things has resulted in a wide range of such architectures in active use), had to be open source (so that we could extend it over the course of our research), and had to lend itself to a range of analyses (and to their composition). Unfortunately, at the time, no framework existed that met these requirements.

Thus, the first contribution of my dissertation work became the creation of a principled binary analysis framework built to enable the seamless development and combination of diverse program analysis techniques. This framework, angr, provided a powerful base for my research, and has since been adopted as one of the main binary analysis engines of researchers and enthusiasts around the world. The philosophy and design behind this system in detailed in Chapter 3.

My first work using angr tried to detect a class of logical vulnerabilities which has traditionally been difficult to detect automatically: backdoors inserted (either due to malice or for later support and maintenance convenience) into internet-connected embedded devices, such as smart power-meters. To address the requirement of having to reason about logical bugs and to analyze enormous amounts of binary code, I created a novel combination of static and dynamic-symbolic analysis techniques, composing them by leveraging angr's modular design. Combining this with an insight into a new way to define a backdoor, I was able to build a system that analyzed the firmware of real-world devices to identify such vulnerabilities in them. This technique is described in Chapter 4.

A later (and more high-profile) application of angr was the aforementioned Cyber Reasoning System, the Mechanical Phish, which leveraged angr to great effectiveness for the autonomous analysis, exploitation, and defense of binary code. Interestingly, however, its performance in the Cyber Grand Challenge provided an insight that shaped the concluding chapter of my dissertation: even with the current program analysis techniques combined into a coherent Cyber Reasoning System, serious limitations still exist. In the final work of my graduate studies, I explored the careful reintegration of human assistance into our analysis automation, in a way that addresses its limitations without compromising its scalability advantage over manual analysis. I present this work in Chapter 5.

My journey was enlightening, rewarding, and humbling. In the course of my graduate studies, I saw the first staggering steps toward the realization of autonomy in program analysis. I finish the PhD program with a newfound sense of the profound complexity of understanding, but also with the hope of supporting future researchers in their endeavors to push my work ever-forward. Perhaps there is another kid, somewhere out there, growing tired of his digital dollhouse.

1.1 Permissions and Attributions

1. The content of chapters 2 and 3 is the result of a collaboration with Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna, and has previously appeared in the 2016 edition of the IEEE Symposium on Security and

Privacy.

2. The content of chapter 4 is the result of a collaboration with Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna, and has previously appeared in the 2015 edition of the Network and Distributed Systems Security Symposium.

3. The content of chapter 5 is the result of a collaboration with Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna, and will appear in the 2017 edition of the ACM Conference on Computer and Communications Security.

Chapter 2

Setting the Stage

Program analysis began quite soon after the invention of computing itself. The first device widely considered as conforming to some modern definition of a computer is the Analytical Engine, developed by Charles Babbage in the late 1830s. Shortly thereafter, Ada Lovelace published a set of notes about the Analytical Engine and proposed the first known "complex" program for it [2]. In one of these notes, Note G, Ada Lovelace provided an example of an execution of her program. At each instruction in the example, Ada produced an arithmetic expression, in terms of the input variables into the program, of each modified variable. I consider this, from a "popular science" viewpoint, to be the first symbolic trace in history. Thus, the underpinnings of, for example, symbolic-assisted fuzzing, can be traced back to 1842.

Further developments in program analysis were made over a hundred years later, when Alan Turing proposed that we might be able to develop a principled way to check programs for bugs [3]. While reading this work in the course of writing this dissertation, I was struck by the pronoun used for the "program checker" in Turing's paper: *he*, not *it*. In the 1940s, program analysis was a manual task, done by humans. In fact, in what can be viewed as the invention of (manual) fuzzing in the 1950s, programmers began to "test programs by inputting decks of punch cards taken from the trash" [4].

Eventually, automation made its way into the field. In 1975, the first paper describing symbolic execution, in a system called SELECT, was published [5]. 1977 saw the invention of Abstract Interpretation, providing a great boon to static analysis [6]. Finally, the field of fuzzing saw automation in 1981 [7].

Program analysis has continued to be an active area of research in the decades since. Techniques, prototypes, frameworks, and entire commercial products have been proposed, developed, and abandoned long before I started the research in this dissertation.

In this section, I will set the stage for the state of research, as it was when I started my work. Through the rest of my dissertation, I will discuss the contributions that I made to this state of the art, and will talk about what is left to do.

Bug-hunting vs Program Verification. The automated analysis of programs can be used toward a number of purposes, including the automatic identification of vulnerabilities in software, which is my field of interest. However, it is important to differentiate this goal from *Program Verification*, which is the assurance of the correctness of a program. In short, vulnerability identification seeks to be able to say "This program is unsafe.", while program verification seeks to say "This program is safe." In both cases, the lack of a positive result does not imply a negative result. Approaches for vulnerability discovery cannot guarantee that a program is safe if they do not find a bug (due to the lack of soundness leading to false negatives), and program verification approaches cannot guarantee that a program is buggy if they cannot verify its safety (due to over-approximation leading to false positives).

In this chapter, I will mostly talk about techniques for vulnerability detection, although the static techniques I discuss (many of which are implemented by angr) can support program verification tasks.

2.1 Analysis Trade-offs

It is not hard to see why binary analysis is challenging: in a sense, asking “will it crash?” is analogous to asking “will it stop?”, and any such analysis quickly runs afoul of the halting problem [8]. Program analyses, and especially offensive binary analyses, tend to be guided by carefully balanced theoretical trade-offs to maintain feasibility. For example, we can explore two such trade-offs:

Replayability. Bugs are not all created equal. Depending on the trade-offs made by the system, bugs discovered by a given analysis might not be *replayable*. This boils down to the *scope* that an analysis operates on. Some analyses execute the whole application, from the beginning, so they can reason about what *exactly* needs to be done to trigger a vulnerability. Other systems analyze individual pieces of an application: they might find a bug in a specific module, but cannot reason about how to *trigger* the execution of that module, and therefore, cannot automatically *replay* the crash.

Semantic awareness. Some analyses lack the ability to reason about the program in semantically meaningful ways. For example, a dynamic analysis might be able to trace the code executed by an application but not understand *why* it was executed or *what parts* of the input caused the application to act in that specific way. On the other hand, a symbolic analysis that can determine the specific bytes of input responsible for certain program behaviors would have a higher semantic understanding.

In order to offer replayability of input or semantic insight, analysis techniques must make certain trade-offs. For example, high replayability is associated with low coverage. This is intuitive: since an analysis technique that produces replayable input must understand how to reach any code that it wants to analyze, it will be unable to analyze as much code as an analysis that does not. On the other hand, without being able to replay triggering inputs to validate bugs, analyses that do not prioritize bug replayability suffer from a high level of *false positives* (that is, flaw detections that do not represent actual vulnerabilities). In the absence of a replayable input, these false positives must be filtered by heuristics which can, in turn, introduce false negatives.

Likewise, in order to achieve semantic insight into the program being analyzed, an analysis must store and process a large amount of data. A semantically-insightful dynamic analysis, for example, might store the conditions that must hold in order for specific branches of a program to be taken. On the other hand, a static analysis tunes semantic insight through the chosen data domain – simpler data domains (i.e., by tracking *ranges* instead of actual values) represent less semantic insight.

Analyses that attempt both reproducibility and a high semantic understanding encounter issues with scalability. Retaining semantic information for the entire application, from the entry point through all the actions it might take, requires a processing capacity conceptually identical to the resources required to execute the program under all possible conditions. Such analyses do not scale, and, in order to be applicable, must discard information and sacrifice *soundness* (that is, the guarantee that all potential vulnerabilities will be discovered).

Aside from these fundamental trade-offs, there are also implementation challenges. The biggest one of these is the *environment model*. Any analysis with a high semantic understanding must model the application's interaction with its environment. In modern operating systems, such interactions are incredibly complex. For example, modern versions of Linux include over three hundred system calls, and for an analysis system to be complete, it must model the effects of all of them.

2.2 Static Vulnerability Discovery

Static techniques reason about a program without executing it. Usually, a program is interpreted over an *abstract domain*. Memory locations containing bits of ones and zeroes contain other abstract entities (at the familiar end, this might simply be integers, but, as we explain below, these can include more abstract constructs). Additionally, program constructs such as the layout of memory, or even the execution path taken, may be abstracted as well.

Here, we split static analyses into two paradigms: those that model program properties as graphs (i.e., a *control-flow graph*) and those that model the data itself.

Static vulnerability identification techniques have two main drawbacks, relating to the trade-offs discussed in Section 2.1. First, the results are not *replayable*: detection by static analysis must be verified by hand, as information on *how* to trigger the detected vulnerability is not recovered. Second, these analyses tend to operate on simpler data domains, reducing their *semantic insight*. In short, they over-approximate: while they can often authoritatively reason about the *absence* of certain program properties (such as vulnerabilities), they suffer from a high rate of false positives when making statements regarding the *presence* of vulnerabilities.

2.2.1 Vulnerability Detection with Flow Modeling

Some vulnerabilities in a program can be discovered through the analysis of graphs of program properties.

Graph-based vulnerability discovery. Program property graphs (*e.g.*, control-flow graphs, data-flow graphs and control-dependence graphs) can be used to identify vulnerabilities in software. Initially applied to source code [9, 10], related techniques have since been extended to binaries [11]. These techniques rely on building a model of a bug, as represented by a set of nodes in a control-flow or data-dependency graph, and identifying occurrences of this model in applications. However, such techniques are geared toward searching for copies of vulnerable code, allowing the techniques to benefit from the preexisting knowledge of an already existing vulnerability. Unlike these techniques, the focus of this chapter is on the discovery of completely new vulnerabilities.

2.2.2 Vulnerability Detection with Data Modeling

Static analysis can also reason over abstractions of the data upon which an application operates.

Value-Set Analysis. One common static analysis approach is *Value-Set Analysis* (VSA) [12]. At a high level, VSA attempts to identify a tight over-approximation of the program state (*i.e.*, values in memory and registers) at any given point in the program. This can be used to understand the possible targets of indirect jumps or the possible targets of memory write operations. While these approximations suffer from a lack of accuracy, they are *sound*. That is, they may over-approximate, but never under-approximate.

By analyzing the approximated access patterns of memory reads and writes, the locations of variables and buffers can be identified in the binary. Once this is done, the recovered variable and buffer locations can be analyzed to find *overlapping* buffers. Such overlapping buffers can be, for example, caused by buffer overflow vulnerabilities, so each detection is one potential vulnerability.

2.3 Dynamic Vulnerability Discovery

Dynamic approaches are analyses that examine a program's execution, in an actual or emulated environment, as it acts given a specific input. In this section, we will focus specifically on dynamic techniques that are used for identifying vulnerabilities, rather than the general binary analysis techniques on which they are based.

Dynamic techniques are split into two main categories: concrete and symbolic execution. These techniques produce inputs that are highly *replayable*, but vary in terms of *semantic insight*.

2.3.1 Dynamic Concrete Execution

Dynamic concrete execution is the concept of executing a program in a minimally-instrumented environment. The program functions as normal, working on the same *domain* of data on which it would normally operate (i.e., ones and zeroes). These analyses typically reason at the level of single paths (i.e., "what path did the program take when given this specific input"). As such, dynamic concrete execution requires *test cases* to be provided by the user. This is a problem, as with a large or unknown dataset (such as ours) such test cases are not readily available.

Fuzzing

The most relevant application of dynamic concrete execution to vulnerability discovery is fuzzing. Fuzzing is a dynamic technique in which malformed input is provided to an application in an attempt to trigger a crash. Initially, such input was generated by hardcoded rules and provided to the application with little in-depth monitoring of the execution [13]. If the application crashed when given a specific input, the input was considered to have triggered a bug. Otherwise, the input would be further randomly mutated. Unfortunately, fuzzers suffer from the *test case* requirement. Without carefully crafted test cases to mutate, a fuzzer has trouble exercising anything but the most superficial functionality of a program.

Coverage-based fuzzing. The requirement for carefully-crafted test cases was partially mitigated with the advent of code-coverage-based fuzzing [14]. Code-coverage-based fuzzers attempt to produce inputs that maximize the amount of code executed in the target application based on the insight that the more code is executed, the higher the chance of executing vulnerable code. American Fuzzy Lop (AFL) [15], a state-of-the art fuzzer responsible for the discovery of many recent vulnerabilities, uses a code coverage metric as its sole guiding principle, and its success at finding vulnerabilities has driven an increase of interest in fuzzing in recent years.

Coverage-based fuzzing suffers from a lack of semantic insight into the target application. This means that, while it is able to detect that a certain piece of code has not yet been executed, it cannot understand what parts of the input to mutate to cause the code to be executed.

Taint-based fuzzing. Another approach to improve fuzzing is the development of *taint-based* fuzzers [16, 17]. Such fuzzers analyze how an application processes input to understand what parts of the input to modify in future runs. Some of these fuzzers combine taint tracking with static techniques, such as data dependency recovery [18, 19]. Others introduce work from protocol analysis to improve fuzzing coverage [20].

While a taint-based fuzzer can understand what parts of the input should be mutated to drive execution down a given path in the program, it is still unaware of *how* to mutate this input.

Dynamic Symbolic Execution

Symbolic techniques bridge the gap between static and dynamic analysis and provide a solution to cope with the limited semantic insight of fuzzing. Dynamic symbolic execution, a subset of symbolic execution, is a dynamic technique in the sense that it executes a program in an emulated environment. However, this execution occurs in the *abstract* domain of *symbolic variables*. As these systems emulate the application, they track the state of registers and memory throughout program execution and the *constraints* on those variables. Whenever a conditional branch is reached, execution forks and follows *both* paths, saving the branch condition as a constraint on the path in which the branch was taken and the inverse of the branch condition as a constraint on the path in which the branch was not taken [21].

Unlike fuzzing, dynamic symbolic execution has an extremely high semantic insight into the target application: such techniques can reason about *how* to trigger specific desired program states by using the accumulated path constraints to retroactively produce a proper input to the application when one of the paths being executed has triggered a condition in which the analysis is interested. This makes it an extremely powerful tool in identifying bugs in software and, as a result, dynamic symbolic execution is a very active area of research.

Classical dynamic symbolic execution. Dynamic Symbolic Execution can be used directly to find vulnerabilities in software. Initially applied to the testing of source code [22, 23], dynamic symbolic execution was extended to binary code by Mayhem [24] and S2E [25]. These engines analyze an application by performing path exploration until a vulnerable state (for example, the instruction pointer is overwritten by input from the attacker) is identified.

However, the trade-offs discussed in Section 2.1 come into play: all currently proposed symbolic execution techniques suffer from very limited scalability due to the problem of *path explosion*: because new paths can be created at every branch, the number of paths in a program increases exponentially with the number of branch instructions in every path. There have been attempts to survive path explosion by *prioritizing* promising paths [26, 27] and by *merging* paths where the situation is appropriate [28, 29, 30]. However, in general, this challenge to pure dynamic symbolic execution analysis engines has not yet been surmounted, and (as we show later in this chapter), most bugs discovered by such systems are *shallow*.

Symbolic-assisted fuzzing. One proposed way to address the path explosion problem is to offload much of the processing to faster techniques, such as fuzzing. This approach leverages the strength of fuzzing, *i.e.,* its speed, and attempts to mitigate the main weakness, *i.e.,* its lack of semantic insight into the application. Thus, researchers have paired fuzzing with symbolic execution [31, 32, 33, 34, 35, 36]. Such *symbolically-guided fuzzers* modify inputs identified by the fuzzing component by processing them in a dynamic symbolic execution engine. Dynamic symbolic execution uses a more in-depth understanding of the analyzed program to properly mutate inputs, providing additional test cases that trigger previously-unexplored code and allow the fuzzing component to continue making progress (i.e., in terms of code coverage).

Under-constrained symbolic execution. Another way to increase the tractability of dynamic symbolic execution is to execute only *parts* of an application. This approach, known as Under-constrained Symbolic Execution [37, 38], is effective at identifying *potential* bugs, with two drawbacks. First, it is not possible to ensure a proper context for the execution of parts of an application, which leads to many false positives among the results. Second, similar to static vulnerability discovery techniques, under-constrained symbolic execution gives up the replayability of the bugs it detects in exchange for scalability.

2.4 Simple Motivating Example

To demonstrate the various challenges of binary analysis, we provide a concrete example of a program with multiple vulnerabilities in Listing 1. For clarity and space reasons, this example is simplified, and it is meant only to expose the reader to ideas that will be discussed later in the chapter.

Observe the three calls to `memcpy`: the ones on lines 10 and 30 will result in buffer overflows, while the one on line 16 will not. However, depending on the amount of information tracked, a static analysis technique might report all three calls to `memcpy` as potential bugs, including the one on line 16, because it would not have the information to determine that no buffer overflow is possible. Additionally, while the report from a static analysis would include the locations of these bugs, it will not provide inputs to trigger them.

A dynamic technique, such as fuzzing, has the benefit of creating actionable inputs that will trigger any bugs found. On the other hand, simple fuzzing techniques typically only find shallow bugs and fail to pass through code requiring precisely crafted input. In Listing 1, dynamic techniques will have difficulty finding the bug at line 10, because it requires a specific input for the condition to be satisfied. However, because the overflow on line 30 can be triggered through random testing, fuzzing techniques should be able to find an input which triggers the bug.

To find the bug on line 10, one could introduce an abstract data model to reason about many possible inputs at once. One such approach is Dynamic Symbolic Execution (DSE). However, dynamic symbolic techniques, while powerful, suffer from the "path explosion problem", where the number of paths grows exponentially with each branch and quickly becomes intractable. A symbolic execution will detect the bug on line 10 and generate an input for it using a constraint solver. Additionally, it should be able to prove that the `memcpy` on line 16 cannot overflow. However, the execution will likely not be able to find the bug at line 30, as there are too many potential paths which do not trigger the bug.

```
1  int main(void) {
2    char buf[32];
3
4    char *data = read_string();
5    unsigned int magic = read_number();
6
7    // difficult check for fuzzing
8    if (magic == 0x31337987) {
9      // buffer overflow
10     memcpy(buf, data, 100);
11   }
12
13   if (magic < 100 && magic % 15 == 2 &&
14       magic % 11 == 6) {
15     // Only solution is 17; safe
16     memcpy(buf, data, magic);
17   }
18
19   // Symbolic execution will suffer from
20   // path explosion
21   int count = 0;
22   for (int i = 0; i < 100; i++) {
23     if (data[i] == ’Z’) {
24       count++;
25     }
26   }
27
28   if (count >= 8 && count <= 16) {
29     // buffer overflow
30     memcpy(buf, data, count*20);
31   }
32
33   return 0;
34 }
```

Listing 1: An example where different techniques will report different bugs.

2.5 The DARPA Cyber Grand Challenge

Historically, evaluating the effectiveness of proposed techniques in program analysis has been a tough problem. This was caused by several reasons, among which were the difficulty of building support for complex real-world programs into analysis engines and the lack of a standardized dataset for evaluation.

In October of 2013, DARPA announced the DARPA Cyber Grand Challenge [39]. Like DARPA Grand Challenges in other fields (such as robotics and autonomous vehicles), the CGC pitted teams from around the world against each other in a competition in which all participants must be autonomous programs. A participant's goal in the Cyber Grand Challenge is straight-forward: their system must autonomously identify, exploit, and patch vulnerabilities in the provided software.

The organizers of the Cyber Grand Challenge put much thought into designing a competition for automated binary analysis systems. For example, they addressed the *environment model* problem by creating a new OS specifically for the CGC: the DECREE OS. DECREE is an extremely simple operating system with just 7 system calls: `transmit`, `receive`, and `waitfd` to send, receive, and wait for data over file descriptors, `random` to generate random data, `allocate` and `deallocate` for memory management, and `terminate` to exit.

Despite the simple environment model, the binaries provided by DARPA for the CGC have a wide range of complexity. They range from 4 kilobytes to 10 megabytes in size, and implement functionality ranging from simple echo servers, to web servers, to image processing libraries. DARPA has open-sourced all the binaries used in the competition thus far, complete with proof-of-concept exploits and write-ups about the vulnerabilities [40].

Because the simple environment model and ready presence of ground truth makes it feasible to accurately implement and evaluate diverse binary analysis techniques, and because the binaries vary greatly in size and complexity and attempt to model real security flaws, the CGC dataset has provided the community with one of the first tailor-made experimental datasets for binary analysis. As such, the work presented in this dissertation is primarily evaluated on CGC binaries.

Chapter 3

A Principled Base for Cyber-autonomy

Despite the rise of interpreted languages and the World Wide Web, binary analysis has remained an extremely important topic in computer security. There are several reasons for this. First, interpreted languages are either interpreted by binary programs or Just-In-Time (JIT) compiled down to binary code. Second, “core” OS constructs and performance-critical applications are still written in languages (usually, C or C++) that compile down to binary code. Third, the rise of the Internet of Things is powered by devices that are, in general, very resource-constrained. Without cycles to waste on interpretation or Just-In-Time compilation, the firmware of these devices tends to be written in languages (again, usually C) that compile to binary.

Unfortunately, many low-level languages provide few security guarantees, often leading to vulnerabilities. For example, buffer overflows stubbornly remain as one of the most-common software flaws despite a concerted effort to develop technologies to mitigate such vulnerabilities. Worse, the wider class of “memory corruption vulnerabilities”, the vast majority of which also stem from the use of unsafe languages, make up a substantial portion of the most common vulnerabilities [41]. This problem is not limited to software on general-purpose computing devices: remotely-exploitable vulnerabilities have been discovered in devices ranging from smartlocks, to pacemakers, to automobiles [42].

Another important aspect to consider is that compilers and tool chains are not bug-free. Properties that were proven by analyzing the source code of a program may not hold after the very same program has been compiled [43]. This happens in practice: recently, a malicious version of Xcode, known as Xcode Ghost [44], silently infected over 40 popular iOS applications by inserting malicious code at compile time, compromising the devices of millions of users. These vulnerabilities have serious, real-world consequences, and discovering them before they can be abused is paramount. To this end, the security research community has invested a substantial amount of effort in developing analysis techniques to identify flaws in binary programs [45]. Such "offensive" (because they find "attacks" against the analyzed application) analysis techniques vary widely in terms of the approaches used and the vulnerabilities targeted, but they suffer from two main problems.

First, many implementations of binary analysis techniques begin and end their existence as a research prototype. When this happens, much of the effort behind the contribution is wasted, and future researchers must often start from scratch in terms of implementation of work based upon these approaches. This startup cost discourages progress: every week spent re-implementing previous techniques is one less week devoted to developing novel solutions.

Second, as a consequence of the amount of work required to reproduce these systems and their frequent unavailability to the public, replicating their results becomes impractical. As a result, the *applicability* of individual binary analysis techniques relative to other techniques becomes unclear. This, along with the inherent complexity of modern operating systems and the difficulty to accurately and consistently model the applications' interaction with their environment, makes it extremely difficult to establish a common ground for comparison. Where comparisons do exist, they tend to compare systems with different underlying implementation details and different evaluation datasets.

In an attempt to mitigate the first issue, we have created `angr`, a binary analysis framework that integrates many of the state-of-the-art binary analysis techniques in the literature. We did this with the goal of systematizing the field and encouraging the development of next-generation binary analysis techniques by implementing, in an accessible, open, and usable way, effective techniques from current research efforts so that they can be easily compared with each other. `angr` provides building blocks for many types of analyses, using both static and dynamic techniques, so that proposed research approaches can be easily implemented and their effectiveness compared to each other. Additionally, these building blocks enable the *composition* of analyses to leverage their different strengths.

Over the last year, a solution has also been introduced to the second problem, aimed towards comparing analysis techniques and tools, with research reproducibility in mind. Specifically, DARPA organized the Cyber Grand Challenge, a competition designed to explore the current state of automated binary analysis, vulnerability excavation, exploit generation, and software patching. As part of this competition, DARPA wrote and released a corpus of applications that are specifically designed to present realistic challenges to automated analysis systems and produced the *ground truth* (labeled vulnerabilities and exploits) for these challenges. This dataset of binaries provides a perfect test suite with which to gauge the relative effectiveness of various analyses that have been recently proposed in the literature. Additionally, during the DARPA CGC qualifying event, teams around the world fielded automated binary analysis systems to attack and defend these binaries. Their results are public, and provide an opportunity to compare existing offensive techniques in the literature against the best that the competitors had to offer[1].

[1]The top-performing 7 teams each won a prize of 750,000 USD. We expect that, with this motivation, the teams fielded the best analyses that were available to them.

Our goal is to gain an understanding of the relative efficacy of modern offensive techniques by implementing them in our binary analysis system. In this chapter, we detail the implementation of a next-generation binary analysis engine, `angr`. We present several offensive analyses that we developed using these techniques (specifically, replications of approaches currently described in the literature) to reproduce results in the fields of vulnerability discovery, exploit replaying, automatic exploit generation, compilation of ROP shellcode, and exploit hardening. We also describe the challenges that we overcome, and the improvements that we achieved, by combining these techniques to augment their capabilities. By implementing them atop a common analysis engine, we can explore the differences in effectiveness that stem from the theoretical differences behind the approaches, rather than implementation differences of the underlying analysis engines. This has enabled us to perform a comparative evaluation of these approaches on the dataset provided by DARPA.

In short, we make the following contributions:

1. We reproduce many existing approaches in offensive binary analysis, in a single, coherent framework, to provide an understanding of the relative effectiveness of current offensive binary analysis techniques.

2. We show the difficulties (and solutions to those difficulties) of combining diverse binary analysis techniques and applying them on a large scale.

3. We open source our framework, `angr`, for the use of future generations of research into the analysis of binary code.

3.1 Analysis Engine

The analyses that we described in Chapter 2 were proposed at various times over the last several years, implemented with different technologies, and evaluated on disparate datasets with varying methodologies. This is problematic, as it makes it hard to understand the relative effectiveness of different approaches and their applicability to different types of applications.

To alleviate this problem, we have developed a flexible, capable, next-generation binary analysis system, `angr`, and used it to implement a selection of the analyses we presented in the previous sections. This section describes the analysis system, our design goals for it, and the impact that this design has had on the analysis of realistic binaries.

3.1.1 Design Goals

Our design goals for `angr` are the following:

Cross-architecture support. With the rise of embedded devices, often running ARM and MIPS processors, modern software is made for varying hardware architectures. This is a departure from the previous decade, where x86 support was enough for most analysis engines: a modern binary analysis engine must be able to perform cross-architecture analyses. Furthermore, 32-bit processors are no longer the standard; a modern analysis engine must support analysis of 64-bit binaries.

Cross-platform support. In a similar vein to cross-architecture support, a modern analysis system must be able to analyze software from different operating systems. This means that concepts specific to individual operating systems must be abstracted, and support for *loading* different executable formats must be implemented.

Support for different analysis paradigms. A useful analysis engine must provide support for the wide range of analyses described in earlier sections. This requires that the engine itself abstract away, and provide different types of memory models as well as data domains.

Usability. The purpose of angr is to provide a tool for the security community that will be useful in reproducing, improving, and creating binary analysis techniques. As such, we strove to keep angr's learning curve low and its usability high. angr is almost completely implemented in Python, with a concise, simple API that is easily usable from the IPython interactive shell [46]. While Python results in constant-time lower performance than other potential language choices, most binary analysis techniques suffer from *algorithmic* slowness, and the language-imposed performance impact is rarely felt. When language overhead *is* important, angr can run in the Python JIT engine, PyPy for a significant speed increase.

Our goal was for angr to allow for the reproduction of a typical binary analysis technique, on top of our platform, in about a week. In fact, we were able to reproduce Veritesting [28] in eight days, guided symbolic execution in a month, AEG [47] in a weekend, Q [48] in about three weeks, and under-constrained symbolic execution [38] in two days. It is hard to produce an implementation effort estimate for dynamic symbolic execution and value-set analysis, as we implemented those while building the system itself over two years.

In order to meet these design goals, we had to carefully build our analysis engine. We did this by creating a set of modular building blocks for various analyses, being careful to maintain strict separation between them to reduce the number of assumptions that higher-level parts of angr (such as the state representation) make about the lower-level parts (such as the data model). This makes it easier for us to mix and convert between analyses on-the-fly. We hope that it will also make it easier for other researchers to reuse individual modules of `angr`. In the next several sections, we discuss the technical design of each `angr` submodule.

3.1.2 Submodule: Intermediate Representation

In order to support multiple architectures, we translate architecture-specific native binary code into an intermediate representation (IR) atop which we implement the analyses. Rather than writing our own "IR lifter", which is an extremely time-consuming engineering effort, we leveraged libVEX, the IR lifter of the Valgrind project. libVEX produces an IR, called VEX, that is specifically designed for program analysis. We used PyVEX, which we originally wrote for Firmalice [49], to expose the VEX IR to Python. By leveraging VEX, we can provide analysis support for 32-bit and 64-bit versions of ARM, MIPS, PPC, and x86 (with the 64-bit version of the latter being amd64) processors. Improvements are constantly being made by Valgrind contributors, with, for example, a port to the SPARC architecture currently underway.

As we will discuss later, there is no fundamental restriction for `angr` to always use VEX as its IR. As implemented, supporting a different intermediate representation would be a straightforward engineering effort.

3.1.3 Submodule: Binary Loading

The task of loading an application binary into the analysis system is handled by a module called `CLE`, a recursive acronym for `CLE Loads Everything`. `CLE` abstracts over different binary formats to handle loading a given binary and any libraries that it depends on, resolving dynamic symbols, performing relocations, and properly initializing the program state. Through `CLE`, `angr` supports binaries from most POSIX-compliant systems (Linux, FreeBSD, etc.), Windows, and the DECREE OS created for the DARPA Cyber Grand Challenge.

`CLE` provides an extensible interface to a binary loader by providing a number of base classes representing binary objects (i.e., an application binary, a POSIX `.so`, or a Windows `.dll`), segments and sections in those objects, and symbols representing locations inside those sections. `CLE` uses file format parsing libraries (specifically, `elftools` for Linux binaries and `pefile` for Windows binaries) to parse the objects themselves, then performs the necessary relocations to expose the memory image of the *loaded* application.

3.1.4 Submodule: Data Model

The *values* stored in the registers and memory of a `SimState` are represented by abstractions provided by another module, `Claripy`.

`Claripy` abstracts all values to an internal representation of an *expression* that tracks all operations in which it is used. That is, the expression x, added to the expression 5, would become the expression $x + 5$, maintaining a link to x and 5 as its arguments. These expressions are represented as “expression trees” with values being the leaf nodes and operations being non-leaf nodes.

At any point, an expression can be translated into data domains provided by `Claripy`’s *backends*. Specifically, `Claripy` provides backends that support the concrete domain (integers and floating-point numbers), the symbolic domain (symbolic integers and symbolic floating point numbers, as provided by the Z3 SMT solver [50]), and the value-set abstract domain for Value Set Analysis [12]. `Claripy` is easily extensible to other backends. Specifically, implementing other SMT solvers would be interesting, as work has shown that different solvers excel at solving different types of constraints [51].

User-facing operations, such as interpreting the constructs provided by the backends (for example, the symbolic expression $x + 1$ provided by the Z3 backend) into Python primitives (such as possible integer solutions for $x + 1$ as a result of a constraint solve) are provided by *frontends*. A frontend augments a backend with additional functionality of varying complexity. `Claripy` currently provides several frontends:

FullFrontend. This frontend exposes symbolic solving to the user, tracking constraints, using the Z3 backend to solve them, and caching the results.

CompositeFrontend. As suggested by KLEE and Mayhem, splitting constraints into independent sets reduces the load on the solver. The CompositeFrontend provides a transparent interface to this functionality.

LightFrontend. This frontend does not support constraint tracking, and simply uses the VSA backend to interpret expressions in the VSA domain.

ReplacementFrontend. The ReplacementFrontend expands the LightFrontend to add support for *constraints* on VSA values. When a constraint (i.e., $x + 1 < 10$) is introduced, the ReplacementFrontend analyzes it to identify bounds on the variables involved (i.e., $0 <= x <= 8$). When the ReplacementFrontend is subsequently consulted for possible values of the variable x, it will intersect the variable with the previously-determined range, providing a more accurate result than VSA would otherwise be able to produce.

HybridFrontend. The HybridFrontend combines the FullFrontend and the ReplacementFrontend to provide fast approximation support for symbolic constraint solving. While Mayhem [24] hinted at such capability, to our knowledge, `angr` is the first publicly available tool to provide this capability to the research community.

This modular design allows `Claripy` to combine the functionalities provided by the various data domains in powerful ways and to expose it to the rest of `angr`.

3.1.5 Submodule: Full-Program Analysis

Program state representation. The `angr` module is responsible for representing the *program state* (that is, a snapshot of values in registers and memory, open files, *etc.*). The state, named `SimState` in `angr` terms, is implemented as a collection of *state plugins*, which are controlled by *state options* specified by the user or analysis when the state is created. Currently, the following state plugins exist:

Registers. `angr` tracks the values of registers at any given point in the program as a state plugin of the corresponding program state.

Symbolic memory. To enable symbolic execution, `angr` provides a symbolic memory model as a state plugin. This implements the indexed memory model proposed by Mayhem [24].

Abstract memory. The abstract memory state plugin is used by static analyses to model memory. Unlike symbolic memory, which implements a continuous indexed memory model, the abstract memory provides a *region*-based memory model which is used by most static analyses.

POSIX. When analyzing binaries for POSIX-compliant environments, `angr` tracks the *system state* in this state plugins. This includes, for example, the files that are open in the symbolic state. Each file is represented as a memory region and a symbolic position index.

History. `angr` tracks a log of everything that is done to the state (i.e., memory writes, file reads, etc.) in this plugin.

Inspection. `angr` provides a powerful debugging interface, allowing breakpoints to be set on complex conditions, including taint, exact expression makeup, and symbolic conditions. This interface can also be used to *change* the behavior of `angr`. For example, memory reads can be instrumented to emulate memory-mapped I/O devices.

Solver. The Solver is a plugin that exposes an interface to different data domains, through the data model provider (`Claripy`, discussed below). For example, when this plugin is configured to be in `symbolic` mode, it interprets data in registers, memory, and files symbolically and tracks path constraints as the application is analyzed.

Architecture. The architecture plugin provides architecture-specific information that is useful to the analysis (i.e., the name of the stack pointer, the wordsize of the architecture, etc). The information in this plugin is sourced from the `archinfo` module, that is also distributed as part of `angr`.

These state plugins provide building blocks that can be combined in various ways to support different analyses.

Additionally, `angr` implements the base unit of an analysis: the `SimEngine` abstraction allows the implementation of different techniques to take an *input* state, apply the effects of a block of code under some domain, and generate an *output* state (or a set of output states, in case we encounter a block from which multiple output states are possible, such as a conditional jump). Again, this part of `angr` is modular: in addition to a `SimEngine` to process the VEX translations of basic blocks, `angr` currently allows the user to provide a handcrafted state-modification function written in Python (termed a `SimProcedure`), providing a powerful way to instrument blocks with Python code. In fact, this is how we implement our environment model: system calls are implemented as Python functions that modify the program state.

The analyst-facing part of `angr` provides complete analyses, such as dynamic symbolic execution and control-flow graph recovery. The "entry point" into these analyses is the `Project`, representing a binary with its associated libraries. From this object, all of the functionality of the other submodules can be accessed (i.e., creating states, examining shared objects, retrieving intermediate representation of basic blocks, hooking binary code with Python functions, etc.). Additionally, there are two main interfaces for full-program analysis: Path Groups and Analyses.

Simulation Manager. A `SimulationManager` is an interface to dynamic symbolic execution – it tracks paths as they run through an application, split, or terminate.The creation of this interface stemmed from frustration with the management of paths during symbolic execution. Early in `angr`'s development, we would implement ad-hoc management of paths for each analysis that would use symbolic execution. We found ourselves re-implementing the same functionality: tracking the hierarchy of paths as they split and merge, analyzing which paths are interesting and should be prioritized in the exploration, and understanding which paths are not promising and should be terminated. We unified the common actions taken on groups of paths, creating the `SimulationManager` interface. Furthermore, we designed customizable plugins, called `ExplorationTechniques`, that can change certain behavior of `SimulationManager` objects (path prioritization, pruning, merging, etc).

Analyses. `angr` provides an abstraction for any full program analysis with the `Analysis` class. This class manages the lifecycle of static analyses, such as control-flow graph recovery, and complex dynamic analyses as those presented in Section 3.5.

When `angr` identifies some *truth* about a binary (i.e., "the basic block at address X can jump to the basic block at address Y"), it stores it in the *knowledge base* of the corresponding Project. This shared knowledge base allows analyses to collaboratively discover information about the application.

3.1.6 Open-Source Release

We started to work on `angr` with the goal of developing a platform on which we could implement new binary analysis approaches. As we faced the unexpected challenges associated with the analysis of realistic binaries, we realized that such an analysis engine would be extremely useful to the security community. We have open-sourced `angr` in the hope that it will provide a basis for the future of binary analysis, and it will free researchers from the burden of having to re-address the same challenges over and over. `angr` is implemented in just over $70,000$ lines of code, usable directly from the IPython shell or as a python module, and easily installable via the standard Python package manager, `pip`.

The open-source release of `angr` includes the analysis engine modules (as described in Sections 3.1.1 through 3.1.5) on top of which we implemented the applications discussed in Section 3.7. Of the latter, we have open-sourced our control-flow graph recovery, the static analysis framework, our dynamic symbolic execution engine, and the under-constrained symbolic execution implementation. While we plan to release the other applications in the future, they are currently in a state that is a mix of being prototype-level code and being actively applied toward the DARPA Cyber Grand Challenge.

`angr` has been met with extreme enthusiasm by the community. In the first 2 years after the open-source release, we gathered about 1700 "stars" (measures of persons valuing the software) on GitHub across the different modules that make up the system. `angr` has been used in industry prototypes and many a research papers around the world, and is becoming a standard part of security courses. More importantly, we are starting to see significant contributions back to the project from the community at large, enabling `angr` to function on more targets more effectively.

3.2 IR Translation

Because software is made for devices with widely diverse architectures, binary analysis systems must be able to carry out their analysis in the context of many different hardware platforms. To address this challenge, Firmalice translates the machine code of different architectures into an intermediate representation, or *IR*. The IR must abstract away several architecture differences when dealing with different architectures:

Register names. The quantity and names of registers differ between architectures, but modern CPU designs hold to a common theme: each CPU contains several general purpose registers, a register to hold the stack pointer, a set of registers to store condition flags, and so forth. The IR must provide a consistent, abstracted interface to registers on different platforms.

Memory access. Different architectures access memory in different ways. For example, ARM can access memory in both little-endian and big-endian modes. The IR must be able to abstract away these differences.

Memory segmentation. Some architectures, such as x86, which is beginning to be used in embedded applications, support memory segmentation through the use of special segment registers. The chosen IR needs to be able to model such memory access mechanisms.

Instruction side-effects. Most instructions have side-effects. For example, most operations in Thumb mode on ARM update the condition flags, and stack push/pop instructions update the stack pointer. Tracking these side-effects in an *ad hoc* manner in the analysis would be error-prone, so the IR should make these effects explicit.

There are many existing intermediate representations available for use, including REIL [52], LLVM IR [53], and VEX, the IR of the Valgrind project [54]. We decided to utilize VEX due to its ability to address our IR requirements and an active and helpful developer community. However, our approach would work with any intermediate representation. To reason about VEX IR in Python, we implemented Python bindings for libVEX. We have open-sourced these bindings [55] in the hope that they will be useful for the community.

VEX is an architecture-agnostic representation of a number of target machine languages, of which the x86, AMD64, PPC, PPC64, MIPS, MIPS64, ARM (in both ARM and Thumb mode), ARM64, and S390X architectures are supported. VEX abstracts machine code into a representation designed to make program analysis easier by modeling instructions in a unified way, with explicit modeling of all instruction side-effects. This representation has four main classes of objects.

Expressions. IR Expressions represent a calculated or constant value. This includes values of memory loads, register reads, and results of arithmetic operations.

Operations. IR Operations describe a *modification* of IR Expressions. This includes integer arithmetic, floating-point arithmetic, bit operations, and so forth. An IR Operation applied to IR Expressions yields an IR Expression as a result.

Temporary variables. VEX uses "temporary variables" as internal registers: IR Expressions are stored in temporary variables between use. The content of a temporary variable can be retrieved using an IR Expression.

Statements. IR Statements model changes in the state of the target machine, such as the effect of memory stores and register writes. IR Statements use IR Expressions for values they may need. For example, a memory store statement uses an IR Expression for the target address of the write, and another IR Expression for the content.

Blocks. An IR Block is a collection of IR Statements, representing an extended basic block in the target architecture. A block can have several exits. For conditional exits from the middle of a basic block, a special "Exit" IR Statement is used. An IR Expression is used to represent the target of the unconditional exit at the end of the block.

Relevant IR Expressions and IR Statements for an analysis are detailed in Tables 3.1 and 3.2.

The IR translation of an example ARM instruction is presented in Table 3.3. In the example, the subtraction operation is translated into a single IR block comprising 5 IR Statements, each of which contains at least one IR Expression. Register names are translated into numerical indices given to the *GET* Expression and *PUT* Statement. The astute reader will observe that the actual subtraction is modeled by the first 4 IR Statements of the block, and the incrementing of the program counter to point to the next instruction (which, in this case, is located at 0x59FC8) is modeled by the last statement.

3.3 CFG Recovery

We will describe the process that `angr` uses to generate a CFG, including specific techniques that were developed to improve the completeness and soundness of the final result.

Given a specific program, `angr` performs an iterative CFG recovery, starting from the entry point of the program, with some necessary optimizations. `angr` leverages a combination of *forced execution*, *backwards slicing*, and *symbolic execution* to recover, where possible, all jump targets of each indirect jump. Moreover, it generates and stores a large quantity of data about the target application, which can be used later in other analyses such as data-dependence tracking.

This algorithm has three main drawbacks: it is slow, it does not automatically handle "dead code", and it may miss code that is only reachable through unrecovered indirect jumps. To address this issue, we created a secondary algorithm that uses a quick disassembly of the binary (without executing any basic block), followed by heuristics to identify functions, intra-function control flow, and direct inter-function control flow transitions. The secondary algorithm, however, is much less accurate – it lacks information about reachability between functions, is not context sensitive, and is unable to recover complex indirect jumps.

In the reminder of this section, we discuss our advanced recovery algorithm, which we dub `CFGAccurate`. We then discuss our fast algorithm, `CFGFast`, in Section 3.3.7.

3.3.1 Recovering Control Flow

The recovery of a *control-flow graph* (CFG), in which the nodes are basic blocks of instructions and the edges are possible control flow transfers between them, is a prerequisite for almost all static techniques for vulnerability discovery.

Control-flow recovery has been widely discussed in the literature [56, 57, 58, 59, 60, 61]. CFG recovery is implemented as a recursive algorithm that disassembles and analyzes a basic block (say, B_a), identifies its possible *exits* (i.e., some successor basic block such as B_b and B_c) and adds them to the CFG (if they have not already been added), connects B_a to B_b and B_c, and repeats the analysis recursively for B_b and B_c until no new exits are identified. CFG recovery has one fundamental challenge: indirect jumps. Indirect jumps occur when the binary transfers control flow to a target represented by a value in a register or a memory location. Unlike a *direct* jump, where the target is encoded into the instruction itself and, thus, is trivially resolvable, the target of an indirect jump can vary based on a number of factors. Specifically, indirect jumps fall into several categories:

Computed. The target of a computed jump is determined by the application by carrying out a calculation specified by the code. This calculation could further rely on values in other registers or in memory. A common example of this is a jump table: the application uses values in a register or memory to determine an index into a jump table stored in memory, reads the target address from that index, and jumps there.

Context-sensitive. An indirect jump might depend on the context of an application. The common example is `qsort()` in the standard C library – this function takes a *callback* that it uses to compare passed-in values. As a result, some of the jump targets of basic blocks inside `qsort()` depend on its caller, as the caller provides the callback function.

Object-sensitive. A special case of context sensitivity is object sensitivity. In object-oriented languages, object polymorphism requires the use of virtual functions, often implemented as *virtual tables* of function pointers that are consulted, at runtime, to determine jump targets. Jump targets thus depend on the type of object passed into the function by its callers.

Different techniques have been designed to deal with different types of indirect jumps, and we will discuss the implementation of several of them in Section 3.3. In the end, the goal of CFG recovery is to *resolve* the targets of as many of these indirect jumps as possible, in order to create a CFG. A given indirect jump might resolve to a *set* of values (*i.e.*, all of the addresses in a jump table, if there are conditions under which their use can be triggered), and this set might change based on both object and context sensitivity. Depending on how well jump targets are resolved, the CFG recovery analysis has two properties:

Soundness. A CFG recovery technique is *sound* if the set of all potential control flow transfers is represented in the graph generated. That is, when an indirect jump is resolved to a *subset* of the addresses that it can actually target, the soundness of the graph decreases. If a potential target of a basic block is missed, the block it targets might never be seen by the CFG recovery algorithm, and any direct and indirect jumps made by that block will be missed as well. This has a cumulative effect: the failure to resolve an indirect jump might severely reduce the completeness of the graph. Soundness can be thought of as the *true positive* rate of indirect jump target identification in the binary.

Completeness. A complete CFG recovery builds a CFG in which all edges represent actually possible control flow transfers. If the CFG analysis errs on the side of *completeness*, it will likely contain edges that cannot really exist in practice. Completeness can be thought of as the inverse of the *false positive* rate of indirect jump target identification.

A CFG recovery analysis that produces an empty graph would be considered *complete*, and an analysis that produces a graph in which every instruction points to every other instruction is considered *sound*. [2] While the ideal is somewhere in between, this is difficult to achieve with a scalable algorithm. Thus, different analyses require a different compromise between the two.

[2] Xu et. al. defines soundness and completeness of a CFG in the opposite way, where an empty graph is sound and a full graph is complete [56]. In this chapter, we stick to the definition in Section 3.3.1.

A further difficulty of control-flow graphs is accurately measuring *code coverage*, which is the measure of how much code is discovered by a control-flow graph. This is often complicated by the presence of *dead code*, code which is unreachable by any jumps.

3.3.2 Assumptions

`angr`'s `CFGAccurate` makes several assumptions about binaries to optimize the run time of the algorithm.

1. All code in the program can be distributed into different functions.
2. All functions are either called by an explicit call instruction (or its equivalents), or are preceded by a tail jump (an optimization, often used to reduce stack space for recursive functions, in which a call at the very end of a function is changed to a *jump* so that the newly called function simply reuses the return address of its caller) in the control flow.
3. The stack cleanup behavior of each function is predictable, regardless of where it is called from. This lets `CFGAccurate` safely skip functions that it has already analyzed while analyzing a caller function and keep the stack balanced.

These assumptions place constraints on the types of binaries that `angr` is designed to analyze. Assumptions 1, 2, and 3 require that the binary being analyzed is not obfuscated and behaves in a "normal" way. We *can* remove those assumptions when analyzing obfuscated or abnormal binaries, but this would lead to a higher run time of the CFG recovery.

Our CFG recovery code is built upon techniques proposed by related literature [56, 57, 58, 59, 60]. However, these techniques make assumptions that are overly strict or are unrealistic for real-world binaries. Specifically, we do *not* assume any of the following, unlike the work that our CFG recovery is based on:

1. All functions return to the next instruction after their call-site [56].
2. The jump target of an indirect branch is always determined by a control flow path, not by a program state or context [56]. For example, some existing literature assumes that indirect jumps are all *computed*, as opposed to being passed in as a function pointer from prior contexts.
3. Expressions for jump targets of indirect jumps must match a set of common idioms [57],[58]. Unlike existing work, we make no assumptions on the type of operations that can be applied to pointers.
4. The stack pointer is the same before entering a function as it is after returning from it.
5. No two functions overlap (in other words, they cannot share basic blocks [60].) `CFGAccurate` handles functions that share code.
6. Additional information, such as symbol tables or relocation information, is available [59].

The actual algorithm to recover a control-flow graph from a binary is described in the next few sections.

3.3.3 Iterative CFG Generation

Unfortunately, no single technique meets `CFGAccurate`'s goal of recovering a complete and sound CFG. Thus, `CFGAccurate` constructs a CFG by interleaving a series of techniques to achieve speed and completeness. Specifically, four techniques are used: forced execution, lightweight backward slicing, symbolic execution, and value set analysis. The CFG to be iteratively recovered by these techniques, C, is initialized with the basic block at the entry point of the application.

Throughout CFG recovery, `CFGAccurate` maintains a list of indirect jumps, L_j, whose jump targets have not been resolved. When the analysis identifies such a jump, it is added to L_j. After each iterative technique terminates, `CFGAccurate` triggers the next one in the list. This next technique may resolve jumps in L_j, may add new unresolved jumps to L_j, and may add basic blocks and edges to the CFG C. `CFGAccurate` terminates when a run of all techniques results in no change to L_j or C, as that means that no further indirect jumps can be resolved with any available analysis.

3.3.4 Forced Execution

`angr`'s `CFGAccurate` leverages the concept of Dynamic Forced Execution for the first stage of CFG recovery [56]. Forced Execution ensures that both directions of a conditional branch will be executed at every branch point.

`CFGAccurate` maintains a work-list of basic blocks, B_w, and a list of analyzed blocks, B_a. When the analysis starts, it initializes its work-list with all the basic blocks that are in C but not in B_a. Whenever `CFGAccurate` analyzes a basic block from this work-list, the basic block and any *direct* jumps from the block are added to C. Indirect jumps, however, cannot be handled this way. Under forced execution, the targets of indirect jumps may differ from those of an actual run of the program because forced execution will execute code in an unexpected order. Thus, each indirect jump is stored in the list L_j for later analysis.

As it cannot resolve any indirect jumps, this analysis functions as a fast-pass CFG recovery analysis to quickly seeds the other analyses with detected basic blocks and unresolved indirect jumps.

3.3.5 Symbolic Execution

The main issue with dynamic forced execution is the presence of indirect jumps, as there is no way to make sure that the target of an indirect jump is correctly resolved. On the one hand, an indirect jump may be completely unresolvable (i.e., the forced execution resulted in a state where the jump target is read from uninitialized memory), which leaves a broken control flow transition in the recovered CFG. On the other hand, an indirect jump may also be partially solvable (i.e. our analysis only retrieves a portion of all the possible jump targets).

For each jump $J \in L_j$, `CFGAccurate` traverses the CFG backwards until it find the first *merge point* (that is, multiple paths converging on the way to the indirect jump) or up to a threshold number of blocks (empirically, we found a reasonable threshold to be 8). From there, it performs forward symbolic execution to the indirect jump and uses a constraint solver to retrieve possible values for the target of the indirect jump.

`CFGAccurate` considers a jump successfully resolved if the computed set of possible targets is smaller than a threshold size. We use a value of 256 for this threshold but we have found that, in practice, in the cases where jumps are *not* resolved successfully, this value is *unconstrained* (meaning, the set of possible targets is bounded only by the number of bits in the address).

If the jump is resolved successfully, J is removed from L_j and edges and nodes are added to the CFG for each possible value of the jump target.

3.3.6 Backward Slicing

`angr`'s forced execution and symbolic execution analyses fail to resolve many of the unresolved jumps due to the lack of *context*. Those analyses are carried out in a context-insensitive manner: if a function takes pointer as an argument, and that pointer is used as the target of an indirect jump, the analyses will be unable to resolve it.

To achieve better completeness, our CFG generation requires a context-sensitive component. We accomplish this with *backward slicing*. `CFGAccurate` computes a backward slice starting from the unresolved jump. The slice is extended through the beginning of the previous *call context*. That is, if the indirect jump being analyzed is in a function F_a that is called from both F_b and F_c, the slice will extend backward from the jump in F_a and contain two start nodes: the basic block at the start of F_b and the one at the start of F_c.

`CFGAccurate` then executes this slice using `angr`'s symbolic execution engine and uses the constraint engine to identify possible targets of the symbolic jumps, with the same threshold of 256 for the size of the solution set for the jump target. If the jump target is resolved successfully, the jump is removed from L_j and the edge representing the control flow transition, and the target basic blocks are added to the recovered CFG.

3.3.7 CFGFast

The goal of the fast CFG generation algorithm is to generate a graph, with high code coverage, that identifies at least the location and content of functions in the binary. This graph lacks much of the *control flow*, so it is not complete. However, such a graph can still be useful for both manual and automated analysis of binaries.

`CFGFast` carries out the following steps:

Function identification. We use hard-coded function prologue signatures, which can be generated from techniques like ByteWeight [62], to identify functions inside the application. If the application includes *symbols*, specifying the locations of functions, they are also used to seed the graph with function start positions. Additionally, the basic block representing the entry point of the program is added to the graph.

Recursive disassembly. Recursive disassembly is used to recover the direct jumps within the identified functions.

Indirect jump resolution. Lightweight alias analysis, data-flow tracking, combined with pre-defined strategies are used to resolve intra-function control flow transfers. Currently `CFGFast` includes strategies for jump table identification and indirect call target resolution.

The goal is to quickly recover a CFG with a high coverage, without a concern for understanding the reachability of functions from one another.

3.3.8 Using the CFG Recovery

`angr` exposes the CFG recovery algorithms as two analyses: `CFGFast` and `CFGAccurate`. These analyses output CFG data to `angr`'s knowledge base, as discussed in Section 3.1.5. This data can then be used in the course of manual analysis or later automated analyses.

3.4 Value Set Analysis

Once a CFG is generated, more advanced analyses can be run. One of these is Value-Set Analysis [12]. Value-Set Analysis (VSA) is a static analysis technique that combines numeric analysis and pointer analysis for binary programs. It uses an abstract domain, called the Value-Set Abstract domain, for approximating possible values that registers or abstract locations may hold at each program point.

VSA analyzes a program until it reaches a *fix-point* for all program points in the function. This fix-point represents a tight over-approximation of all values that any register or abstract memory location can have at any point in the function. With respect to, for example, a memory write to a computed address A, consulting the values of A in the computed fix-point will contain a complete list of all possible write targets.

The original VSA design, proposed by Balakrishnan et al. [12], does not perform well when analyzing real-world binaries. To make VSA work on such binaries, we had to develop a number of improvements to increase the precision of our analysis.

Creating a discrete set of strided-intervals. The basic data type of VSA, the strided interval, is essentially an approximation of a set of numbers. It is great for approximating a set of normal concrete values. However, if those values are used as jump targets in the program, the over-approximating nature of strided-intervals yields unsoundness in our recovered CFG by creating control flow transitions to addresses that should not be jump targets. To effectively solve this problem, we developed a new data type called "strided interval set", which represents a set of strided intervals that are not unioned together. A strided interval set will be unioned into a single strided interval only when it contains more than K elements, where K is a threshold that can be adjusted. In our model discussed in Section 2.1, this threshold controls a trade-off of semantic insight versus scalability – a higher value of K allows us to maintain high precision, but comes at a cost of increased analysis complexity.

Applying an algebraic solver to path predicates. Tracking branch conditions helps us constrain variables in a state after taking a conditional exit or during a merging procedure, which produces a more precise analysis result. Affine-Relation Analysis has been proposed as a technique to track these conditions [63]. However, it is both complicated to implement (generally leading to support for very few arithmetic operations in constraint expressions), and is computationally expensive in reality.

Our solution is to implement a lightweight algebraic solver that works on the strided interval domain, based on modulo arithmetic which take care of some of the affine relations. When a new path predicate is seen (i.e., when following a conditional branch), we attempt to simplify and solve it to obtain a number range for the variables involved in the path predicate. Then we perform an intersection between the newly generated number range and the original values for each corresponding variable. This allows us to continuously refine the result of our value-set analysis as new branch conditions are encountered, increasing the precision of the eventual fix-point.

Adopting a signedness-agnostic domain. As originally proposed, VSA operates on a signed strided interval domain, which assumes all values are signed. That is, for an n-bit strided-interval with l as its lower bound and h as its upper bound, we always have $l \in [-2^{n-1}, 2^{n-1} - 1] \wedge h \in [-2^{n-1}, 2^{n-1} - 1] \wedge l \leq h$. However, this results in heavily over-approximated results of unsigned arithmetic calculations. In fact, this over-approximation is exacerbated by the fact that, since jump addresses are unsigned, the computation of jump addresses generally relies on unsigned values (i.e., in the case of unsigned comparisons). The solution to this problem is to adopt a signedness-agnostic domain for our analysis. *Wrapped Interval Analysis* [64] is such an interval domain for analyzing LLVM code, which takes care of signed and unsigned numbers at the same time. We based our signedness-agnostic strided-interval domain on this theory, applied to the VSA domain.

We use VSA for memory corruption detection in three phases. First, we collect all read and write access patterns in the program during the VSA. On top of those access patterns, we perform a variable recovery for variables on both the stack and heap regions. Our implementation is similar to the variable recovery in TIE [65]. Next, we scan all stack and heap regions to find abnormal buffers, including a) overlapping buffers, and b) out-of-bound buffers. Then we simply report all abnormal buffers as potential memory corruptions.

3.4.1 Using VSA

The main interface that `angr` provides into a full-program VSA analysis is the *Value Flow Graph*. The VFG is an enhanced CFG that includes the *program state* representing the VSA fix-point at each program location. Depending on the parameters passed to the VFG analysis, this can include a single function, a tree of function calls, or the entire program.

The program states contained in the VFG present memory in an abstract layout provided by `angr` (specifically, the `SimAbstractMemory` memory model), with values in memory represented by value-sets, as provided by `Claripy`. We performed our buffer overlap analysis over the data contained in these program states by analyzing the range of values that memory accesses may take.

3.5 Symbolic Execution

The dynamic symbolic execution module of our analysis platform is mainly based on the techniques described in Mayhem [24]. Our implementation follows the same memory model and path prioritization techniques. This module represents one of the core functionalities of `angr`, other analyses, such as Veritesting and under-constrained symbolic execution, use it as a base.

We use `Claripy`'s interface into Z3 to populate the symbolic memory model (specifically, `SimSymbolicMemory`) provided by `angr`. Individual execution states through a program are managed by `SimState` objects, provided by `angr`, which have plugins to track the actions taken by the program, the path predicates, and various other information. Groups of these paths are managed by `angr`'s `SimulationManager` functionality, which provides an interface for managing the splitting, merging, and filtering of paths during dynamic symbolic execution.

`angr` has built-in support for Veritesting [28], implementing it as a `Veritesting` analysis and exposing transparent support for it with an option passed to `Simulation-Manager` objects. This advanced state merging technique helps mitigate the problem of exponential state explosion by statically (and selectively) merging paths.

3.5.1 Under-Constrained Symbolic Execution

We implemented under-constrained symbolic execution (UCSE), as proposed in UC-KLEE [38], and dubbed it `UC-angr`. UCSE is a dynamic symbolic execution technique where execution is performed on each function separately. Since the analysis cannot reason about *how* to get to the specific function, detections by UCSE are not replayable. Because each function is generated without its *context* (i.e., the arguments and global variables with which it is called in actual executions), the analysis is not accurate and suffers from false positives.

UCSE tags missing context in the state as *under-constrained*. When such under-constrained data is used as a pointer, a new under-constrained region is created and the pointer is directed at the new region. This "on-demand" memory allocation enables code that manages complex data structures to be analyzed. When a security violation is identified (i.e., a write to the saved return address on the stack), the values involved are checked for their *under-constrained* status. Under certain conditions (i.e., if all data involved is under-constrained), the violation is filtered out as a false positive.

We made two changes to the technique described in UCSE:

Global memory under-constraining. The original UC-KLEE implementation does not treat access to global memory as under-constrained. However, such memory is part of the program context that is impossible to predict with UCSE, since, when analyzing a given function, global data could have already potentially been overwritten. Thus, we mark all global data as under-constrained, allowing us to lower our false positive rate.

Path limiters. The original UC-KLEE implementation had several built-in limitations to prevent a path explosion. For example, they would limit the depth of under-constrained pointer dereferences to avoid a search through an under-constrained linked list never terminating. We added an additional limiter: we abort the analysis of a function when we find that it is responsible for a path explosion. We detect this by hard-coding a limit (in our experiments, we used an empirically-determined limit of 64 paths) and, when a single function branches over this many paths, we replace the function with an immediate `return`, and rewind the analysis from the call site of that function. This keeps the analysis tractable by avoiding path explosions, but makes the analysis even less accurate.

False positive filtering. We introduced several additional false positive filters into our implementation of UC-`angr`. Specifically, when we detect an exploitable state, we attempt to ensure that the state is not incorrectly made exploitable by a lack of constraints on under-constrained data. First, we perform a constraint solve with an additional constraint, E, that expresses the fact that the state is *not* exploitable (i.e., if the security violation was an overwrite of the return address, we constrain the state so that the return address could *not* have been overwritten). Then, we constrain each under-constrained value to its possible solution from this unexploitable state. We call these constraints U. Finally, we remove the constraint E, keeping the constraints U, and check that the state can still be exploited. If it can, this means that the function likely has some inherent flaw, and the flaw does not necessarily depend on missing *data* from the context. Note that the flaw could still be a false positive due to missing *constraints*, or due to the limited context on data that is not under-constrained.

`UC-angr` is implemented as a `SimState` plugin that tracks under-constrained data accesses and carries out the required relocations. Once this plugin is initialized, under-constrained symbolic execution can be performed using the same `SimulationManager` paradigm as dynamic symbolic execution.

3.5.2 Symbolic-Assisted Fuzzing

While we give a summary of our symbolic-assisted fuzzing implementation here, the full approach, called `Driller`, is detailed in a separate paper [36].

Our implementation of symbolic-assisted fuzzing uses the AFL fuzzer as its foundation and `angr` as its symbolic tracer. By monitoring AFL's performance, we can decide when to begin symbolically-tracing the inputs that AFL has created. To make this decision, we act on the rate at which the fuzzer is discovering new state transitions. If AFL reports that it has discovered no new state-transitions after performing a round of mutations of the input, we assume the fuzzer to be having trouble making progress, and invoke `angr` on all paths AFL has deemed as *unique* (i.e., any path that has a jump, identified by a tuple of the source and destination address, that no other path has), looking for transitions that AFL was unable to find inputs for.

`Driller`'s symbolic component is implemented using `angr`'s symbolic execution engine, so as to symbolically trace paths based on the concrete inputs provided by AFL. This avoids the path explosion problem inherent to symbolic execution, as each concrete input corresponds to a single (traced) path, and these inputs are heavily filtered by AFL to ensure that only promising ones are traced. Each concrete input corresponds to an individual path in a `SimulationManager`. At each step of the `SimulationManager`, every branch is checked so as to ensure that the most recent jump instruction leads to a path previously unknown to AFL. When such a jump is found, the SMT solver is queried to create an input that would drive execution to the new jump. This input is fed back to AFL, which goes on to mutate it in future fuzzing steps. This feedback loop allows us to balance expensive symbolic execution time with cheap fuzzing time, and mitigates fuzzing's low semantic insight into program operation.

3.6 Exploitation

Vulnerability discovery analyses actually discover *crashing inputs*. Triaging these crashing inputs – that is, understanding which crashes represent actual security issues, is outside of the scope of most such approaches. However, some work does exist on the reproduction and analysis of the discovered vulnerabilities. In this section, we go through the process of reproducing an identified crash, automatically generating the exploit to verify the security impact of the crash, and hardening the exploit to make it resilient in the presence of modern mitigation techniques.

3.6.1 Background: Crash Reproduction

Most vulnerability discovery analyses execute a tested application in less-than-realistic conditions. For example, many fuzzers will *de-randomize* execution. That is, they will hard-code any sources of randomness, such as the PID of the executable, the current time, and so on. This is done for two main reasons. First, in most modern fuzzing approaches, there is an implicit assumption that the same input, provided to two instantiations of an application, will produce the same result both times. Second, the modeling of randomness in other techniques, such as dynamic symbolic execution, is not a well-explored research area.

Because of de-randomization, the crashes reported by vulnerability discovery techniques might not be trivially replayable outside of the analysis environment. Consider the case of an application that generates a random token and requires the token to be provided by the user before entering an unsafe section of code and crashing. In the de-randomized analysis environment, the generated token will always have the same value, and the crashing input identified by the analysis will always take the same path, resulting in a crash. However, *outside* of the analysis environment, the token will always be different, and the previously-crashing input might instead take a *non-crashing* path.

Crashing inputs that are not trivially replayable generally fall into two categories.

Missing data. Vulnerability discovery techniques sometimes manage to "guess" correct response values without having first received them from the application. The token in our example is always a constant value in the de-randomized environment, and an analysis engine such as a fuzzer might accidentally guess it without first retrieving it from the program. When replaying the resulting crashing input outside of the analysis environment, the token value will not match and the crash will not occur.

Missing relationships. Techniques with low semantic insight, such as fuzzing, are unable to recover the *relationships* between data retrieved from the program and subsequent data provided to it. In our example, even though the crashing input might cause the application to provide the token to the user, so it can later be used to cause the crash, the output of the fuzzer lacks the relationship between the token value that the application provides to the user and the token value that the user must provide to the application.

In the case of missing data, the input is simply not replayable outside of the analysis environment, and a new crashing input might be found. Analyses exist that specialize in the identification of data leaks [19], but we have not yet implemented such analyses in `angr`.

In the latter case, the de-randomized crashing input must be converted into an *input specification* that defines how to communicate with an application in terms of the relationship between data received from the application and data later provided to it. One approach that does this is Replayer [66], which computes preconditions on program paths to understand how to reproduce a program path under real-world conditions.

3.6.2 Background: Exploit Generation

With a productive vulnerability excavation engine utilizing one or more of the methods described above, many crashes might be produced for a tested application. However, not all of these crashes will be exploitable. An example of a non-exploitable input is a NULL-pointer dereference. Because modern operating systems disallow the mapping of memory at address 0, these previously-exploitable situations have been reduced to non-exploitable crashes. Understanding whether a crash is exploitable helps with the *triaging* of bugs (that is, understanding which bugs to investigate and fix first).

The obvious way to test if a crash would be exploitable is to try to exploit it. To this end, several systems have been proposed that attempt to take a crashing input and automatically convert it into an exploit for the application [67, 47, 68].

3.6.3 Background: Exploit Hardening

In recent years, binary hardening techniques, such as non-executable stack regions and Address Space Layout Randomization (ASLR), have severely reduced the efficacy of traditional exploits, such as those generated by first-generation automatic exploitation engines. Thus, even an exploitable vulnerability might be mitigated by modern protections.

Current automatic exploitation techniques were designed before the widespread adoption of modern mitigation techniques, and modern software protections make the exploits they produce non-functional. To circumvent this, approaches have been created to automatically *harden* the exploits generated using current techniques against such defenses. Such techniques work by translating a traditional, shellcode-based exploit into an equivalent exploit utilizing Return-Oriented Programming [69]. As such, an automatic approach to constructing Return-Oriented Programs is required, and several such approaches have been developed [70, 48].

3.6.4 Implementation: Crash Reproduction

We implemented the approach proposed by Replayer [66] to recover missing relationships between input values (i.e., values that the attacker sends) and output values (i.e., values that the attacker leaks from the application).

Our implementation of Replayer is built atop our symbolic execution engine. We can define the problem of replaying a crashing input as the search for an input specification i_s to bring a program from an initial state s to a crash state q. Our algorithm takes, as input, the program P, an initial state s_a (i.e., the state at the entry point of the executable), the crash state q_a, and the input i_a that brings s_a to q_a in the instrumented (de-randomized) environment, but does not properly replay in an uninstrumented environment. Our implementation symbolically executes the path from s_a to q_a, using the input i_a. It records all constraints that are generated while executing P. Given the constraints, the execution path, the program P, and the new initial state s_b, we can symbolically execute P with an *unconstrained* symbolic input, following the previously recorded execution path until the new crash state q_b is reached. At this point, the input constraints on the input and output can be analyzed, and relationships between them can be recovered. This relationship data is used to generate the input specification i_s, allowing the crashing input to be replayed.

The implementation proposed by Replayer has two main limitations in its application to crash reproduction. First, as we discuss in Section 3.6.1, it is possible that a given crash does not retrieve all of the data that is required to properly replay the crash. Replayer is unable to handle these cases, and new crashing inputs must be found.

Second, Replayer uses only the exact path, as executed by the application in the de-randomized environment while processing the crashing input, to generate the input specification. If the execution trace of a binary changes, based on the exact value of random data, then Replayer cannot compute the correct input. For example, if the random cookie introduces path predicates, by causing the execution of a specific path through a decoding function, replaying execution with that exact path will constrain the cookie to a value that might differ from the initial one. When this happens, the replayed cookie will not be correct, and the replaying attempt will fail. As we will discuss later, AEG is facing a similar limitation. This suggests that research in this area could make progress for both of these tasks.

3.6.5 Implementation: Exploit Generation

By implementing algorithms similar to those described in AXGEN [68], AEG [47] and Mayhem [24], we were able to evaluate the effectiveness of the current state of the art in automatic exploit generation. Our implementation allows us to create exploits for vulnerabilities, allowing the attacker to take control of the program's execution by overwriting a saved instruction pointer (*e.g.,* by overwriting function pointers, or exploiting buffer overflows on the stack).

Vulnerable States. Unlike AEG/Mayhem, but similar to AXGEN, we generate exploits by performing concolic execution on crashing program inputs using `angr`. We drive concolic execution forward, forcing it to follow the same path as a dynamic trace gathered by concretely executing the crashing input applied to the program. Concolic execution is stopped at the point where the program crashed, and we inspect the symbolic state to determine the cause of the crash and measure exploitability. By counting the number of symbolic bits in certain registers, we can triage a crash into a number of categories such as `frame pointer overwrite`, `instruction pointer overwrite`, or `arbitrary write`, among others.

Instruction Pointer Overwrite Technique. The simplest exploitable bug we can encounter is where symbolic bits appear in the instruction pointer at crash time. When detecting that symbolic bits are contained in the instruction pointer, we can constrain our instruction pointer to point to either a controlled sequence of instructions, such as shellcode, or a ROP gadget that pivots the stack to a symbolic buffer where we can execute a ROP chain (generated by our exploit hardening step). `angr` itself handles many of the implementation details discussed in AEG and AXGEN, such as taint tracking and path condition building, allowing us to limit ourselves to finding symbolic memory buffers and applying constraints to register values to generate an exploit, as proposed by these approaches.

Exploiting CGC Binaries. The Cyber Grand Challenge hosts the game on a custom OS which includes only 7 system calls. The lack of system calls which can execute programs and open files means exploitation within the Cyber Grand Challenge is limited to demonstrating register control and the ability to read and write memory. By DARPA standards, two type of exploits exist for the CGC:

- A Type 1 exploit demonstrates that the attacker controls a general purpose register and the instruction pointer register.
- A Type 2 exploit demonstrates that the attacker can perform a controlled read from the process memory space.

Out of the 126 binaries we applied AEG to, we succeeded in exploiting only a total of 4 binaries. For only two of these binaries, we were able to generate a "Type 2" exploit. Both of these "Type 2" exploits were unable to be hardened with ROP and resorted to jumping to shellcode. Additionally, AEG was only able to generate 2 hardened, ROP "Type 1" exploits. We believe these results show that much more work in the field of automated exploit generation is to be done, and that the current methods are not well-applicable to modern vulnerabilities.

Challenges Faced. Here we demonstrate some of the challenges that our tool faced while attempting to exploit Cyber Grand Challenge binaries, using `CROMU00019` [40]. We will focus on the exploitation of the second vulnerability mentioned in this challenge's `README` (specifically, a buffer overflow on the stack that exists during the decoding of an attacker-supplied string).

The major issue we ran into during exploit generation was the presence of path predicates that constrained each byte of the overflowing data to being a single value, despite the values of these bytes being chosen based on symbolic input. `CROMU00019` demonstrates this in its `decode` function. Each byte of the payload takes a branch of the switch statement contained in `decode`, placing restrictive predicates on our path representing the vulnerable state. While the arms of this switch statement are taken based on symbolic data, the data returned is concrete, and each of these arms represents a separate path through the program. The traditional AEG approach assumes the ability to place the proper constraints on symbolic data to carry out control flow hijacking, but this behavior requires finding the *single* path through the `decode` function which places our desired bytes into the output buffer.

The solution to this problem would be to search for a single path which performs a desirable control flow hijack out of the many paths which present vulnerable conditions. However, modern exploit generation capabilities do not have this capacity, and cases like these prevent many of the stack buffer overflow vulnerabilities presented in the CGC Qualifier event from being exploited with the current state-of-the-art automatic exploit generation.

3.6.6 Implementation: Exploit Hardening

To harden exploits against modern mitigation techniques, we implemented a ROP chain compiler based on the ideas in Q [48]. This means that we can automatically generate ROP payloads to fulfill an end goal, such as writing data to memory or calling an arbitrary function in a library. This section focuses on the differences and improvements we made over Q itself.

Our approach comprises the following steps:

Gadget discovery. We scan all executable code in the application, at every byte offset, to identify ROP gadgets and classify them according to their effects. For example, the instruction sequence: `mov [ebx], eax; pop ebx; ret` would be classified as a memory write and a register load. To carry out the classification, our analysis leverages the action history provided by `angr`'s `Path` objects and symbolic relations provided by `Claripy`.

Gadget arrangement. The ROP chain compiler then determines *arrangements* of gadgets that can be used to perform *high-level* actions. For example, a gadget that pushes data to the stack can be paired with a gadget that pops data to create an arrangement that moves data from one register to another.

Payload generation. After the ROP compiler identifies the requisite set of gadget arrangements, it combines these gadgets into a *chain* to carry out high-level actions (such as executing attacker-specified system calls with specified arguments). This is done by writing gadget arrangements into a program state in `angr`, constraining their outputs to the provided arguments, and querying the SMT solver for a solution for their inputs.

Our implementation differs from Q in minor ways. First, Q made no use of the stack as scratch storage space. It is not clear why this is: one explanation is that their analysis platform did not support the modeling of stack operation, while another is that the approach remains more general if we assume that the stack is *not* necessarily pointed to by the stack pointer (and, thus, in an unknown location). In our integrated system, we could identify whether the stack pointer was pointing to the stack, since we had this metadata from the exploit that we generated with our implementation of AEG.

Another improvement has to do with the gadget classification. Q used a *value sampling* method to identify specific classes of gadgets, which led to some number of missed gadgets chains due to the limited coverage of the sample data. In our approach, we symbolically analyze every gadget, using careful caching techniques to keep the analysis fast.

3.7 Comparative Evaluation

By leveraging `angr`'s design, we were able to reproduce the binary analysis techniques that we have discussed, on the same codebase, enabling a comparative evaluation of their effectiveness. To the best of our knowledge, this has not been done before: previous comparisons were carried out on different implementations, leaving the possibility of differences in results being introduced by implementation differences. With the exception of the fuzzer itself (AFL), our analyses are all implemented on the same analysis engine and share over 90% of the same code base with each other.

We use a corpus of CGC binaries, released by DARPA for the CGC Qualification Event, to carry out our evaluation. As discussed in Section 2.5, these binaries vary widely in complexity, but utilize a simple environment model, designed by DARPA to reduce the implementation effort of analysis systems.

We evaluate the techniques that we implemented for CFG recovery, dynamic and static vulnerability discovery, crash replay, exploitation, and exploit hardening. A summary of the analyses we implemented and evaluated, along with the literature on which they are based and the sections in this chapter in which they are described, is produced in Table 3.4.

3.7.1 CFG recovery

As the CFG is used as a pre-requisite for other analyses in `angr`, it is important to understand how well `angr`'s CFG recovery performs. As we discussed in detail in Section 3.3, `angr` has two CFG recovery algorithms: `CFGAccurate` relies on a base approach of *forced execution* and provides two methods of indirect jump resolution (*backwards slicing* and *symbolic back-traversal*), while `CFGFast` mainly uses recursive disassembly and heuristics to quickly identify functions and inter-function control flow.

To understand the effectiveness of these recovery techniques, we compared `CFGFast` and `CFGAccurate` against the CFG recovery of a state-of-the-art commercial tool, IDA Pro 6.9, on CGC binaries. While little details about how IDA Pro recovers the CFG are available, based on descriptions in previous work [56] as well as our observations, we believe that IDA Pro disassembles a binary recursively, uses symbols and other heuristics to determine locations of functions throughout a binary, and then utilizes some lightweight data-flow analyses to further solve the targets of indirect jumps. This makes it more similar, conceptually, to `CFGFast` than to `CFGAccurate`. As ground truth CFG information is not available, we evaluate our results in terms of the relative number of recovered basic blocks and control flow transfers between the results of IDA's and our CFG recovery.

We first evaluate the completeness of our CFG, comparing the blocks and edges identified by `CFGFast` and the graph generated by IDA Pro. Table 3.5 shows our results. `CFGFast` has a slightly better code coverage than IDA Pro, and recovers more edges. We believe that this is because the lightweight data-flow analyses and heuristics that are used by `CFGFast` are more advanced than those used by IDA. Manual analysis of recovery results on a few binaries indicates that `CFGFast` is more aggressive in terms of code recovery: while IDA Pro believes certain parts of code are not reachable and refuses to disassemble it as code, `CFGFast` identifies such locations as code. A possible explanation for this is that our approach might be *overly* aggressive, and as such, it might mis-identify such locations. However, we have not identified such cases when analyzing CGC binaries.

As some binary analyses require *reachability information* from the entry point, we have also included a comparison against the *reachable* portion of a CFG generated by IDA Pro (that is, a CFG comprising those blocks for which a path from the entry point can be determined) against the CFG recovered by `angr`'s `CFGAccurate` analysis. Table 3.5 shows our results. By improving the forced execution technique with backward slicing, `angr` substantially improves its ability to reconstruct the CFG. However, since `CFGAccurate` does not leverage *ad hoc* heuristics, the resulting CFG's code coverage is not as high as IDA Pro's. To achieve a better coverage, the user can provide `CFGAccurate` with all recovered functions from `CFGFast` as starting points.

3.7.2 Evaluation of Vulnerability Analysis Techniques

In Sections 3.4 through 3.5.2, we describe the implementation of several vulnerability discovery techniques. Here, we present the result of a comparative evaluation of these techniques as applied to the CGC dataset. We ran these evaluations with a timeout of 24 hours, which is the time period of the DARPA competition from which we retrieved the evaluation dataset.

We provide a summary of these results in Table 3.7. Additionally, to provide a better context for the number of crashes identified by our techniques, we have included the number of crashes identified by the competitors at the actual CGC Qualification Event, in Table 3.6. The overall scores of the teams relied on more than just crash counts, so the placement in the qualifying event is not correlated with the position of the competitors. Two of these competitors, the first-place team [72] and the seventh-place team [73], have written blog posts describing their techniques in the competition. Both teams used a symbolically-assisted fuzzing technique, conceptually similar to Driller. Note that, while our implementation of Driller identifies the same number of vulnerabilities as the first place team, this is a coincidence (likely driven by the similarity between the techniques).

Dynamic symbolic execution. We chose to evaluate dynamic symbolic execution both alone and in the presence of the Veritesting path explosion mitigation technique. We describe the implementation details of these approaches in Section 3.5.

As expected, dynamic symbolic execution frequently succumbed to the path explosion problem. In total, the standard approach identified vulnerabilities in 16 of the CGC binaries. Veritesting, which is designed to partially mitigate the path explosion problem, identified only 11, for a combined count of 23 applications in which vulnerabilities were identified.

We were initially surprised to find that, despite the better results, the Veritesting approach found less vulnerabilities than dynamic symbolic execution alone. Investigating these four binaries, we identified an interesting trade-off inherent to Veritesting. Veritesting uses efficient path merging to combat path explosion, which is responsible for its ability to explore deeper paths in the binary before path explosion renders further progress impossible. However, such path merging introduces complex expressions (e.g., if the value of register `eax` differs between two merged paths, the value of the merged path must be a complex expression encoding both previous values) and overloads the constraint solver. Thus, the solve times of the constraint solver tend to increase as more and more of these merges are done. As constraint solving is an NP-complete problem, the increased complexity leads to vulnerabilities becoming unreachable within a reasonable time. The result of this is that Veritesting is able to identify shallow bugs that dynamic symbolic execution otherwise experiences a path explosion with, but overwhelms the constraint solver for longer paths.

Symbolic-assisted fuzzing. Assisted fuzzing has proven to be extremely effective in the literature. In Section 3.5.2, we discuss an implementation of a symbolic-assisted fuzzing method, dubbed Driller [36].

This symbolic-assisted fuzzer uses AFL for the fuzzing component. Each input that AFL produces is traced in the dynamic symbolic execution engine to identify code sections that could be reached by careful mutation of the input. This careful mutation is carried out by the symbolic constraint solver, and the input is reintroduced to AFL for further execution and mutation. Because the individual inputs traced by the DSE engine do not branch (as all the input is concrete), there is no path explosion during tracing, and AFL limits the number of inputs passed to the DSE engine by filtering out all the inputs that do not increase code coverage.

It should be mentioned that AFL alone is able to identify vulnerabilities in a significant amount of the CGC services. In fact, of the 77 vulnerabilities that our symbolic-assisted fuzzer detected, 68 were detected by AFL alone. The remaining 9 were found through the use of symbolic assistance.

DSE vs. fuzzing. The difference between the results of the various dynamic symbolic execution approaches are surprising. One might reasonably expect DSE to identify roughly as many vulnerabilities as symbolically-assisted fuzzing, and more than fuzzing alone. In reality, fuzzing identified almost *three times* as many vulnerabilities. In a sense, this mirrors the recent trends in the security industry: symbolic analysis engines are criticized as impractical while fuzzers receive an increasing amount of attention. However, this situation seems at odds with the research directions of recent years, which seem to favor symbolic execution.

Analyzing the crashing inputs that these approaches did find, we identified an interesting result: the exploits found by dynamic symbolic execution engines tend to represent *short* paths. This result is presented in Figure 3.1. By spot-checking several applications where dynamic symbolic execution (even with Veritesting) *failed* to find vulnerabilities, we have concluded that this is due to an increase in analysis complexity, exponentially proportional to the length of the path.

Specifically, given a path A, there is a chance p_a that A will split at the end of the next conditional jump, and A_1 will follow the path that takes the jump, while A_2 will follow the path that does not. At the next conditional jump, there is a chance that A_1 and A_2

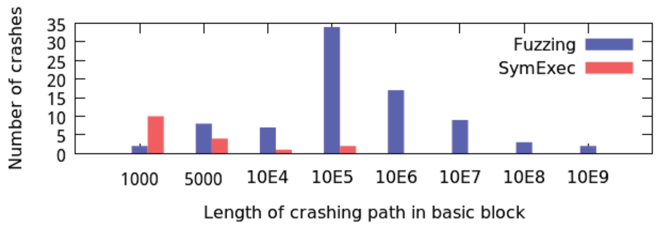


Figure 3.1: Length of crashing paths discovered by fuzzing vs. dynamic symbolic execution.

will fork as well. Thus, the amount of resulting paths to analyze increases exponentially, and the chance that an unreasonable number of paths will have to be analyzed at any point is exponentially proportional to how many basic blocks have been executed by the analysis. As a result, the typical dynamic symbolic execution approach is best for finding *shallow* crashes that do not require the execution of many basic blocks. *Deep* crashes, on the other hand, tend to be hidden and made unreachable by the path explosion.

To further understand the relative effectiveness of the techniques, we calculated the *code coverage* of the generated test cases. We found that symbolic execution (including Veritesting) covered an average of 330 blocks per binary (with a median of 260), while fuzzing covered 689 (with a median of 402) and symbolic-assisted fuzzing covered 698 (with a median of 406). These results yield another interesting conclusion: if the paths generated by fuzzing or symbolic-assisted fuzzing were combined into a *graph*, it would represent a CFG with more code coverage than the one recovered by `CFGAccurate` (and, by virtue of each edge in the graph being reachable by definition, with perfect completeness), implying a need for further improvement of the accurate CFG recovery algorithm.

Under-constrained symbolic execution. We extended `angr` to support under-constrained symbolic execution to better understand how effective such techniques are on our dataset. These details are presented in Section 3.5.1.

UC-`angr` reported 371 vulnerabilities in the CGC binaries. However, as this approach analyzes functions without their context, it suffers from similar problems as static analyses: the results contain a large number of false positives, and the results are not replayable (that is, they do not generate crashing inputs, but instead point out the location of vulnerabilities). In fact, we identified 346 false positives in UC-`angr`'s results, leaving 25 true positives and resulting in a false positive rate of 93%, which is in line with those reported by UC-KLEE [38].

Static buffer overlap detection. To be able to compare the different types of vulnerabilities identified by fuzzing, symbolic execution, and other static analyses, we implemented a VSA-based memory corruption detection analysis. We describe it in detail in Section 3.4.

Similarly to UC-`angr`, our VSA results are not replayable and suffer from false positives. In total, VSA was able to identify 27 actual vulnerabilities in CGC binaries while producing 130 false positives, resulting in a false-positive rate of 82.8%.

Non-replayable vs replayable analyses. Another surprising result is the comparatively low performance of non-replayable techniques (VSA and under-constrained symbolic execution). While these techniques, freed from the replayability requirement, can achieve more coverage in their analysis, we found that the context they lacked resulted in an enormous amount of false positives in this dataset. To keep the false positive rate reasonable, we had to implement aggressive false positive filtering (as discussed in Section 3.5.1), which filtered out many true positives as well.

The improvement of static analysis techniques on real binaries appears to be an area in need of research attention, and we are considering it as a direction for future work.

3.7.3 Exploitation Evaluation.

After a crash is identified by the above approaches, we attempt to replay and exploit it to understand its severity.

Crash replay. As we discuss in Section 3.6.1, crashing inputs identified by vulnerability discovery analyses might not be trivially replayable due to environmental data (such as the random seed) having been *de-randomized* by the analysis. We analyzed crashes for each CGC binary, using the reference crashing inputs provided by DARPA for binaries where we were unable to identify vulnerabilities with our vulnerability identification techniques. Of these crashing inputs, 6 were not trivially replayable. That is, rather than simply replaying the crashing input provided to us by the vulnerability identification engines, we had to re-analyze the interaction with the binary to recover challenge-response components present in these binaries.

Interestingly, DARPA imposes a limitation on the authors of CGC binaries from the CGC Qualifying Event that disallows control flow from being impacted by random data. This means that the limitation of Replayer discussed in Section 3.6.4, the introduction of different path predicates due to different values of random data, does not apply to its operation on CGC binaries. Though `angr` did hang on one of the applications, manual analysis revealed this to be an implementation issue, rather than one with the approach and, as expected, Replayer was able to recover the input specification of the remaining 5.

While 6 binaries are not a large dataset, this result suggests that current techniques in this area are able to adequately handle binaries in the absence of control flow variance caused by random data. Further work is needed to evaluate, and possibly extend, these techniques on real-world binaries with more complex control flow.

Automatic exploit generation. After identifying the crash and running it through

Replayer, we are left with an input specification that reliably crashes the target application. However, such inputs might still not be *exploitable*. For example, crashes caused by null pointer dereferences, of which there are many in the CGC dataset, are not exploitable on modern systems. To separate exploitable from non-exploitable inputs, we attempt to generate an exploit from the crash.

We attempted to automatically generate exploits for all of the CGC applications, using techniques proposed in the AEG system [47]. However, we were surprised to find that only *4* crashing exploits could be weaponized into exploits using these techniques. Looking deeper into the binaries, we understood why. First, the goal of the CGC Qualification Event was to find *crashes* for the binaries, not exploits. As such, many of the vulnerabilities in these binaries are not actually exploitable (i.e., null-pointer dereferences). Second, as the CGC binaries model a wide range of realistic exploitation scenarios, we found that the techniques proposed by AEG were not applicable to the majority of them.

The current state of the art in this field is fairly basic, and it appears in these results. Further research is required into this field to enable the automatic exploitation of complex vulnerabilities.

Exploit hardening. Even an exploitable vulnerability might be mitigated by modern protections. As a result, *exploit hardening* is required, and has been investigated by recent work. We reimplemented the techniques proposed by Q [48] and attempted to harden the exploits generated by AEG.

The Q implementation was able to harden 2 of the 4 exploits that AEG generated. Our analysis as to why the remaining two exploits could not be hardened revealed that the Q approach does not utilize enough information in the binary. In these two examples, there is not enough attacker-controlled data on the stack and a *stack pivot* is required to use attacker-controlled data in other parts of the program. The Q approach has no basis for reasoning about such operations and, as a result, these exploits cannot be hardened.

3.8 Conclusions

In this chapter, we presented `angr`, a system that implements, in a unified framework, a number of techniques for the automated identification and exploitation of vulnerabilities in binaries. We presented, in a systematized fashion, the different analyses and the challenges we encountered when including them in our framework. By implementing these approaches in a single system, we were able to meaningfully compare their effectiveness on a dataset that was created for the evaluation of these techniques. The results of this evaluation can be used as a basis to highlight research directions, and to improve existing techniques.

We made `angr` open-source, so that the community can build on top of it and focus on addressing open challenges in the field of binary analysis.

IR Expression	**Evaluated Value**
Constant	A constant value.
Read Temp	The value stored in a VEX temporary variable.
Get Register	The value stored in a register.
Load Memory	The value stored at a memory address, with the address specified by another IR Expression.
Operation	A result of a specified IR Operation, applied to specified IR Expression arguments.
If-Then-Else	If a given IR Expression evaluates to 0, return one IR Expression. Otherwise, return another.
Helper Function	VEX uses C helper functions for certain operations, such as computing the conditional flags registers of certain architectures. These functions return IR Expressions.

Table 3.1: A list of relevant VEX IR Expressions for Firmalice's analysis.

IR Statement	**Effect**
Write Temp	Set a VEX temporary variable to the value of the given IR Expression.
Put Register	Update a register with the value of the given IR Expression.
Store Memory	Update a location in memory, given as an IR Expression, with a value, also given as an IR Expression.
Exit	A conditional exit from a basic block, with the jump target specified by an IR Expression. The condition is specified by an IR Expression.

Table 3.2: A list of relevant VEX IR Statements for Firmalice's analysis and their effects on the program state.

ARM Assembly	**VEX Representation**

subs R2, R2, #8	t0 = GET:I32(16)
	t1 = 0x8:I32
	t3 = Sub32(t0,t1)
	PUT(16) = t3
	PUT(68) = 0x59FC8:I32

Table 3.3: An example of a VEX IR translation of a machine code instruction located at 0x59FC4. VEX converts register names to numerical identifiers: 16 refers to R2 and 68 refers to the program counter.

Technique	**Based On**	**Described In**
Dynamic Symbolic Execution	*Various* [22, 24, 71]	2.3.1, 3.5
Veritesting	Veritesting [28]	2.3.1, 3.5
Under-constrained DSE	UCSE [38]	2.3.1, 3.5.1
Symbolic-Assisted Fuzzing	Driller [36]	2.3.1, 3.5.2
Static Analyses	VSA [12]	2.2.2, 3.4
Crash Replay	Replayer [66]	3.6.1, 3.6.4
Exploit Generation	AEG [47]	3.6.2, 3.6.5
Exploit Hardening	Q [48]	3.6.3, 3.6.6

Table 3.4: Analyses implemented and evaluated in this chapter, the literature on which they are based, and the sections of this chapter in which they are discussed.

Approach	**Functions**		**Function Edges**		**Blocks**		**Block Edges**		**Bytes**		**Time (s)**	
	M	A	M	A	M	A	M	A	M	A	M	A
IDA Pro 6.9	48	52.96	76.5	99.62	829	3589.93	1188	6487.68	14037	104779.66	1.14	1.80
`angr` - `CFGFast`	61	70.08	88	118.74	843	3609.45	1193	6538.52	14296	105007.49	0.87	5.01
IDA Pro 6.9 - reachability	37	40.96	74	90.76	496	1043.81	759	1693.01	7874	21721.85	1.14	1.80
`angr` - forced execution	31	33.24	48	55.22	349.5	413.85	612	751.96	6125	13963.5	23.50	36.96
`angr` - symbolic back traversal	32	33.76	50	56.28	368	635.41	645	1089.78	6323	10883.51	27.22	34.10
`angr` - backward slicing	30	32.80	47.5	53.89	344.5	653.56	594	1178.98	6109.5	14641.85	24.78	79.46

Table 3.5: Evaluation of `CFGFast`'s and `CFGAccurate`'s recovered CFG versus the CFG recovered by IDA Pro. The median number (M) and average number (A) of each value across all binaries are shown.

CGC Qualifying Position	Binaries Crashed
First	77
Second	12
Third	57
Fourth	9
Fifth	23
Sixth	57
Seventh	44
Eighth (did not qualify)	39
Ninth (did not qualify)	65

Table 3.6: Number of crashed binaries for the top 9 competitors in the CGC Qualification Event.

Technique	Replayable	Semantic Insight	Scalability	Crashes	False Positives
Dynamic Symbolic Execution	Yes	High	Low	16	0
Veritesting	Yes	High	Medium	11	0
Dynamic Symbolic Execution + Veritesting	Yes	High	Medium	23	0
Fuzzing (AFL)	Yes	Low	High	68	0
Symbolic-Assisted Fuzzing	Yes	High	High	77	0
VSA	No	Medium	High	27	130
Under-constrained Symbolic Execution	No	High	High	25	346

Table 3.7: Evaluation results across all vulnerability discovery techniques.

Chapter 4

Application of Cyber-autonomy to the Detection of Authentication Bypass Vulnerabilities

Over the last few years, as the world has moved closer to realizing the idea of the *Internet of Things*, an increasing amount of the things with which we interact every day have been replaced with embedded devices. These include previously non-electronic devices, such as locks[1], lightswitches[2], and utility meters (such as electric meters and water meters)[3], as well as increasingly more complex and ubiquitous devices, such as network routers and printers. These embedded devices are present in almost every modern home, and their use is steadily increasing. A study conducted in 2011 noted that almost two thirds of US households have a wireless router, and the number was slated to steadily increase [74]. The same report noted that, in South Korea, Wi-Fi penetration had reached 80%. The numbers are similar for other classes of devices: it has been predicted that the market penetration of smart meters will hit 75% by 2016, and close to 100% by 2020.

These examples are far from inclusive, as other devices are becoming increasingly intelligent as well. Modern printers and cameras include complex social media function-

[1]For example, the Kwikset Kevo smart lock.
[2]Most popularly, Belkin's WeMO line.
[3]Such as the ION, a smart meter manufactured by Schneider Electric.

ality, “smart” televisions are increasingly including Internet-based entertainment options, and even previously-simple devices such as watches and glasses are being augmented with complex embedded components.

The increasingly-complex systems that drive these devices have one thing in common: they must all communicate to carry out their intended functionality. Smart TVs communicate with (and accept communication from) online media services, smart locks allow themselves to be unlocked by phones or keypads, digital cameras contact social media services, and smart meters communicate with the user’s utility company. Such communication, along with other functionalities of the device, is handled by software (termed “firmware”) embedded in the device.

Because these devices often receive privacy-sensitive information from their sensors (such as what a user is watching, or how much electricity they are using), or carry out a safety-critical function (such as actuators that lock the front door), errors in the devices’ firmware, whether present due to an accidental mistake or purposeful malice, can have serious and varying implications in both the digital and physical world. For example, while a compromised smart meter might allow an attacker to determine a victim’s daily routine or otherwise invade their privacy based on their energy usage, a compromised smart lock can permit unauthorized entry into a victim’s home (or, in a corporate setting, a compromised badge access verifier can allow entry into extremely critical physical areas of an organization). In fact, this is not just a theoretical concern: there have already been examples of “smart-home” invasions [75].

Firmware is susceptible to a wide range of software errors. These include memory corruption flaws, command injection vulnerabilities, and application logic flaws. Memory corruption vulnerabilities in firmware have received some attention [76, 77], while other vulnerabilities have, as of yet, been relatively unexplored in the context of firmware.

One common error seen in firmware is a logic flaw called an *authentication bypass* or, less formally, a *backdoor*. An authentication bypass occurs when an error in the authentication routine of a device allows a user to perform actions for which they would otherwise need to know a set of credentials. In other cases, backdoors are deliberately inserted by the manufacturer to get access to deployed devices for maintenance and upgrade. As an example, an authentication bypass vulnerability on a smart meter can allow an attacker to view and, depending on the functionality of the smart meter, modify the recorded energy usage of a victim without having to know the proper username and password, which, is generally kept secret by the utility company. Similarly, in the case of a smart lock, an authentication bypass could allow an attacker to unlock a victim's front door without knowing their passcode.

Authentication bypass vulnerabilities are not just a theoretical problem: recently publicized vulnerabilities in embedded devices describe authentication bypass flaws present in several devices' firmware [78, 79], and a recent study has suggested that up to 80% of consumer wireless routers are vulnerable to *known* vulnerabilities [80]. In fact, an authentication bypass in Linksys routers was used by attackers to create a botnet out of 300,000 routers in February 2014 [81].

Detecting authentication bypasses in firmware is challenging for several reasons. To begin with, the devices in question are usually proprietary, and, therefore, the source code of the firmware is not available. While this is a problem common to analyzing binary software in general, firmware takes it one step further: firmware often takes the form of a single binary image that runs directly on the hardware of the device, without an underlying operating system[4]. Because of this, OS and library abstractions do not exist in some cases, and are non-standard or undocumented in others, and it is frequently unknown how to properly initialize the runtime environment of the firmware sample

[4]The operating system is self-contained in the binary, and we cannot rely on *a-priori* knowledge or known interfaces to such systems.

(or, even, at what offset to load the binary and at what address to begin execution). We term such firmware as “binary blob” firmware. These blobs can be very large and, therefore, any analysis tool must be able to handle such complex firmware. Additionally, embedded devices frequently require their firmware to be cryptographically signed by the manufacturer, making modification of the firmware on the device for analysis purposes infeasible.

These challenges make existing approaches infeasible for identifying logic flaws in firmware. Systems that are based on the instrumentation and execution monitoring of firmware on a real device [82, 77] would not be able to operate in this space, because they require access to and modification of the device in order to run custom software. In turn, this is made difficult by the closed nature (including the aforementioned cryptographic verification of firmware images) and the hardware disparity (any sort of on-device instrumentation would represent a per-device development effort) of embedded devices. Additionally, existing firmware analysis systems that take a purely symbolic approach (and, thus, do not require analyses to be run on the device itself) [76] would not be able to scale their analysis to complex firmware samples, like those used by printers and smart meters. Moreover, they require source code, which is typically not available for proprietary firmware. As a result of these challenges, most detections of authentication bypasses in firmware are done manually. This is problematic for two reasons. First, manual analysis is impractical given the raw number of different embedded devices on the market. Second, even when manual analysis is performed, the complexity of firmware code can introduce errors in the analysis.

To address the shortcomings of existing analysis approaches, we developed a system, called *Firmalice*, that automates most of the process of searching firmware binaries for the presence of logic flaws. To the best of our knowledge, Firmalice is the first firmware analysis system working at the binary level, in a scalable manner, and with no requirement to instrument code on the original device.

We applied Firmalice to the detection of authentication bypass flaws, as seen in several recent, high-profile disclosures of firmware backdoors. To allow Firmalice to reason about such flaws, we created a novel model of authentication bypass vulnerabilities, based around the concept of an attacker's ability to determine the input necessary to execute privileged operations of the device. Intuitively, if an attacker can derive the necessary input for driving a firmware to perform a privileged operation simply by analyzing the firmware, the authentication mechanism is either flawed or bypassable. Additionally, this model allows us to reason about complicated backdoors, such as cases when a backdoor password is secretly disclosed to the user by the firmware itself, as we consider such information determinable by an attacker.

Because the definition of a *privileged operation* (*i.e.,* an operation that requires preliminary authencation) varies between devices, Firmalice requires the specification of a security policy for each firmware sample, to locate such operations. Our system receives a firmware sample and the specification of its security policy, and then loads the firmware sample, translates its binary code into an intermediate representation, and performs a static full-program control and data flow analysis, followed by symbolic execution of firmware slices, to detect the presence of any violations of the security policy.

We evaluated our approach against three real-world firmware samples: a network printer, a smart meter, and a CCTV camera. These devices demonstrate Firmalice's ability to analyze diverse hardware platforms, encompassing both ARM and PPC, among other supported architectures. Additionally, these samples represent both bare-metal binary blobs and user-space programs, and their backdoors are triggered in several different ways.

To summarize, we make the following contributions:

- We develop a model to describe, in an architecture-independent and implementation-independent way, authentication bypass vulnerabilities in firmware binaries. This model is considerably more general than existing techniques, and it is focused around the concept of *input determinism*. The model allows us to reason about, and detect, complicated backdoors, including intentionally-inserted authentication, bugs in authentication code, and missing authentication routines, without being dependent on implementation details of the firmware itself.

- We implement a tool that utilizes advanced program analysis techniques to analyze binary code in complex firmware of diverse hardware platforms, and automate much of the process of identifying occurrences of authentication bypass vulnerabilities. This tool uses novel techniques to improve the scalability of the analysis.

- We evaluate our tool on three real-world firmware samples, detailing our experiments and successfully detecting authentication bypass vulnerabilities, demonstrating that manual analysis is not sufficient for authentication bypass detection.

4.1 Authentication Bypass Vulnerabilities

The increased prominence of embedded consumer electronics have given rise to a new challenge in access control. Specifically, many embedded devices contain *privileged operations* that should only be accessible by *authorized users*. One example of this is the case of networked CCTV cameras: it is obvious that only an authenticated user should be able to view the recorded video and change recording settings. To protect these privileged operations, these devices generally include some form of user verification. This verification (i.e., only an authorized user can access privileged functionality) almost always takes the form of an authentication of the user's credentials before the privileged functionality is executed.

The verification can be avoided by means of an authentication bypass attack. Authentication bypass vulnerabilities, commonly termed "backdoors," allow an attacker to perform privileged operations in firmware without having knowledge of the valid credentials of an authorized user.

The backdoors that we have analyzed tend to assume one of several forms, which we will detail here, before describing how we designed Firmalice to detect the presence of these vulnerabilities.

Intentionally hardcoded credentials. The most common type of authentication bypass is the presence of hardcoded authentication credentials in the authentication routine of an embedded device. Most commonly, this takes the form of a hardcoded string against which the password is compared (e.g., using `strcmp()`). If the comparison succeeds, access is granted to the attacker. There have been many recent high-profile cases of such backdoors. We discuss one such case, a backdoor in the web interface of a networked CCTV camera [83], in Section 4.8.2.

In some cases, the credentials might not be directly hardcoded in this manner, but still predictable. One example is a popular model of smart meter, that calculates a "factory access" password by hashing its publicly-known model number [84].

Intentionally hidden authentication interface. Alternatively, an authentication bypass can take the form of a hidden (or undocumented) authentication interface. Such interfaces grant access to privileged operations without the need for an attacker to authenticate. Hidden authentication interfaces have been featured in some recent vulnerabilities [79, 85], and we describe one such case, affecting a popular model of network printer.

Unintended bugs. Sometimes, unintended bugs compromise the integrity of the authentication routine, or allow its bypass completely. One example is command injection: some routers allow unauthenticated users to test connectivity by providing a web interface to the ping binary, and incorrect handling of user input frequently results in command injections.

By analyzing known authentication bypass vulnerabilities in firmware samples, we identified that a lack of *secrecy*, or, inversely, the *determinism* of the input necessary to perform a privileged operation, lies at the core of each one. That is, the authentication bypass exists either because the required input (most importantly, the credentials) was insufficiently secret to begin with (i.e., a comparison with a hardcoded string embedded in the binary), because the secrecy was compromised during communication (for example, by leaking information that could be used to derive a password), or because there was no authentication to begin with (such as the case of an administrative interface, listening, sans authentication, on a secret port).

To reason about these vulnerabilities, we created a model based on the concept of *input determinism*. Our model is a generalization of this class of vulnerability, leveraging the observation that input determinism is a common theme in authentication bypass vulnerabilities. Our authentication bypass model specifies that all paths leading from an entry point into the firmware (e.g., a network connection or a keyboard input handler) to a privileged operation (e.g., a command handler that performs some sensitive action) must validate some input that the attacker cannot derive from the firmware image itself or from prior communication with the device. In other words, we report an authentication bypass vulnerability when an attacker can craft (a possible sequence of) inputs that lead the firmware execution to a privileged operation. Whenever the attacker is able to extract such input from the analysis of the firmware itself, he has found an authentication bypass vulnerability.

This model is considerably more general than existing approaches: it is not important how the actual authentication code is implemented, or, to an extent, what the actual vulnerability is; the analysis needs only to reason about the attacker's ability to determine the input. Note that our model does not require any knowledge of a specific authentication function. In fact, as an interesting special case, our system reports an authentication bypass for all instances where the authentication function is entirely missing.

Unlike classical memory corruption vulnerabilities, such as buffer overflows, logic vulnerabilities such as authentication bypasses require a semantic understanding of the actual device in question. Specifically, the definition of a *privileged operation* will differ for different devices. Firmalice requires the analyst to provide this information as part of a "Security Policy", which specifies resources that a device may not access or actions that a device cannot perform without authentication. We describe these policies in detail in Section 4.4.

In the next section, we will provide an overview of Firmalice's operation, from the input of a firmware sample and security policy to the detection of authentication bypass vulnerabilities.

4.2 Approach Overview

The identification of authentication bypasses in firmware proceeds in several steps. At a high level, Firmalice loads a firmware image, parses a security policy, and uses static analysis to drive a symbolic execution engine. The results from this symbolic execution are then checked against the security policy to identify violations.

We summarize each individual step in this section, and describe them in detail in the rest of the chapter.

Firmware Loading. Before the analysis can be carried out, firmware must be loaded into our analysis engine. We describe this process, and the special challenges that firmware analysis introduces, in Section 4.3. The output of this step is an internal representation of a loaded, ready-to-analyze firmware sample.

Security Policies. Firmalice has the capability to translate security policies into analyzable properties of the program itself. Specifically, Firmalice takes the *privileged operation*, described by a security policy, and identifies a set of *privileged program points*, which are points in the program that, if executed, represent the privileged operation being performed. Security policies, and how Firmalice translates them into privileged program points, are described in Section 4.4.

Static Program Analysis. The loaded firmware is then passed to the *Static Program Analysis* module. This module generates a program dependency graph of the firmware and uses this graph to create an *authentication slice* from an entry point to the privileged program point. This is detailed in Section 4.5.

Symbolic Execution. The authentication slice created by the *Static Program Analysis* module is passed to Firmalice's *Symbolic Execution* engine, presented in Section 4.6. The symbolic execution engine attempts to find paths that successfully reach a *privileged program point*. When such a path is found, the resulting symbolic state (termed the *privileged state*), is passed to the *Authentication Bypass Check* module.

Authentication Bypass Check. Every *privileged state* found by the *Symbolic Execution* engine is passed to the *Bypass Check* module. This module uses the concept of *input determinism* to determine whether the state in question represents the use of an authentication bypass vulnerability. The authentication bypass model, and the procedure to check a privileged state against it, are described in Section 4.7. If the state is determined to represent an authentication bypass, Firmalice's analysis terminates, and the input required to trigger the bypass is extracted and provided as Firmalice's output. If the input required to bypass authentication depends on prior communication with the device, Firmalice produces a function that, given the output of such communication, produces a valid input.

State	**Constraints**	**Input**
Backdoor	input_0 = "GO" && input_1 = "ON"	"GO\nON\n"
Normal	input_0 = get_username_0 && input_1 = get_password_0	(undetermined)

Table 4.1: The privileged states resulting from Firmalice's symbolic execution.

To better explain how Firmalice operates on a firmware sample, we present an example in this section. For simplicity, the example is a user-space firmware sample with a hardcoded backdoor, shown in Listing 2 (the backdoor is the check in lines 2 and 3). Note that while Listing 2 presents source code, our approach operates on binary code.

In this example, the security policy provided to Firmalice is: "The Firmware should not present a prompt for a command (specifically, output the string `Command:`) to an unauthenticated user."

Firmalice first loads the firmware program, using the techniques described in Section 4.3, and carries out its Static Program Analysis, as described in Section 4.5. This results in a control flow graph and a data dependency graph. The latter is then used to identify the location in the program where the string `Command:` is shown to the user. This serves as the privileged program point for Firmalice's analysis. The control flow graph, which is part of the end result of the Static Program Analysis, is shown in Figure 4.1, with the privileged program point marked with a dashed outline.

Firmalice utilizes its Static Program Analysis module to create an authentication slice to the privileged program point. In our example, this slice comprises the nodes in Figure 4.1 that are not greyed out.

The extracted authentication slice[5] is then passed to Firmalice's Symbolic Execution engine. This engine explores the slice symbolically, and attempts to find user inputs that would reach the privileged program point. In this case, it finds two such states: one that authenticates the user via the backdoor, and one that authenticates the user properly. The symbolic constraints associated with these states are shown in Table 4.1.

As these privileged states are discovered, they are passed to the Authentication Bypass Check module. In this case, the component would detect that the first state (with a username of "GO" and a password of "ON") contains a completely deterministic input, and, thus, represents an authentication bypass. Upon detecting this, Firmalice's analysis terminates and outputs the input required to reach the privileged program point.

[5]Starting at the user input in line 19, traversing the `auth()` function, and ending at the privileged program point in line 20.

```c
int auth(char *u, char *p) {
  if ((strcmp(u, "GO") == 0) &&
     (strcmp(p, "ON") == 0))
             return SUCCESS;

  for (int i = 0; i < 10000000; i++)
            pointless();

  char *stored_u = get_username();
  char *stored_p = get_password();
  if ((strcmp(u, stored_u) == 0) &&
      (strcmp(p, stored_p) == 0))
     return SUCCESS;
  else return FAIL;
}

int main() {
  puts("Hello!");
  if (auth(input("User:"), input("Password:")))
    system(input("Command:"));
}
```

Listing 2: Example of authentication code containing a hard-coded backdoor.

4.3 Firmware Loading

The first step of analyzing firmware is, of course, loading it into the analysis system. Firmware takes one of two forms:

user-space firmware. Some embedded devices actually run a general-purpose OS, with much of their functionality implemented in user-space programs. A common ex-

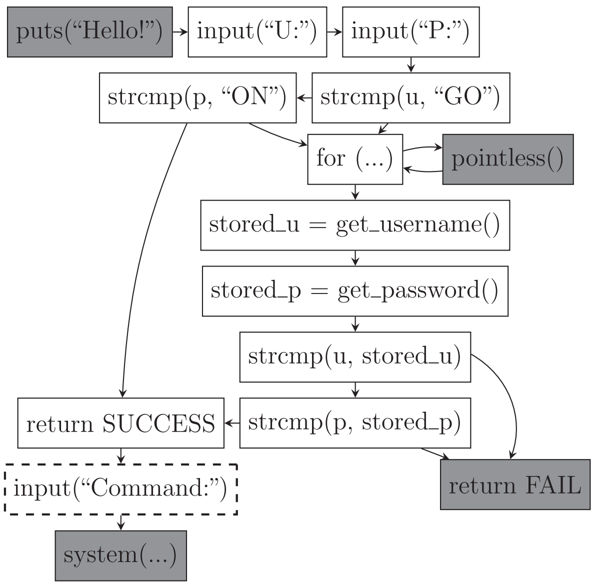


Figure 4.1: Firmalice's CFG for the example. The darkened nodes are excluded from the authentication slice.

ample of this is the wide array of Wi-Fi routers on the market, generally running a stripped-down version of Linux. All of the OS primitives (i.e., system calls), program entry points, and library import symbols are well-defined.

Binary-blob firmware. Firmware often takes the form of a single binary image that runs directly on the bare metal of the device, without an underlying operating system. OS and library abstractions do not exist in such cases, and it is generally unknown how to properly initialize the runtime environment of the firmware sample, or at what offset to load the binary and at what address to begin execution.

Analyzing user-space firmware samples is analogous to analyzing a normal user-space program, whereas binary-blob firmware presents several challenges unique to firmware analysis, which we discuss in this section. The output of this phase of the analysis is an internal representation of the firmware, properly loaded in memory and ready to be analyzed. This is then passed to the Static Program Analysis step, described in Section 4.5.

4.3.1 Disassembly and Intermediate Representation

Before an analysis of a firmware sample can be carried out, the binary must be disassembled. This is complicated by the fact that, in the case of binary-blob firmware, the base address where the binary should be loaded and the entry point are not known. Disassembling binary code without this knowledge has been well explored by existing work [60]. Therefore, we leverage existing techniques to acquire a reliable disassembly of the firmware.

As with any static analysis tool, proper disassembly of the firmware sample is a strict requirement for Firmalice's operation. However, modern disassembly techniques have been honed on evasive code, such as malware, and we feel (and, in fact, it has been our experience) that there are no issues disassembling firmware code. Unlike malware, and due to the power and performance requirements of embedded devices, firmware is not obfuscated, and the aforementioned techniques are effective.

Firmalice supports a wide range of processor architectures by carrying out its analyses over an intermediate representation (IR) of binary code. While the choice of a representation itself is not important for our analysis, we present the IR that Firmalice uses in Appendix 3.2.

4.3.2 Base Address Determination

Binary-blob firmware typically comes with no information as to the memory location at which it expects to be loaded in the device's memory. Before an analysis of the firmware can be carried out, this value must be determined. Firmalice accomplishes this by leveraging jump tables in the binary.

Jump tables comprise a set of absolute code addresses, stored sequentially in memory. These addresses are read (in many cases, using absolutely-addressed memory accesses) by indirect jumps to determine the jump target. Firmalice identifies the expected location of a binary-blob firmware in memory by analyzing the relationship between jump table positions and the memory access pattern of the indirect jump instructions.

The targets of jump tables tend to exhibit high spatial locality, as they are commonly different cases of a switch statement in the same function. That is, jump tables are typically stored as consecutive values in memory, each of these values being a target address. To identify jump tables, Firmalice scans a binary blob (in steps equal to the architecture's address bit width) for consecutive values that differ only in their least significant two bytes. Firmalice then analyzes all indirect jumps found in the disassembly phase and identifies the memory locations from which they read their jump targets. The binary is then relocated so that the maximum number of these accesses are associated with a jump table.

4.3.3 Entry Point Discovery

Unfortunately, without a standard executable file format, binary blobs lack entry point information. That is, even after disassembling a binary, it is unclear from which start instruction the analysis should begin. As an additional complication, there may be multiple entry points to support features such as interrupt requests, with each interrupt request handler representing an additional entry point into the firmware.

Firmalice's static analysis requires knowledge of the entry points to the firmware. Prior work, such as Avatar [77], has required the manual specification of entry points, but in order to reduce the amount of required manual input, Firmalice attempts to automatically identify potential execution entry points. This is done in several steps.

First, Firmalice attempts to identify functions in the binary blob. This is done by scanning through the binary blob for common function prologue instructions (depending on the architecture in question) and analyzing the control flow until a return is encountered. If the function being analyzed issues a call instruction, Firmalice adds the called function to its analysis as well.

Next, Firmalice creates a coarse directed call graph from the list of functions, and identifies all the weakly-connected components of this graph. Any root node of a weakly-connected component is identified as a potential entry point. This is based on the assumption that, since it is not called in the code, it may be called as an interrupt handler. For Firmalice's purposes, an over-estimation of entry points is acceptable in practice. The reason for this is that the privileged program points are not reachable from most of the entry points, and hence the static analysis discards superfluous entry points from further consideration.

4.4 Security Policies

Traditional vulnerability detection systems such as KLEE [22], AEG [86], and Mayhem [24], among others, are designed to identify memory corruption vulnerabilities in software. Since such vulnerabilities are easily described in a general way (i.e., a control flow hijack occurs whenever the program being analyzed jumps to a user-specified location), these systems can be created with a specific vulnerability model and that is then leveraged in the analysis of many different programs.

Firmalice's task is more difficult, as authentication bypass vulnerabilities are a class of logic flaws. Logic flaws take many forms based on, intuitively, the actual intended logic of the developers of the software (or, in our case, firmware) that is analyzed. Since a logic flaw is a deviation of a program's execution from the logic intended by the developers of the program, what actually constitutes one is highly dependent on what the device in question is designed to do. This holds true for authentication bypass vulnerabilities, the specific class of logic flaws that Firmalice is designed to detect. For example, the ability to watch videos without authentication might be acceptable when dealing with a streaming media set-top box, but represents an authentication bypass when analyzing a network-connected camera.

Automatically reasoning about the *intended* logic of a program requires reasoning about the intentions of the programmer, which we consider outside of the scope of program analysis. Thus, Firmalice requires a human analyst to provide a security policy. For our purposes, a security policy must specify what operations should be considered privileged (and, hence, must always require the user to be authenticated).

When provided a security policy, Firmalice analyzes the firmware in question to convert the policy into a set of *privileged program points*: that is, a set of points in the code of the firmware that, when executed, would cause the privileged operation to be performed. This set of program points is then utilized by Firmalice in its analysis to identify if the execution can reach the specified program point without proper authentication.

These policies vary in the amount of knowledge that they require the analyst to have about the inner working of the firmware: from information that any user moderately familiar with the device would possess, to intricate details about code reachability or memory accesses. The rest of this section describes the policies that Firmalice supports and discusses how Firmalice utilizes these policies to identify *privileged program points*.

Static output. A security policy can be specified as a rule about some static data (usually ASCII text, but in general any sequence of bytes) the program must not output to a user that has not been properly authenticated. An example of such policy is "The program must not output `AUTHENTICATION SUCCEEDED` to an unauthenticated user."

When provided such a policy, Firmalice searches the firmware for the static data and utilizes its data dependency graph (described in Section 4.5) to identify locations in the program where this data can be passed into an output routine. These locations become the *privileged program points* for the remainder of the analysis.

Behavioral rules. Another policy that Firmalice supports is the regulation of what *actions* a device may take without authentication. In the case of a smart lock, this policy might be "The lock motor must never turn without proper authentication." For Firmalice to be able to reason about such policies, the user must also specify *how* this action would be accomplished. For example, for a device with peripherals that should never read from an attached camera without authentication, this might be "A file in `/dev` must never be opened without authentication."

Firmalice processes this policy by analyzing its control flow graph and data dependency graph for positions where an action is taken that matches the parameters specified in the security policy. In our example, this would be any location where a string that is data-dependent on any string starting with "/dev" is passed to the `open` system call.

Memory access. Embedded devices often communicate with and act on memory-mapped sensors and actuators. To support identifying authentication bypass vulnerabilities in such devices, Firmalice accepts security policies that reason about access to absolute memory addresses. When supplied such a policy, Firmalice identifies locations in the data dependency graph where such memory locations are accessed, and identifies them as *privileged program points*.

Direct privileged program point identification. If the analyst has detailed knowledge about the firmware, the privileged program points can be specified directly as function addressed in the security policy. These are then passed directly to the rest of the analysis.

These security policies are general enough to cover the intended behavior of the firmware samples that we have seen so far.

Of course, Firmalice's Security Policy Parsing module can be extended to support other types of security policies, if required. However, we see the creation and parsing of more intricate security policy as an orthogonal problem to the identification of authentication bypass vulnerabilities, and thus, consider further work in this area outside of the scope of our contribution.

The security policy, along with the firmware sample itself, represent the inputs to Firmalice.

4.5 Static Program Analysis

Symbolically executing entire binary firmware images is not feasible due to the size of the firmware of complex embedded devices. Instead of analyzing entire binaries, Firmalice focuses on the portions of binaries that are relevant to authentication bypass vulnerabilities. Specifically, the symbolic execution step only needs to be carried out on the parts of the firmware leading to a *privileged program point* in the firmware. Firmalice isolates this code by creating a *slice* through the firmware. Specifically, Firmalice creates a *backward* slice, starting from the privileged program point, backwards to an entry point in the firmware.

The static analysis module requires as input the loaded firmware sample (produced by the Firmware Loading module, described in Section 4.3). The actual slicing step also requires the address of one or more *privileged program points*. These should be instructions in the firmware that should only be reached by authenticated users. As we discuss in Section 4.4, privileged program points are derived from an analyst-provided security policy.

The identification of privileged program points specified by a security policy, and the creation of backward slices leading to them, requires the use of a program dependency graph (PDG) to reason about the control and data flow required to arrive at a specific point in the program. The program dependency graph comprises a data dependency graph (DDG) and a control dependency graph (CDG). Those, in turn, require a control flow graph to be created.

4.5.1 Control Flow Graph

The first step in creating a PDG is the creation of a CFG, a graph of program basic blocks and transitions between them. Firmalice creates a context-sensitive CFG by statically analyzing the firmware, starting from each of the entry points and looking for jump edges in the graph. Firmalice can support computed and indirect jumps (including jump tables) by leveraging its *Symbolic Execution* module, described in Section 4.6. Firmalice's analyses are performed with a *call-site* context sensitivity of 2, to improve the precision of the static analysis. This threshold for the call-site context sensitivity can be changed at the expense of an exponential runtime increase, but, in practice, we have found that a threshold of 2 works well for the firmware samples that we analyzed.

Firmalice leverages several techniques to increase the precision of its control flow graph. During CFG generation, Firmalice utilizes *forced execution* to systematically explore both directions of every conditional branch [87]. When it encounters a computed or indirect jump, Firmalice can leverage its symbolic execute engine (which will be described in Section 4.6) to reason about the possible targets of that jump. By doing this, Firmalice is able to handle complex control flow transfers, such as jump tables. In turn, a precise CFG has a trickle-down effect on the precision of the rest of Firmalice's analysis.

Firmalice stores the context-sensitive CFG as a graph, in which the *contexts* are nodes and edges represent control flow transfers between these contexts. This means that the graph might contain several distinct instances of a basic block γ with a control transfer edge to basic block α, as long as the call-sites of α and γ differ.

4.5.2 Control Dependency Graph

A control dependency graph represents, for each statement X (generally, a binary instruction, but in our case, an IR statement), which other statements Y determine whether X is executed. Together with the CFG, the CDG can be used to identify statements that may be executed before a given statement is executed.

Again, we use a context sensitivity of 2 when generating the CDG, which allows Firmalice to reason about not only basic blocks that may be executed so that a given statement is reached, but also the call context from which those basic block would be executed. The CDG is generated via a straightforward transformation of the CFG [88].

The CDG is not used directly, but is combined with the data dependency graph to create the PDG.

4.5.3 Data Dependency Graph

A data dependency graph (DDG) shows how instructions correlate with each other with respect to the production and consumption of data. Efficiently generating a sound DDG for a binary slice has several challenges. First, program slicing requires a flow-sensitive and context-sensitive data flow analysis, with a runtime complexity exponential to the number of all possible paths in a program. Second, analyzing the data flow of binary programs poses some unique problems. For instance, the precision of the DDG suffers from any imprecision in the CFG from which it is built, and creating a precise CFG statically is a hard problem for arbitrary binary code. Additionally, all information about data structures and types is discarded during compilation, which makes performing a sound data flow analysis even harder. Thus, most data flow analyses are designed to work with high-level languages, but not with binary code. Finally, the analysis result should be sound, otherwise one risks removing instructions that are otherwise required to achieve a proper result.

To handle the issues mentioned above, Firmalice adopts an existing, worklist-based, iterative approach to data flow analysis [89]. The approach is an inter-procedural data flow analysis algorithm that uses *def-use* chains, in addition to *use-def* chains, to optimize the worklist algorithm.

As with the other algorithms in the static analyses phase, the DDG is generated with a context sensitivity of 2.

4.5.4 Backward Slicing

Using the program dependency graph, Firmalice can compute backward slices. That is, starting from a given program point, we can produce every statement on which that point depends. This step leverages slicing techniques from existing work in the literature [88]. Slicing is used to improve the feasibility of the symbolic analysis on large binaries, in two ways. First, it removes entire functions that are irrelevant to the analysis. Since symbolic analysis, in the general case, must explore every path of a program, this represents a substantial decrease in analysis complexity. Second, since our IR translates complex instructions into multiple simple statements, Firmalice's slicing allows one to ignore irrelevant side-effects of these instructions. This is especially relevant for architectures that implicitly update conditional flags (specifically, ARM, x86, and AMD64), as it frees Firmalice from the need to evaluate the flag registers when they are not used (which, on such architectures, is the common case).

4.6 Symbolic Execution Engine

After an authentication slice is created by the *Static Program Analysis* module, Firmalice attempts to identify user inputs that successfully reach the privileged program point. Recall that an authentication slice is a set of instructions between a proposed entry point and the privileged program point that the attacker tries to reach. To enable our analysis, we have implemented a Symbolic Execution Engine. Our approach to symbolic execution draws on concepts proposed in KLEE [22], FuzzBALL [90], and Mayhem [24], adapted to our specific problem domain.

Specifically, the implementation of this module of Firmalice follows ideas presented in Mayhem, adding support for symbolic summaries of functions (described in paragraph 4.6.2), to automatically detect common library functions and abstract their effects on the symbolic state. This greatly reduces the number of paths that the symbolic executor must explore, since it prevents such functions from causing the analysis to branch.

We discuss several details specific to our symbolic execution engine in this section.

4.6.1 Symbolic State and Constraints

Firmalice's symbolic analysis works at the level of symbolic *states*. A symbolic state is an abstract representation of the values contained in memory (*e.g.*, variables), registers, as well as constraints on these values, for any given point of the program (*i.e.*, each program point has an independent state).

Constraints are expressions limiting the range of possible values for a symbolic variable. They may express relations between symbolic variables and constants (i.e., `x < 5`) or between multiple symbolic values (i.e., `x < y + z`).

For user-space firmware processes, the state also contains other program information, such as the status of open files. States are modified by symbolic translations of IR representations of binary instructions that consume an input state and produce one or, in the case of conditional or computed jumps, multiple output states. As the execution goes following paths in the program, Firmalice keeps tracks of symbolic constraints in a set of *path constraints*. Whenever a path reaches the privileged program point, its associated state is labeled as a *privileged state* and passed to the Authentication Bypass Check module for further analysis, based on constraint solving[6]. The term *constraint solving* refers to the problem of finding concrete or symbolic solutions that satisfy a set of constraints on a variable (e.g., determining, in the case of `x < 5 && x >= 0`, that `x` can be 0, 1, 2, 3, or 4).

4.6.2 Symbolic Summaries

Firmalice adopts the concept of "symbolic summaries", a well-known concept in program analysis, which involves descriptions of the transformation that certain commonly-seen functions (or, generally, any piece of code) have on a program state [92]. The intuition behind this concept is that the effects of certain functions can be more efficiently explained through a manual specification of constraints than by analyzing the underlying binary code. This is because an initial analysis of a piece of binary code lacks a *semantic* understanding of what that code is trying to accomplish. A process that had such an understanding, however, could analyze the code as a whole and introduce constraints that took these semantics into account. In fact, we found that such a process has two advantages: properly summarizing the code allows us to avoid branching the analysis state during the execution of such functions, and the constraints that are generated are

[6]Firmalice utilizes Z3 [91] to perform symbolic constraint solving.

often simpler than those that would be generated from an analysis of the code itself.

To explore this concept in our analysis, we implemented support for symbolic summaries in Firmalice. A symbolic summary acts in the same way as a binary instruction: it consumes an input state and produces a set of output states. We implemented symbolic summaries for 49 common functions from the Standard C Library.

While this concept is well-known in the field of program analysis, applying it to automatic binary analysis is not trivial, as Firmalice needs to know which pieces of code should be replaced by these summaries. To determine this automatically, we created a set of test cases for each of the functions that we summarized. These test cases, comprising an input state (representing a set of arguments to the function) and a set of checks of its effect on this state, attempt to determine whether or not an arbitrary binary function is an implementation of the function summarized by the symbolic summary in question.

Generally, more than one test case is required to uniquely identify a library function. For example, several different test cases are required to distinguish between `strcmp()` and `strncmp()`, since the two functions act in the same way for certain sets of inputs (lower case strings for example). Similarly, multiple test cases are required to differentiate between `memcpy()` and `strncpy()`. While this represents more work when writing test cases, it also allows us to speed up the testing procedure, because if a function fails a test case that should be passed by both `memcpy()` and `strncpy()`, we can conclude that it is neither of those functions.

When Firmalice symbolically calls a function for the first time (i.e., analyzing a call instruction), the analysis is paused and the function-testing phase begins. Firmalice first attempts to run the function with the test case states. If all of the test cases of a symbolic summary pass, Firmalice replaces the entry point to the function in question with that symbolic summary, and continues its analysis. Any subsequent jumps to that address will instead trigger execution of the symbolic summary. If no symbolic summary is identified as the right summary for a function, the function is analyzed normally. The test cases should be mutually independent across all symbolic summaries. That is, for any given function, if all test cases of symbolic summary A pass, then there must be no summary B for which all test cases also pass. Such situations arise in the case of inadequate test cases, and must be remedied before Firmalice can properly detect symbolic summaries.

While symbolic summaries allow Firmalice to perform a considerably deeper analysis than would otherwise be possible, there is a trade-off. Because we do not fully analyze the summarized code, our approach would miss any backdoors that were hidden in common library functions. We feel that this trade-off is acceptable.

4.6.3 Lazy Initialization

Binary-blob firmware contains initialization code that is responsible for setting various memory locations to initial values, setting up request handlers, and performing other housekeeping tasks. However, since Firmalice has no prior knowledge of such code[7] it is not executed before beginning the analysis, leading to complications when, for example, kernel-level functionality of firmware attempts to access certain global data structures. If such data structures are not initialized, superfluous paths based on normally infeasible kernel conditions are introduced into the analysis.

To mitigate this, Firmalice adopts a lazy approach to firmware initialization. When the execution engine encounters a memory read from uninitialized memory, it identifies other procedures that contain direct memory writes to that location, and labels them as *initialization procedures*. If an initialization procedure is identified, the state is duplicated: one state continues execution without modification, while the other one runs the initialization procedure before resuming execution. This allows Firmalice to safely execute initialization code without the risk of breaking the analysis.

[7]the execution starts after the input related to the authentication routine

4.7 Authentication Bypass Check

As discussed in Section 5.2, our model of an authentication bypass builds upon the property of *input determination*. That is, if an attacker can analyze the firmware and produce inputs, possibly including valid authentication credentials, to reach a privileged program point, an authentication bypass is said to exist.

This model is not dependent on the implementation of the backdoor itself, but rather on the fundamental idea behind authentication bypass vulnerabilities: the attacker can create an input that, regardless of the configuration of the device, will allow them to authenticate (*i.e.,* reach a privileged program point).

To detect such bypasses, Firmalice leverages the property of *constraint solvability* with respect to the user input required to achieve authentication. Specifically, we model the *determinism* of the input with the ability to concretize it to a unique value, as described in Section 4.7.3. However, we make this determination after taking into account the *exposure* of data from the device, in the form of output to the user. Thus, even in the presence of a challenge-response protocol, Firmalice can detect an authentication bypass vulnerability.

This model can also be expanded to reason about authentication bypasses with a range of valid backdoor credentials. However, as we have not observed this in practice, we did not include such detection in our implementation.

Given an *privileged state* (i.e., the final state of a path that reaches a privileged program point) from the Symbolic Execution engine, the Authentication Bypass Check module identifies the input and output from/to the user and reasons about the *exposure* of data represented by the output. It then attempts to uniquely concretize the user input (*i.e.,* to solve the constraints associated to the user input when the *privileged state* is reached). If the user input can be uniquely concretized, then it represents that the input required to reach the *privileged program point* can be uniquely determined by the attacker, and the associated path is labeled as an *authentication bypass*. At this point, Firmalice terminates its analysis. In cases where the user input depends on data exposed by the device's output, a function that can generate valid inputs for a provided output is produced.

4.7.1 Choosing I/O

What should be considered as user input to the firmware (and, similarly, output from the firmware) is not always obvious. For example, devices might have complex interactions with their environment, and receive input in unexpected ways. Therefore, Firmalice uses several heuristics to identify input and output.

If the firmware is a user-space firmware, Firmalice checks for the presence of network connections in the privileged slice. If a connection is found, it is assumed to represent the user input. Alternatively, if no connection is found, user input is assumed to be stdin (file descriptor 0), and output is assumed to be stdout (file descriptor 1).

In the case of a binary blob, Firmalice attempts a concretization on symbolic values coming from every interrupt. If one of these inputs concretizes mainly to ASCII text, it is considered to be the user input. Similarly, any symbolic value passed into an interrupt that concretize mainly into ASCII text, is considered to be the output of the firmware. Alternatively, to avoid these heuristics, Firmalice can accept a specification of the Application Binary Interface (i.e., which interrupts accept output and which provide input) of the firmware and use that to choose between input and output.

4.7.2 Data Exposure

The core intuition of our approach is that data seen by the user, via an output routine, is *exposed* to the attacker. While seemingly obvious, this has important implications for authentication bypass detection. Specifically, our intuition is that this exposure does not just reveal information about the output data: information is also revealed about any data that *depends on* or is *related to* the output. For example, if a hash of a user-specified, secret password is revealed to the attacker prior to authentication, it reveals some amount of information about the password itself (in the worst case scenario, such a hash could then be brute-forced and the password would be completely revealed). In essence, we take into account the fact that the attackers can deduce information about authentication credentials by observing program outputs.

We implement this in Firmalice by leveraging its constraint solver and output routine detection. Any data, D, that is passed into an output routine is identified as having been exposed. To model this exposure, we use the constraint solver to retrieve a single concrete solution, C, for D, and add the constraint $D == C$ to the constraint set. Adding this constraint has an effect on the concrete solutions associated with other symbolic variables (for example, if a symbolic variable E previously existed with a constraint $E == D$, then the constraint $D == C$ also implies $E == C$). This represents any loss of secrecy that these variables experience from the revelation of D to the attacker.

To avoid false positives from after-the-fact credential revelation on the part of the firmware, Firmalice only applies this policy to data that is output *before* any user input is received.

4.7.3 Constraint Solving

For each privileged state, Firmalice attempts to concretize the user input to determine the possible values that a user can input to successfully reach the privileged program point. A properly-authenticated path contains inputs that concretize to a large set of values (because the underlying passwords that they are compared against are unknown, and thus, unconstrained). Conversely, the existence of a path for which the input concretizes into a limited set of values (for simplicity, and from investigating existing examples of backdoors, we set this threshold to 1) signifies that an attacker can determine, using a combination of information within the firmware image and information that is revealed to them via device output, an input that allows them to authenticate.

Since Firmalice limits its analysis to the authentication slice itself, irrelevant data is not included in the produced user input. This makes Firmalice resilient to cases that would be *arbitrarily non-deterministic*, such as when some data from the user is ignored or not used (and, thus, concretizes to no specific value). While this means that Firmalice's output might not be directly re-playable to achieve authentication bypass, this functionality is outside of the scope of our design.

4.8 Evaluation

We evaluated Firmalice by vetting three devices for authentication bypass vulnerabilities, two of which had actual backdoors. These devices, the Schneider ION 8600 smart meter, the 3S Vision N5072 CCTV camera, and the Dell 1130n Laser Mono Printer, represent a wide range of devices of disparate architectures. ARM (both little-endian and big-endian) and PPC are both represented, as are both binary-blob and user-space program firmware styles. Additionally, the devices have widely different authentication processes.

We chose these devices because the authentication vulnerabilities that they contain were already discovered manually, and, since these vulnerabilities have already been released, we are not endangering the users by discussing them (and providing examples). We chose three devices because, despite the fact that Firmalice's analysis is automated, a security policy needs to be provided for each device. This represents some manual work, and a truly large-scale study was infeasible. Additionally, collecting and unpacking firmware samples is extremely complicated to automate. Firmware is shipped in many different, non-standard formats, and the process to download firmware images is frequently complicated, and varies from vendor to vendor. While this is an addressable problem, as shown by Costin *et al.* [93], we consider it outside of the scope of our work. However, we feel that these samples represent Firmalice's applicability to different devices of different architectures.

In this section, we will describe each firmware, then detail their user interaction, present our analysis results, and describe any backdoors that Firmalice identified. Aside from the device-specific uses of these backdoors, each one can also be used as a pivot point into the victim's network. The nature of some of these devices means that they are frequently either physically positioned *outdoors*, exposed directly to the Internet, or are otherwise not closely monitored, making them a prime target for attackers.

We carried out this evaluation on our prototype of Firmalice, comprising over 14,000 lines of Python and 3,000 lines of C. Our implementation is single-threaded, although the approach itself would scale near-linearly in the symbolic analysis phase. Thus, the execution time presented in this section is representative of what can be accomplished using a *single node* of Firmalice, and significant improvements in runtime can be achieved by parallelizing the symbolic execution.

Measurement	**ION**	**3S**	**Dell**
Total size (KB)	1,988	1,264	7,172
Basic blocks (total)	74,808	10,354	151,005
Basic blocks (slice)	1,144	212	532
Slice (statements)	56,977	7,808	24,387
Static analysis time (seconds)	2,323	315	857
Symbolic execution time (minutes)	12	26	705

Table 4.2: The results of Firmalice's analysis for the ION 8600, the 3S Vision N5072 and the Dell 1130n.

4.8.1 Schneider ION 8600 Smart meter

As the smart meter market exploded worldwide, Schneider Electric corporation released the ION 8600, a smart meter model meant for both residential and commercial use. Such devices play a privacy-critical and safety-critical role: the information that they process can be used to determine the habits of a home's resident, and any malicious tampering can cause extremely dangerous situations due to the amount of electricity involved.

A researcher from IOActive Labs presented a backdoor in the Schneider ION 8600 smart meter model at BlackHat in 2012 [84]. The backdoor was identified through manual static analysis of the firmware. Schneider Electric acknowledged the backdoor in a press release [94] and released an updated firmware image. Our interpretation of the presentation by IOActive, and the press release by Schneider, led us to think that the backdoor was remotely exploitable.

We saw this as a great opportunity to verify Firmalice's functionality. Even better, the described authentication procedure is relatively complex: rather than being a simple comparison against a hardcoded string, it relies on the exposure of the backdoor credentials (which are dynamically generated by hashing the serial number of the device) to the user during the authentication process. Detecting this type of authentication bypass requires reasoning about the *determinism* of the authentication credentials, in relation to information provided by the device during the authentication process.

The security policy. We observed that the ION would output the string "Access Granted" upon a successful authentication by a user. This was leveraged for the security policy: we set the authenticated point to the location in the firmware where "Access Granted" was printed.

The analysis. This firmware's binary blob contained 1,988 kilobytes of binary code spanning 74,808 basic blocks. The static analysis completed in about 38 minutes, and the resulting authentication slice contained 1,144 basic blocks and 56,977 statements.

The authentication slice identified by Firmalice ran from the input routine to the privileged point. Because the ION's firmware places a bound on the size of the user input, and because symbolic summaries of functions greatly reduce the number of branches that Firmalice must analyze, Firmalice was able to *exhaustively* analyze all paths through the authentication slice. This symbolic analysis ran in 12 minutes, analyzing 1,029,156 statements in 23,044 blocks across all analyzed paths. We present these results in Table 4.2.

To our surprise, Firmalice's analysis yielded no bypasses. Since symbolic analysis is, in practice, not sound, we assumed that our system must have missed the vulnerability. We manually analyzed the firmware sample, and even attempted the bypass on an acquired device with the vulnerable firmware (verified by the build number and release date) to try to figure out where Firmalice was getting confused. It turned out that, due to complex logic in the authentication routine (which spanned several nested functions with intricate interactions), a user had to be already authenticated with *valid* credentials before using the hardcoded credentials identified by IOActive. Using the hardcoded credentials, after an actual, secure authentication, would grant access to more features of a device. However, the backdoor account could not be accessed from the Internet, unless the attacker already had the user's actual, valid credentials. Therefore, there was no remotely accessible backdoor.

We contacted the IOActive researcher, and he confirmed that we were mistaken in our interpretation. We feel that this anecdote demonstrates the need for a more automated solution: even with manual analysis, it took us a significant amount of time to verify the results of our analysis due to the complexity of the code involved. Given the difficulty (and cost) involved in updating firmware on embedded devices, such mistakes can represent a real financial impact, and a system to automate parts of this analysis can be extremely valuable.

4.8.2 3S Vision N5072 Camera

The 3S Vision N5072 is a CCTV camera with networking functionality.

In April 2014, Craig Heffner presented backdoors in several common embedded devices at the EELive 2014 conference [95]. Among them was the N5072 camera from 3S Vision. This backdoor, which takes the form of a hardcoded authentication credential, allows an attacker to control and view the camera over the network. Especially given the zooming capability of this camera, such an attack can have serious implications with regards to privacy intrusion.

The camera is built on a little-endian ARM architecture. We found that the firmware of this camera is actually an embedded Linux system, comprising Busybox and several camera-specific binaries, including a custom web server.

The security policy. Our security policy for this firmware reflected the purpose of the device itself: the user must not be able to view camera footage without authentication. However, the footage itself was not static, so we could not directly use it for the policy. Instead, we used the static string "Image Type:", which was included when requesting footage from the camera's web interface.

The analysis. Firmalice was able to identify the backdoor in the `httpd` binary, in a total of 31 minutes. This binary, and the libraries that it depends on, contain a total of 1,264 kilobytes of binary code spanning 10,354 basic blocks. The static analysis completed in 315 seconds, and the resulting authentication slice contained 3,553 statements from a total of 7,808 in the corresponding 212 basic blocks. The detection of the backdoor took just over 26 minutes, analyzing 550,660 statements in 34,544 blocks across all executed paths. We present these results in Table 4.2.

The Backdoor. The backdoor in the N5072 was a hardcoded authentication credential during HTTP authentication. The backdoor allows an attacker to stream video from the camera and modify the camera's configuration. Firmalice provided an HTTP request that would be sufficient to reach the privileged program point, in which an "authorization" parameter is passed in the query string. The base64 decoding of the authorization query string parameter is "3sadmin:27988303", which is the hard-coded username and password of the backdoor. Interestingly, Firmalice also stumbled upon a benign bug in the URL parsing code of the camera: query string parameters are parsed, even without the presence of the "?" character that denotes the start of a query string, if the provided query path is blank.

4.8.3 Dell 1130n Printer

The Dell 1130n is a network-connected laser printer popular in many office and academic settings. Such printers are frequently connected directly to the Internet, with no protection or filtering in place. In fact, in January 2013, researchers made headlines by pointing out the presence of 86,800 network printers that could be found in a Google search [96].

A backdoor affecting a range of printers manufactured by Samsung, including the Dell

1130n, was discovered in 2012 [85]. This backdoor allows an attacker to change printer settings, intercept documents sent to the printer, and use the printer as a pivot point into the victim's network. The backdoor is triggered by sending a specially-crafted SNMPv1 packet to the printer, with a hardcoded *community string*. This attack works even when SNMP is turned off.

This printer runs on a big-endian ARM CPU, and its firmware is a modified VxWorks binary-blob containing 7,172 kilobytes of binary code across 151,005 basic blocks.

The security policy. We used the printer to evaluate our more fine-grained security policy, defining a memory region, containing configuration parameters, that should not be changed by unauthenticated users. Firmalice identified all program points that write to this memory region, and tagged them as *privileged program points* for the analysis.

The analysis. Firmalice finished its static analysis in just over 14 minutes, and created an authentication slice that contained 13,592 of the total 24,387 IR statements in 532 blocks. The analysis of the slice took 11 hours and 45 minutes, executing a total of 134,536,875 statements in 4,264,568 blocks across all of the analyzed paths. The results are presented Table 4.2.

The Backdoor. The backdoor in the 1130n took the form of a specially crafted SNMP packet, allowing the attacker administrative access to the printer. Firmalice provided an input representing the SNMP packet that would let the attacker reach the privileged program point.

4.9 Discussion

In this section, we discuss the implications and the limitations of Firmalice and muse about several ideas for future research directions.

Firmalice's target application is the analysis of authentication bypass vulnerabilities in firmware. In general, such software is not actively evasive (unlike, for example, traditional malware), and lends itself well to static analysis. However, it is possible that a malicious firmware author could attempt to attack Firmalice's analysis. There are two main attack surfaces: the static program slicing and the symbolic execution. Obfuscated firmware could frustrate the former, while specially-crafted operations (designed to overwhelm the constraint solver) could attack the latter. These are weaknesses inherent to any tool based on static slicing or symbolic execution, and Firmalice is also vulnerable to them. Given the status quo in firmware, the presence of such obfuscation or evasive code would be, by itself, an excellent indicator of maliciousness, which Firmalice could be adjusted to detect.

As an alternative to binary obfuscation, malicious firmware authors could attempt to evade Firmalice's input determinism by performing *irreversible* operations. For example, Firmalice's constraint solving module would be unable to solve the constraints generated by a secure hash function, as doing that would be equivalent to reversing the function. As a result, Firmalice, in its current implementation, can be evaded by an authentication bypass that compares a hash of the user's password against the hash of a hard-coded password. A possible mitigation of this evasion is the replacement of the hash function with a symbolic summary that performs a reversible "summary hash". In the case of SHA-256, such a summary hash might simply expand or truncate the input to 256 bits. With this summary hash replacing the original hash function, the constraints generated by Firmalice would be reversible, and the required user input could be identified. However, this represents a large sacrifice in accuracy of the analysis, and false positives could be introduced as a result.

There are also other types of backdoors that Firmalice might fail to detect. Specifically, math-based backdoors with multiple solutions (e.g., "the password must be an integer that is divisible by 10") would, as a result of having multiple valid solutions, be considered as a “correct” authentication. To reason about such backdoors, Firmalice would need to reason about how *restricted* a set of solutions is. This ability would involve extra complexity related to constraint solving and we feel that this analysis is outside of the scope of this research.

Throughout Firmalice’s design, we had to make many trade-offs between soundness and scalability. Symbolic analysis, in general, is infeasible to perform with full soundness, because doing this would mean, in the general case, following every path through a program. This would be exponential in the number of branches, and Firmalice makes trade-offs, similar to other tools in the field.

Many of the challenges that Firmalice must deal with could be addressed through the use of dynamic execution monitoring. For example, Firmalice’s entry point detection would be unnecessary if the entry point could be deduced from observing the boot process of the device. However, the difficulty of this ranges from extremely complex to impossible for most devices. Since many embedded devices require their firmware to be signed by the device manufacturer, loading custom analysis code (such as that required by Avatar [77]) would require bypassing this protection. Even if this limitation could be bypassed, the disparity between different devices would necessitate a significant implementation effort to analyze each new device, limiting the possible scale of such a system’s analysis.

While Firmalice is geared towards detecting authentication bypass vulnerabilities in firmware, the core approach lends itself to any logic flaws that can be similarly modeled. One potential direction of research is a formal language to enable the specification of custom logic flaws for Firmalice to locate. In fact, the Defense Advanced Research Projects Agency has launched a project to explore exactly this, with the goal of assuring the security of embedded devices [97]. DARPA’s goal is to eventually be able to specify such models as Natural Language statements that can be converted into logic flaw descriptions.

Firmalice, and symbolic analysis in general, can be greatly improved by a better approach to symbolic loop analysis. When analyzed symbolically, a loop has the potential to branch analysis states at each iteration (one that exits the loop and one that does not), causing a state explosion. Firmalice partially mitigates this through the use of its symbolic summaries, as many of the loops encountered during a program's execution are actually within common library functions. However, in the general case, advances in loop analysis would directly benefit Firmalice's (and other analyzers') analyses.

4.10 Related Work

While a number of previous efforts have been focusing on analyzing binary applications on commodity software and hardware platforms, including general frameworks such as Valgrind [54], BitBlaze [98], and Pin[99], as well as symbolic execution based frameworks like AEG [86] and Mayhem [24], focusing on automatic exploit generation on binary programs, the case of embedded firmware received little attention and remains challenging. Among existing research on firmware analysis, the current systems either require access to the source code [76] (which in the case of embedded systems is rarely available), or to the physical device [82, 77].

Schuster *et al.* [82], proposed an approach for automatically identifying software backdoors in binary applications running on x86, x64, and MIPS32 architectures. This approach targets flawed authentication routines as well as commands and services hidden in server-side binaries such as FTP and SSH. The approach builds on top of execution monitoring using GDB, and requires actual execution of the binaries on the target physical system, making it difficult to generally apply the technique to embedded devices. While this work proposes a practical approach to detecting backdoors, it is limited to a specific kind of authentication bypass technique where pointers to *handlers* are actually present as-is in memory[8]. Additionally, Schuster models authentication bypass as a control flow problem, leaving them unable to reason about authentication bypasses resulting from disclosed credentials or buggy authentication routines. Not only is our system able to analyze binaries with no hardware requirements, but our symbolic execution approach also targets a wider range of malicious behaviors. In fact, the *authentication deciders* and *command handlers* that Schuster's approach identifies during the analysis can be used as a security policy by Firmalice, allowing Firmalice to vet the firmware against complex authentication bypass vulnerabilities.

Avatar [77] is a framework supporting dynamic analysis of firmware in embedded systems. It is a hybrid approach, involving both the target physical device as well as an emulator based on the selective symbolic execution engine S2E [100]. Communication between the emulator and the target is orchestrated in such a way that I/O operations can be forwarded and executed on the actual hardware and interrupts injected into the emulator. Arbitrary context switches are also supported: execution can be started on the real device and transfered to the emulator for analysis from a specific point in the firmware. Returning execution to the hardware is also supported. In both cases, the execution state is frozen and transferred from/to the hardware or the emulator. While Avatar presents promising capabilities and support for reverse engineering and vulnerability discovery, it requires access to the physical hardware, either through a debugging interface, or by installing a custom proxy in the target environment, which is generally not possible, *e.g.,* in the presence of locked hardware. Our framework is an alternative to such hardware-dependent approaches, by providing a model along with tools for analyzing such firmware with no hardware requirements.

FIE [76] is a platform for detecting bugs in firmware running on the MSP430 family of micro-controllers, mainly focusing on memory safety issues. The source code of the

[8]This approach does not address the cases of obfuscated or indirect addresses to such *handlers*.

analyzed programs is compiled into LLVM bytecode, which is then analyzed using a symbolic execution engine based on KLEE [22]. The latter has been modified to support the target 16-bit architecture, its memory specification, and its interrupt library. FIE supports hardware specific layouts of memory and access to hardware through *special memory*. It also considers the execution of enabled interrupts at any given point in the program. It performs *complete* analysis of firmware images (*i.e.,* all possible execution paths are taken). In order to achieve this without falling into infinite loops or state explosion, *state pruning* is used, removing redundant (equivalent) states from the list of states to explore, and *memory smudging* is used to concretize variables with respect to a given finite set of values. FIE is limited to analyzing small firmware written in C, for which the source code is available. In comparison, our current work is not bound to any specific architecture (in fact, our symbolic execution engine currently supports multiple architectures) and works directly on binary code with no source code requirement.

Recent advancements have also been made in the field of automated firmware analysis. Costin *et al.* [93] carried out an analysis of over 30,000 firmware samples. However, their system performs no in-depth analysis: it instead extracts each firmware sample and investigates it for artifacts such as included private encryption keys and "known-bad" strings (i.e., known values of hardcoded authentication credentials). This latter action makes the system quite well-suited for discovering backdoors in devices whose firmware shares a codebase with devices that have known backdoors, but not for in-depth analysis of individual firmware samples. With a further investment into analysis automation, Costin's system could be used as an input to Firmalice, allowing for large-scale, automated, in-depth firmware analysis.

4.11 Conclusion

We presented Firmalice, a framework for detecting authentication bypass vulnerabilities in binary firmware, for which no source code, and possibly no access to the underlying hardware, is available. Additionally, we have presented a model of authentication bypass vulnerabilities (or backdoors), based on the concept of *input determinism* and have shown that Firmalice is capable of successfully detecting such vulnerabilities in the firmware of two commercially-available systems. Finally, we have demonstrated that current techniques for identifying authentication bypass in firmware, which are mostly limited to manual analysis, are error-prone and insufficient.

Acknowledgements. This work is sponsored by DARPA under agreement number N66001-13-2-4039 and by the ARO through MURI grant W911NF-09-1-0553. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

Additionally, we would like to thank Timon Van Overveldt for his assistance with the selection, extraction, and initial analysis of firmware samples.

Chapter 5

Putting Humans Back in the Loop

Software has become dominant and abundant. Software systems support almost every aspect of our lives, from health care to finance, from power distribution to entertainment. This growth has led to an explosion of software bugs and, more importantly, software vulnerabilities. Because the exploitation of vulnerabilities can have catastrophic effects, a substantial amount of effort has been devoted to discovering these vulnerabilities before they are found by attackers and exploited in the wild.

Traditionally, vulnerability discovery has been a heavily manual task. Expert security researchers spend significant time analyzing software, understanding how it works, and painstakingly sifting it for bugs. Even though human analysts take advantage of tools to automate some of the tasks involved in the analysis process, the amount of software to be analyzed grows at an overwhelming pace. As this growth reached the scalability limits of manual analysis, the research community has turned its attention to *automated program analysis*, with the goal of identifying and fixing software issues on a large scale. This push has been met with significant success, culminating thus far in the DARPA Cyber Grand Challenge (CGC) [39], a cyber-security competition in which seven finalist teams pitted completely autonomous systems, utilizing automated program analysis techniques, against each other for almost four million dollars in prize money.

By removing the human factor from the analysis process, the competition forced the participants to codify the strategy and orchestration tasks that are usually performed by experts, and, at the same time, it pushed the limits of current vulnerability analysis techniques to handle larger, more complex problems in an efficient and resource-aware manner. These systems represented a significant step in automated program analysis, automatically identifying vulnerabilities and developing exploits for 20 of a total of 82 binary programs developed for the event.

Despite the success of these systems, the underlying approaches suffer from a number of limitations. These limitations became evident when some of the CGC autonomous systems participated in a follow-up vulnerability analysis competition (the DEFCON CTF) that included human teams. The autonomous systems could not easily understand the logic underlying certain applications, and, as a result, they could not easily produce inputs that drive them to specific (insecure) states. However, when humans could provide "suggestions" of inputs to the automated analysis process the results were surprisingly good.

This experience suggested a shift in the current vulnerability analysis paradigm, from the existing tool-assisted human-centered paradigm to a new human-assisted tool-centered paradigm. Systems that follow this paradigm would be able to leverage humans (with different level of expertise) for specific well-defined tasks (e.g., tasks that require an understanding of the application's underlying logic), while taking care of orchestrating the overall vulnerability analysis process.

This shift is somewhat similar to introduction of the assembly line in manufacturing, which allowed groups of relatively unskilled workers to produce systems (such as cars) that had, until then, remained the exclusive domain of specially trained engineers. Conceptually, an assembly line "shaves off" small, easy tasks that can be carried out by a large group of people, in loose collaboration, to accomplish a complex goal.

In this chpater, we explore the application of this idea to vulnerability analysis. More

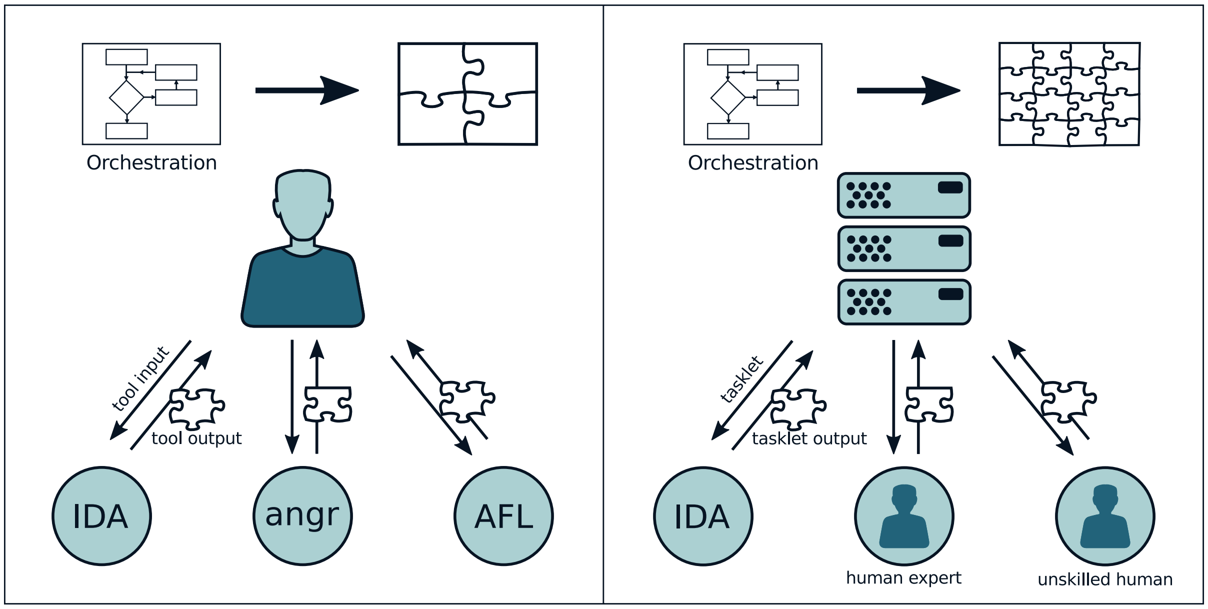


Figure 5.1: Tool-assisted Human-centered Analysis vs. Human-assisted Tool-centered Analysis.

precisely, we develop an approach that leverages *tasklets* that can be dispatched to human analysts by an autonomous program analysis system, such as those used in the Cyber Grand Challenge, to help it surmount inherent drawbacks of modern program analysis techniques (see Figure 5.1). We explore the question of how much our "program analysis assembly line" empowers humans, otherwise unskilled in the field, to contribute to program analysis, and we evaluate the improvement that external human assistance can bring to the effectiveness of automated vulnerability analysis[1]. Our results are significant: by incorporating human assistance into an open-source Cyber Reasoning System, we were able to boost the amount of identified bugs in our dataset by 55%, from 36 bugs (in 85 binaries) using fully-automated techniques to 56 bugs through the use of *non-expert* human assistance.

In summary, this chapter makes the following contributions:

[1]In the rest of the chapter, we refer to "automated vulnerability analysis" as the orchestration process, even though it might include tasks that are outsourced to humans.

- We introduce the design of a *human-assisted automated vulnerability analysis system*, in which the result of well-defined *tasklets* that are delegated to human actors are integrated in the (otherwise) autonomous analysis process. These tasklets help automated analysis systems to bridge the "semantic gap" in the analysis of complex applications.

- We implemented a prototype human-assisted autonomous system on top of Mechanical Phish, a system that participated in the DARPA Cyber Grand Challenge, which was open-sourced by its authors. To support the community and drive the state of (semi-) automated program analysis forward, we open-source our modifications to Mechanical Phish.

- We experimentally evaluated the effectiveness of our tasklets in aiding the vulnerability analysis process of our system by leveraging the assistance of unskilled humans, showing that significant contribution can be made without requiring expert hackers.

In the next section, we will discuss the background of automated program analysis and pinpoint the challenges that we hope to solve with human-analyzed tasklets.

5.1 Background

The field of vulnerability discovery has received a significant amount of research attention. In this section, we will describe the current state of the art of both automated and manual vulnerability discovery techniques, show the challenges facing each of them, and position our approach in the context of related work.

5.1.1 Fully Automated Analysis

Individual techniques have been developed for identification of vulnerabilities [24, 101, 36], automatic exploitation [67, 102, 68], and automatic application protection [103, 104, 105]. However, until recently, researchers did not focus on the integration of various techniques into cohesive end-to-end systems. Over the last two years, DARPA hosted the Cyber Grand Challenge which required contestants to develop *Cyber Reasoning Systems* (CRSes). These are fully autonomous machines capable of identifying, exploiting, and patching vulnerabilities in binary code.

A Cyber Reasoning System represents the culmination of years of research into automated binary analysis. However, being fully autonomous, CRSes suffer from the limitations of their underlying techniques. These limitations were reflected in the Cyber Grand Challenge results, in which only 20 out of the 87 vulnerable challenges were successfully exploited by the machine contenders [106, 107].

5.1.2 Human-based Computation

While the assembly line pioneered the idea of splitting complex physical tasks (such as the assembly of a car) into small, manageable *micro-tasks* as early as the 12th century [108], the *intellectual* equivalent was not explored until modern times. This concept was most popularized with the Manhattan Project, in which specific computation *micro-tasks* were assigned to and carried out by human "computers" [109]. With the emergence of modern computing capability, these micro-tasks came to be chiefly carried out by machines. As computers developed to the point where they could oversee such efforts, a formal specification of the different roles that humans and computer components can take on in computation emerged [110, 111, 112]. This specification defines three roles:

Organization Agent. The organization agent is the *overall intelligence*. It tracks the progress of work toward an overarching goal, determines what should be done, and creates micro-tasks. In the Manhattan Project, the organization agent was the panel of scientists leading the research effort.

Innovation Agent. The innovation agent is the entity responsible for carrying out micro-tasks defined by the organization agent. In the Manhattan Project, the innovation agents were the human "computers" solving computation tasks.

Selection Agent. The selection agent collates the results produced by the innovation agents and determines which are valid. In the Manhattan Project, this task was performed by the scientists leading the effort.

Systems are described using three letters, depending on whether a human or computer agent is responsible for each role. For example, an `HCH` designation would imply a system with a human deciding which tasks to execute, a computer executing them, and the human deciding which of the results are useful. In a security context, this might be the human specifying jobs to a symbolic execution engine, and then analyzing its output to identify exploitable bugs in a piece of software.

Over the last few years, the Internet has achieved enough saturation to support complex combinations of human and computer agents. For example, Amazon's Mechanical Turk provides an API for automatically specifying micro-tasks for human consumption [113], usually used in a `CHC` context. In fact, we use Mechanical Turk for many of our experiments in this chapter. In a similar vein to Mechanical Turk, specific-purpose platforms have been created to leverage human effort in the pursuit of a single overarching goal. One such platform, Galaxy Zoo [114], utilizes human-completed micro-tasks for the classification of astronomical images, while another, Foldit[115], aids protein folding algorithms by having humans play "folding games."

5.1.3 Human-Driven Automated Analysis

Because it is important to understand the interactions between manual and automated processes in binary analysis systems, we provide a few examples of their intersections outside of the context of our work.

Fuzzing. Generational fuzzers, such as Peach [116], attempt to create inputs conforming to a specification that a program is designed to process. Mutational fuzzers, such as AFL [15], mutate previously-known inputs to identify program flaws.

The most common way of creating these inputs and input specifications is *manually*, through human effort. This results in an `HCH` system – a human creates the input specification, the computer performs the fuzzing, and a human analyzes the results.

An example of successful human-computer cooperation in binary analysis is the discovery of the Stagefright vulnerability in the Android multimedia library. This vulnerability was found by repeating the following steps [117]:

Organization - H. The analyst seeds a mutational fuzzer (in this case, AFL), and starts it.

Innovation - C. The fuzzer identifies vulnerabilities in the target application (in this case, the Android multimedia library).

Selection - H. The human collects the vulnerabilities and *fixes them* so that future iterations of the full system will identify *deeper* vulnerabilities.

By repeating this `HCH` process, the analyst was able to identify many high-impact vulnerabilities inside the Android multimedia library, requiring multiple patches and an eventual rewrite of the entire library to fix [118].

5.1.4 Human-*Assisted* Automated Analysis

The Cyber Grand Challenge required a fully autonomous system (`CCC`, by the definitions in Section 5.1.2). This necessitated the development, by participating teams, of complex automation to handle the organizational, innovation, and selection roles. However, we propose that while the organizational and selection roles must be automated to achieve high scalability, some human effort can still be used in the innovation role to mitigate drawbacks currently impacting automated program analysis techniques. That is, our intuition is that it is possible to create a **H**uman-**a**ssisted **C**yber **R**easoning **S**ystem (HaCRS) that would sparingly use human assistance to improve its performance.

HaCRS provides a principled framework for such an integration of manual and automated analysis. It can be modeled as a `C(C|H)C` system: it does most of its work fully autonomously, but relies on human intuition in the innovation phase, when the automated processes get "stuck." In this chapter, we propose that limited human assistance can be used in the scope of otherwise-automated binary analysis systems. While this has been explored in the context of generating inputs for Android applications, it has never been investigated in the context of an other-wise autonomous Cyber Reasoning System [119]. In the next section, we will give an overview of our system, followed by in-depth details and an evaluation of its improvement over fully-autonomous systems from the Cyber Grand Challenge.

5.2 Overview

While DARPA's Cyber Grand Challenge drove the integration of cutting edge automated binary analysis techniques, it also revealed the many *limitations* of these techniques. Our work on HaCRS extends the concept of a Cyber Reasoning System by defining a method for human interaction that compensates for many of these limitations.

Primarily, HaCRS is an autonomous Cyber Reasoning System. However, when it identifies situations that can benefit from human analysis, HaCRS dispatches self-contained *tasklets* and assigns them to human assistants. These human assistants can vary in skill, from abundant low-skill analysts to rare high-skill hackers.

Our HaCRS can dispatch a variety of tasklets to human assistants, depending on changing requirements. Generally, each tasklet includes a specific program that must be analyzed and a request for specific information that the human can extract from this program. These tasklets are created by a centralized orchestration component and disseminated to the assistant through a Human-Automation Link (HAL). In this chapter, as an initial exploration of this idea, we focus on human-assisted input generation, leaving the exploration of other tasklets to future work.

The Cyber Reasoning System. HaCRS is based on *Mechanical Phish*, an open-source Cyber Reasoning System that was created by Shellphish, the hacking team of the SecLab of UC Santa Barbara, and competed in the DARPA Cyber Grand Challenge [106, 120]. Shellphish designed Mechanical Phish as a set of discrete components, providing individual analysis tasks, united by a central component that handles the "overarching intelligence" [106]. This makes it straightforward (though, unfortunately, non-trivial) to extend Mechanical Phish with other analysis techniques, such as tasklet dispatching.

To the interested reader, we describe the relevant design details of Mechanical Phish in Section 5.3.

Human-Automation Link. We extend Mechanical Phish to request assistance, from non-expert humans, in principled ways.

The prototype action that we explore in this chapter is *input generation*. In input generation, input testcases are created through both automated and human-assisted techniques to form a base set of testcases to use in vulnerability discovery. We describe this task, the conveyance of task-specific information in a human-friendly format, and the use of the results in our Human-assisted Cyber Reasoning System in Section 5.4.

Next, we will discuss relevant details of Mechanical Phish before delving into the details of our tasklets. After this, we will evaluate human performance in the execution of these tasklets against automated alternatives derived from the state-of-the-art in program analysis.

5.3 The Cyber Reasoning System

We based our implementation on the Cyber Reasoning System developed for the Cyber Grand Challenge and open-sourced by Shellphish [120]. While Mechanical Phish is composed of modules that are spread over more than 30 different source code repositories, the core design appears to be fairly straightforward [120].

In this section, we will describe Mechanical Phish in terms of the computation framework discussed in Section 5.1.2. First, we will discuss the type of software that Mechanical Phish is designed to analyze. Then, we split the existing design into the Organization Agent, Innovation Agent, and Selection Agent, as defined in Section 5.1. Afterwards, in the next section, we will detail our extensions on top of Mechanical Phish, and the specific points at which we insert human interaction.

5.3.1 Program Analysis Targets

Mechanical Phish was built for participation in the Cyber Grand Challenge. The Cyber Grand Challenge used a custom operating system, DECREE, to ease the imple-

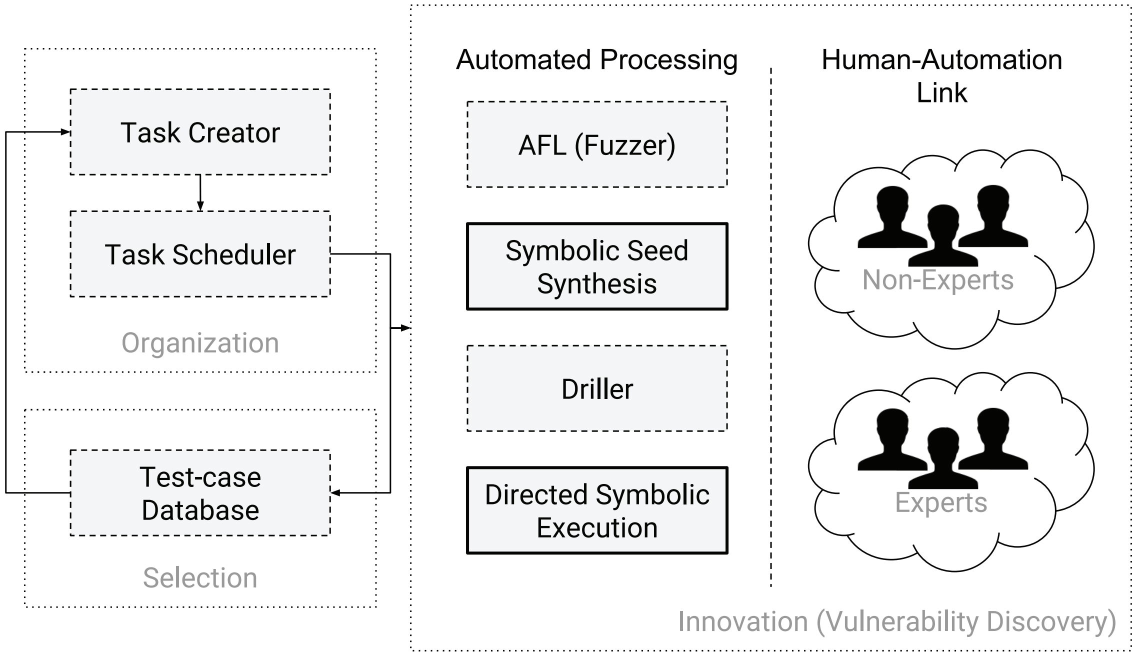


Figure 5.2: The HaCRS configuration. HaCRS builds upon the vulnerability discovery component of the Mechanical Phish and expands it with a Human-Automation Link to leverage non-expert human assistance in the vulnerability discovery process. Each subsystem of the vulnerability discovery component, except for the fuzzer itself, has both an automated and a human-assisted alternative. The components with a dashed border were already present in Mechanical Phish. We created the solid-bordered components for HaCRS.

mentation load on participants. To simplify analysis tasks, DECREE supports software written with a text-based interface, using seven system calls, roughly equivalent to the Linux system calls `exit`, `write`, `read`, `select`, `mmap`, `munmap`, and `getrandom`.

Aside from this simplified environment, DECREE places no restrictions on the complexity of the software itself. As such, applications written for the Cyber Grand Challenge vary widely in complexity, from text-based video games to “Computer-aided design“ software to web servers, and provide significant challenges to the current state-of-the-art in program analysis. Additionally, it is important to stress that all analysis done by HaCRS takes place on *binaries*, and thus functions without the semantic hints present in source code.

5.3.2 Organization Agents

The Mechanical Phish is a *state-less* Cyber Reasoning System, where, for each decision, all of the information available to Mechanical Phish, such as the binaries to be analyzed and the currently-available results of analysis components, is re-analyzed from scratch. According to the authors, this was done in an attempt to reduce the complexity of the organizational components by freeing them from the requirement of tracking their own prior decisions [120].

Mechanical Phish includes several organizational components:

Task Creator. The task creator analyzes currently available results and identifies tasks that should be created, and their priorities. This component is actually a conglomeration of individual, task-specific creators. Each task-specific creator schedules its own tasks without input from other creators: the only interaction between the creators of different tasks happens when results of those tasks influence the current set of analysis results (and, in turn, are used by the subsequent tasks created by these creators).

Task Scheduler. Each task is assigned a priority by its creator. The task scheduler analyzes task priorities and available system resources and determines which tasks to schedule.

Environment Interaction. In order to inject data into Mechanical Phish, and submit the results, interaction with the environment is required. This component handles the retrieval of input into and exposure of output out of the system. While in the CGC this interaction was very straightforward, Cyber Reasoning Systems operating in other environments (for example, in a real-world cyber warfare situation) might require considerably complex agents for this task.

The first task that the system must carry out is the integration of environment information (for example, which binaries are available for analysis), after which the Innovation and Selection Agents can run.

5.3.3 Selection Agents

The selection agents are responsible for the integration of the results that are produced by the innovation agents. However, the Mechanical Phish does not make a distinction between the innovation agents and the integration agents in most cases. One exception is:

Vulnerability triaging. When crashes are identified by the vulnerability discovery component, they are triaged to determine the feasibility of transforming them into exploits. This information is then used by the Task Creator to prioritize exploitation tasks based on the crash.

Exploit selection. The exploits created by the Exploitation Agents are checked against different variations of the target binaries to verify that, for example, opponent systems did not patch the vulnerability. Successful exploits are entered into the database, to be submitted by the Environment Interaction Agent.

Patch selection. Mechanical Phish implements a simple patch selection criteria, preferring patches produced by advanced (but more failure-prone) techniques than simple (but higher-overhead) ones.

The results of these agents are used by the organizational components to schedule further innovation tasks.

5.3.4 Innovation Agents

The tasks that are created and scheduled by the Organization Agents are carried out by the innovation agents. Specifically, Mechanical Phish includes the following agents:

Vulnerability discovery. Mechanical Phish uses a combination of fuzzing and symbolic execution to analyze target binaries. These are implemented as separate agents that interact through cross-pollination of dynamic test cases. Specifically, as proposed by Driller, a coverage-based fuzzer is used in parallel with a symbolic tracing technique to produce inputs that maximize code coverage [36].

Exploitation. Several different exploitation agents are used by Mechanical Phish, depending on the types of vulnerabilities that are discovered.

Patching. Mechanical Phish uses a complex patching agent, in several different configurations, to patch the vulnerabilities that it identifies in binary code.

These innovation agents process inputs and produce updates to the system state. These updates are filtered through selection agents before the system state accepts them.

5.3.5 Automated Vulnerability Discovery - Fuzzing

The fuzzing approach in the Mechanical Phish is based on a mutational fuzzer known as American Fuzzy Lop [15]. This approach requires, as input, a set of test cases that exercise some functionality in the target binary. The seed quality, in terms of how well they exercise the target program, has a scaling effect on the effectiveness of AFL: the more coverage these test cases provide, the more code AFL will be able to explore by mutating them. Unfortunately, the creation of high-quality test case seeds is a complicated problem, and this is generally seen as a human-provided input into a system. For example, lacking human input, Mechanical Phish simply seeds its fuzzer with an input comprised of the word "fuzz."

These seeds are then mutated to explore more and more of the code base and increase the chance of triggering bugs. Eventually, however, the fuzzer will *get stuck* and be unable to exercise new paths through the code of the target program. This can happen for a number of reasons, but is most frequently caused by the inability of the fuzzer's random mutations to satisfy complex conditions, introduced by checks in the program, upon input data.

5.3.6 Automated Vulnerability Discovery - Drilling

Driller proposed a mitigation for the stalling of the fuzzer due to the inability to satisfy complex solutions. It uses concolic execution to trace the paths that the fuzzer finds, identifies conditional checks that the fuzzer fails to satisfy, and synthesizes inputs to satisfy these conditions. Driller triggers its operation when the fuzzer gets "stuck", and is unable to find further testcases (it detects this by checking AFL's progress evaluation heuristics). Once this stall condition is detected, Driller symbolically traces and attempts to mutate all test cases that AFL has found into test cases that reach parts of code not previously seen. These resulting test cases are then synchronized back into the fuzzer, so that it can explore these newly-reached areas of code.

By pairing fuzzing with concolic execution, Driller achieves better results than the naive union of the individual underlying techniques. However, Driller's automated approach to symbolic input synthesis has some drawbacks.

Driller's synthesis works by *diverting* a path and forcing it to satisfy a check that it would have otherwise avoided. There are several limitations, inherent in Driller, that hamper its effectiveness in certain situations. These include, but are not limited to:

SMT solver. Driller uses an SMT solver to solve negated *path predicates* (constraints on the input values to the program that must be satisfied in order to trigger the path in question) to synthesize inputs that diverge from the original execution. However, depending on the complexity of the path predicates involved, the SMT solving process may not terminate. While this represents a significant challenge for Driller, the complexity of these predicates might not translate to the complexity of interaction with the software. If this is the case, a human assistant might be able to controllably divert the path taken through the program, even when the constraint solver cannot.

Inflexible path predicates. Depending on implementation details in the program, earlier path predicates might prevent the deviation of later path predicates. Such predicates are frequently created by certain input transformation procedures. For example, string-to-int translation (such as the `atoi` function) takes different conditional branches, based on the values in the input string, while converting an input string to an integer. These conditional branches create path predicates. Later, the program might perform some action based on the value of this integer. When Driller attempts to divert this decision to take a different action, the earlier path predicates on the input string prevent this diversion.

Humans, of course, do not share this inflexible way of reasoning about path predicates.

Semantic transitions versus control flow transitions. Driller cannot understand the program semantically, and simply attempts to deviate the control flow of the program. A human, on the other hand, can identify much more intricate *semantic* deviations (for example, winning, as opposed to losing, a game), allowing for the triggering of whole new areas of code to deal with these new semantic settings.

These limitations conspire to erode Driller's ability to produce deviating inputs in many cases. In the next section, we will discuss how these limitations can be worked around with human assistance.

5.4 Human Assistance

As we discuss in the previous section, automated input synthesis techniques suffer from limitations that cause them to eventually get *stuck* in the exploration of a program. Even Driller, which leverages the power of symbolic execution to divert testcases, is only a partial solution. This is because, while Driller can make major changes to the input testcase it analyzes, it can only (by design and fundamental limitation) achieve only minor deviations.

On the other hand, a human can leverage intuition and a semantic understanding of the target program to achieve very large deviations, potentially allowing further analyses to continue to make progress. In this chapter, we explore the integration of human assistance into a Cyber Reasoning System as Innovation agents, keeping the Organizational and Selectional agents fully automated. We focus on the vulnerability discovery stage of the analysis and explore ways to integrate human effort to improve analysis efficiency.

Human assistance takes place over an interface (the *Hardware Abstraction Link*, or HAL) which will be described later this section. To maximize the effectiveness of this effort, HaCRS carries out a number of analyses that enhance the data it is able to expose to the humans. In this section, we describe how human assistants are selected, the interface over which HaCRS and humans communicate, and how the resulting data is used to enhance the vulnerability detection ability of HaCRS.

5.4.1 Assistant Expertise

The style of human assistance differs according to the assistant's expertise level. For example, while HaCRS could reasonably ask an expert human to analyze a control flow graph and identify potential paths through it, a non-expert would be flabbergasted by such a request. The information presented, and the interfaces which are used, must be adapted to the chosen assistant's level of expertise.

Since expert humans (i.e., binary analysts) are rare and expensive, the integration of assistance from non-expert humans (i.e., an average internet citizen) is of particular interest. While they do not scale to the extent of automated processes, non-expert humans scale considerably easier than experts, due to their higher availability. When more knowledge is required, semi-experts (i.e., undergraduates in Computer Science) can be leveraged more readily than experts. Thus, in this chapter, we focus mainly on techniques to integrate non-expert assistance, with a detour into semi-expert assistants for completion.

Over the decades that humans have been interacting with software, the skill of performing such interaction has become gradually instilled in the human population. As such, even non-experts are well-trained to understand and drive computer software. Thus,

Concept	Computer	Expert	Non-Expert
Symbolic Equations	✓		
Control-Flow Graph	✓	✓	
Execution Path	✓	✓	
I/O (Text)	✓	✓	✓
Semantic Meaning		✓	✓

Table 5.1: Program analysis concepts, as they are easily understood by automated techniques, expert humans, and non-expert humans. To be understandable to non–experts, the Human-Automation Link must avoid complex program analysis topics.

we can tailor HAL to non-experts by sticking to concepts that they can grasp and avoiding complex program analysis concepts, as shown in Table 5.1. For example, rather than ”triggering transitions”, we used the term “triggering functionality”, which requires less technical knowledge to understand. Additionally, we expose non-experts only to the input and output log associated with prior interactions with the programs that the HaCRS is trying to analyze, and avoid any use of program analysis terms in task descriptions.

5.4.2 Human-assisted Input Generation

HaCRS uses human assistance to break through the “semantic barriers” that limit the effectiveness of automated analyses described in Sections 5.3.5 and 5.3.6. It gives its human assistants a goal: generate an input testcase that executes some amount of code in the target program that has not been reached by previously-known testcases (i.e., those previously found by automated analyses or other humans).

Human assistants interact with the target program to generate testcases, and these testcases are synchronized throughout HaCRS’ components.

Human-to-automation. Human-produced testcases are synchronized to the automated program exploration components, which proceed to mutate them in an attempt to trigger new functionality.

Human-to-human. Humans can view and modify the testcases produced by other human assistants. This enables a collective effort of the understanding and leveraging of program semantics toward a higher code coverage. HaCRS

Automation-to-human. The resulting automation-mutated testcases can then be shown to the human assistants (we term such a testcase an "example testcase"), who can review them, understand possible further improvements and changes that can be made, and relay those changes back to the automation by producing human-modified testcases.

Testcase conversion. The synchronization of testcases from automated components to a human assistant poses a challenge: automated systems, driven by either random input generation or input synthesis via constraint solving, have no guarantee to produce printable characters when the target program does not require it. Non-printable testcases look like gibberish when shown to a human, which hinders the human's ability to reason semantically about what actions the testcase is causing the target program to take.

To address this issue, we use the existing `afl-tmin` utility shipped with AFL [15]. This utility is a *testcase minimizer*. It takes an input testcase and uses lightweight dynamic techniques to a) remove unnecessary input characters and b) convert as many characters as possible to be printable, without changing the code coverage achieved by the input. In practice, it achieves very good results on programs with a text interface.

5.4.3 Automation-assisted Human Assistance

Simply presenting previously-discovered testcases to human assistants enables an improvement over a base-case Cyber Reasoning System (we show this in Section 5.5). However, since the communication between HaCRS and humans takes place over a well-defined interface, HaCRS can provide extra information and capabilities to enhance the humans' abilities to complete the assistance task.

Interaction assistance. One such capability provided by HaCRS is the automated re-formatting of input data. HaCRS traces each program testcase to detect if input data must be provided in a specific format. It achieves this by leveraging existing techniques in protocol recovery [121, 122, 123]. Depending on configuration (and expertise of human assistants), this information can either be presented to the human assistants or utilized automatically to mutate human-created inputs into a format understood by the application.

In our prototype, we mainly utilize these techniques to automatically recover non-standard field delimiters used by the binaries in our dataset, but they can also be used to support information packing protocols, such as ASN1.

High-level guidance. Having enabled human interaction for binaries with complex input data specifications, HaCRS turns to the question of maximizing the ability of its humans to understand how to interact with the target program. It does this by identifying and categorizing constant string references in the binary.

HaCRS identifies static string references by analyzing its CFG, and performs a static data flow analysis to categorizes these into strings produced as output by the program and strings compared against user input into the program. HaCRS identifies output strings by detecting when they are passed into common output functions (such as `puts` and `printf`). Input strings are considered to be anything that is passed to a string comparison function. In the case of statically-linked binaries, HaCRS can leverage the function identification functionality built into the Mechanical Phish, which detects common functions in programs using pre-defined dynamic test-cases [106]).

HaCRS provides a list of potential output strings in the target program to help its human assistants, relaying which of these strings have not yet been triggered (i.e., caused to be output by the program) by other testcases. These can provide useful semantic information regarding the untapped functionality of the target program.

While HaCRS focuses on text-based software, it is important to keep in mind that analogous information can be recovered for software with a graphical user interface. For example, a similar analysis can identify GUI widgets, render them, and display them as potential directions of exploration for human assistants.

Symbolic tokens. First, HaCRS creates suggestions for human assistants for ways that testcases might be modified to divert program flow. This is done through a process of *symbolic tokenization*. HaCRS symbolically traces the target program in the context of each testcase to recover constraints placed on the input by the target program. It analyzes these constraints to identify contiguous bytes on which the constraints are *similar* (in terms of the number of constraint expressions and the types of arithmetic and boolean operations the constraint expressions are composed of). These contiguous bytes represent tokens processed and reasoned about by the binary.

HaCRS then identifies alternate values for each symbolic token. It rewinds its symbolic trace to the address at which the first constraint of the token was introduced, and performs a symbolic from that point to retrieve other potential values. The symbolic exploration runs until a timeout (we found 30 seconds to be a reasonable timeout in our experiments). At the end of the timeout, the constraints of the various resulting paths are solved to alternate values for the token. These values are further refined by matching them against the input strings retrieved previously, and HaCRS produces two different sets of suggestions to its assistants: "educated guesses", which are the input strings that are prefix-matched by the recovered alternatives and "brute-force guesses", which are the raw alternatives themselves.

Note that, while the concept of generating alternatives for input is shared with Driller, the goal is different. Driller generates alternative testcases to drive execution down different paths. However, the alternatives generated by this method are meant to be learned by humans, understood, and reasoned about to produce new inputs through human intuition and previously-learned experience.

Input type annotation. Programs process different inputs differently, and HaCRS exposes this to its assistants by highlighting input bytes that are constrained by similar constraints (as with the symbolic token analysis, we use constraint count and operation types to compute constraint similarity). Input bytes highlighted with similar colors in the input testcases will be bytes that have been treated similarly to each other by the program, and may represent similar type of data. Most importantly, this differentiates string input (such as a command) against numeric input (which is passed to functions such as `atoi`, which impose specific constraints on the data).

5.4.4 Human-Automation Link

The interface between the HaCRS and its human assistants must be designed in such a way as to be understandable by both parties. To do this, we created a Human-Automation Link (HAL) that exposes, to the humans, only the concepts of program analysis that non-experts might be familiar with. For the curious reader, we reproduce a mock-up of the HAL interface in Figure 5.3.

The HAL interface in Figure 5.3 consists of the following elements:

Program description. When a description of the target program is available, it can aid assistants in interacting with it. In the case of Cyber Grand Challenge binaries, this description includes a very brief (usually four to five sentences) summary of the program's purpose, as written by the program authors. In a real-world setting, human assistants can be provided with the technical manual for the piece of software being tested.

Tasklet instructions. The HaCRS provides human-readable instructions, which are presented to the assistant alongside each tasklet.

Example interactions. The HaCRS provides logs of previous interactions with the software, in the form of input and output data. For the text-based software of DECREE OS, to help assistants understand what data was originated from them (program input) and what came from the program (program output), the input and output are displayed in different colors. A version of HaCRS for software with a graphical user interface could instead have a video record of the interaction, but this is not supported by our prototype.

CRS-Generated Suggestions. To help assistants understand how to deviate from a test case, they can invoke the deviation annotation interface. This interface displays data recovered through the automated analyses described in Section 5.4.3 to present the assistant with a better idea of how to make a program behave differently than in the example testcase.

Interaction terminal. To facilitate the interaction between human assistants and the target program, a terminal is presented to interact with the software. Again, to help assistants understand differentiate user input from program output, the input and output are displayed in different colors.

Tasklet goal and feedback. Any human-facing task must have an understandable end goal to avoid confusion on the part of the assistants. HaCRS requires its human assistants to trigger previously-unseen functionality in the target programs. To this end, it provides feedback to the assistant regarding the amount of previously-unseen control flow transitions that the assistant was able to trigger.

Along with this, it provides a display of *untriggered output strings*, as described in Section 5.4.3. With their human ability to reason about semantic information, assistants can leverage the bounty strings to better target untriggered functionality in the program.

Each tasklet also has a timeout and an abort button: if the assistant is unable to complete the tasklet before a timeout, or presses the abort button, the tasklet is terminated. This acts as a guard against the situation when the tasklet is not actually completable (for example, if the remaining untriggered functionality is dead code).

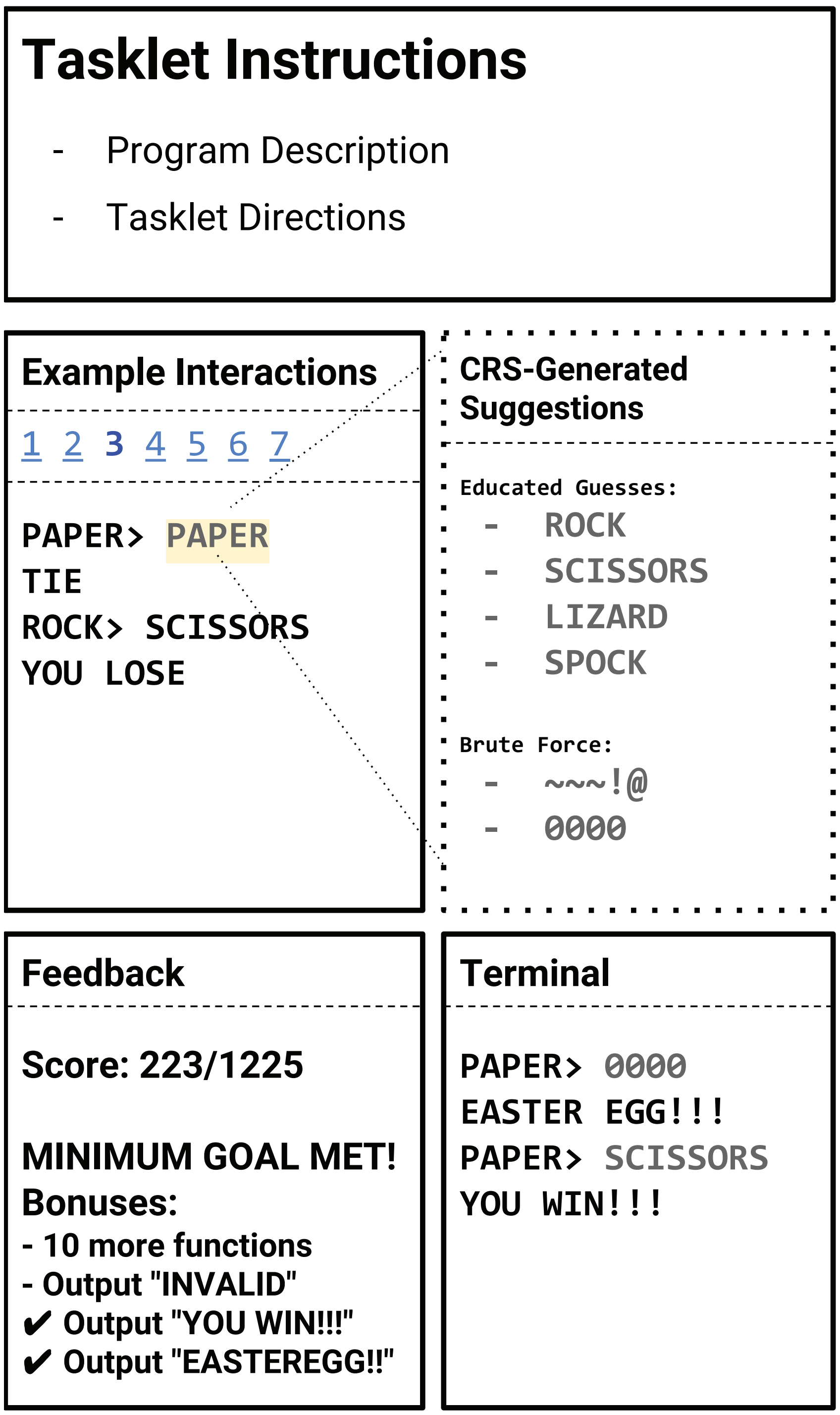
In the next section, we will explore the implication of human assistance by evaluating the performance of HaCRS against the performance of the unaided Mechanical Phish.

5.5 Evaluation

In this section, we evaluate the impact produced by our integration of non-expert human effort into the Cyber Reasoning System paradigm. We measure the result of this as a whole, in terms of the overall number of vulnerabilities identified in our dataset, but also explore certain low-level details of the system.

5.5.1 Dataset

As previously mentioned, Mechanical Phish was designed to operate on binaries for DECREE, the operating system designed for the DARPA Cyber Grand Challenge. A



```
PAPER> PAPER
TIE
ROCK> SCISSORS
YOU LOSE
```

```
Educated Guesses:
 -  ROCK
 -  SCISSORS
 -  LIZARD
 -  SPOCK

Brute Force:
 -  ~~~!@
 -  0000
```

Score: 223/1225

MINIMUM GOAL MET!
Bonuses:
- 10 more functions
- Output "INVALID"
✔ Output "YOU WIN!!!"
✔ Output "EASTEREGG!!"

```
PAPER> 0000
EASTER EGG!!!
PAPER> SCISSORS
YOU WIN!!!
```

Figure 5.3: A diagram of the HaCRS user interface, showing different components for seeding tasklets (all solid-bordered components), drilling tasklets (dotted-bordered components), and seeking tasklets (d ashed-bordered components).

total of 250 binaries were produced by DARPA for the Cyber Grand Challenge[2]. These binaries vary in complexity, but are designed to mimic a wide range of vulnerabilities and behaviors found in real-world software. Each Cyber Grand Challenge binary is guaranteed to have at least one vulnerability, and proof-of-concept exploits, along with high-quality testcases, are provided for each. This makes it possible to measure, with some degree of certainty (after all, previously-unknown vulnerabilities might also be present), the effectiveness of vulnerability detection techniques. As such, they have already been used in the evaluation of various other scientific work [124, 36, 103].

Our dataset is the subset of DECREE programs that present a human-usable text protocol or for which the interaction assistance provided by HaCRS (as discussed in Section 5.4.3) was able to facilitate a human-usable text protocol. We selected these by automatically detecting the presence of non-printable characters in the author-provided testcases (we did not otherwise use these testcases in the experiments). We filtered binaries in this way because, to our human assistants, such protocols are understandable, and, therefore, they allow for manual interaction. Among the CGC binaries, a total of 85 binaries meet this criteria.

While this requirement to filter the dataset to binaries designed for human interaction is limiting, certain approaches do exist to alleviate it. For example, IARPA funded a multi-year effort, dubbed STONESOUP [125] that developed a number of approaches to *gamify* software. Such approaches can be used to expand the amount of binaries with which humans can assist, but they generally fail to recreate the valuable semantic hints in software designed for humans. We leave the integration of such program mutation into our interaction assistance component as future work.

Even though a protocol might be text only, it might still be hard for humans to

[2]DARPA recently funded the creation of a human-readable repository with information on these applications, hosted at `http://www.lungetech.com/cgc-corpus`.

understand. As an example of this, consider PDF, which is a text-only file format that is designed to be parsed exclusively by computer programs. To better understand the implications of human assistance on the binaries in our dataset, we manually categorized them according to the following qualities:

Technical expertise. We determined whether a program requires technical expertise to be used. For example, some of the programs in the dataset are language interpreters or databases, requiring users to be familiar with such Computer Science concepts as programming languages. These programs would be rated as requiring high technical expertise.

Semantic complexity. We attempted to identify whether actions taken by the program yield themselves to high-level reasoning about the program's *intent*. For example, a move taken in a chess match would have high semantic complexity, whereas an iteration of a compression algorithm would not. Thus, a chess engine would be ranked as having high semantic complexity, whereas a compression utility would not.

CGC binaries are fairly small, and the small size of these binaries makes them well-suited for such classification. Specifically, because the binaries tend to be "single-purpose" (i.e., a recipe storage application, as opposed to a web browser), most binaries do not have different modules with different semantic complexity or technical expertise requirements.

The binaries, by their various classifications, are presented in Table 5.2. We expect human assistants to do best on binaries with a high semantic complexity, and unskilled humans to do best with binaries requiring a low technical expertise.

Semantic Complexity	Technical Expertise	Binaries
High	Low	CADET_00001 CADET_00003 CROMU_00001 CROMU_00003 CROMU_00005 CROMU_00017 CROMU_00029 CROMU_00031 CROMU_00037 CROMU_00040 CROMU_00041 CROMU_00044 CROMU_00046 CROMU_00054 CROMU_00065 CROMU_00076 CROMU_00087 EAGLE_00005 KPRCA_00011 KPRCA_00017 KPRCA_00018 KPRCA_00022 KPRCA_00023 KPRCA_00026 KPRCA_00030 KPRCA_00042 KPRCA_00043 KPRCA_00049 KPRCA_00051 KPRCA_00052 KPRCA_00053 KPRCA_00055 KPRCA_00071 KPRCA_00079 NRFIN_00004 NRFIN_00005+ NRFIN_00065 TNETS_00002 YAN01_00001
High	High	CROMU_00002 CROMU_00008 CROMU_00009 CROMU_00010 CROMU_00011 CROMU_00014 CROMU_00015 CROMU_00021 CROMU_00022 CROMU_00023 CROMU_00035 CROMU_00042 CROMU_00048 CROMU_00051 CROMU_00071 CROMU_00083 CROMU_00096 CROMU_00098 KPRCA_00007 KPRCA_00013 KPRCA_00021 KPRCA_00028 KPRCA_00031 KPRCA_00036 KPRCA_00041 KPRCA_00045 KPRCA_00054 KPRCA_00068 LUNGE_00002 NRFIN_00001+ NRFIN_00009+ NRFIN_00054 NRFIN_00055 YAN01_00002 YAN01_00007 YAN01_00011
Low	Low	NRFIN_00008* NRFIN_00064 NRFIN_00069+ YAN01_00015
Low	High	CROMU_00025* CROMU_00030* CROMU_00034* KPRCA_00010* KPRCA_00064* NRFIN_00071

Table 5.2: The binaries in our dataset, grouped by semantic complexity of their operation and the required technical (Computer Science) expertise. These binaries were filtered for receiving mostly printable input, but some of them (marked with *) decoded that into raw binary input, making them suboptimal for human interaction. Others (marked with +) received their inputs in protocols which were automatically translated by HaCRS' interaction assistance layer to be easily human-interactable. We expect humans to do best on binaries with a high semantic complexity, and unskilled humans to do best with binaries requiring a low technical expertise.

5.5.2 Human Assistants

HaCRS was designed to support different levels of assistant expertise, from non-experts to experts. We evaluated the impact of both non-expert and semi-expert assistants.

Non-experts. For the non-experts, We used Amazon's Mechanical Turk service to dispatch tasklets to humans with no required Computer Science knowledge [113]. This provided HaCRS with an API to interact with human intelligence in a scalable way, allowing it to submit tasklets, as Mechanical Turk Human Intelligence Tasks (HITs), without concerning itself with human availability.

Because we had finite funds for our experiments, we implemented a *human interaction cache*. When the HaCRS would create tasklets for non-expert human assistance, we would first check the interaction cache to determine if this human assistance task had already been requested in by a prior experiment. If it had, and if at least one of the cached human testcases "solved" the tasklet (in the sense of triggering new code), the HaCRS would reuse it instead of paying for a HIT. We used the human interaction cache whenever we were running experiments on identical configurations of the Hardware-Automation Link. This allowed us to re-run some of the experiments throughout the design and development of the system and remain within our budget.

In the end, between the different experiments to fully understand our system, we spent about $2000 on Mechanical Turk HITs, resulting in 21268 unique testcases across our experiment. While this is a large amount for a research lab, it would be trivial spending for a nation state or large corporation looking to scale out their analyses.

Semi-experts. We recruited five undergraduate Computer Science students, familiar with programming topics but not with program analysis, to act as our semi-expert human assistants. These undergraduates interacted with a random sampling of 23 binaries from our dataset, generating a total of 115 testcases.

Ethics. As our experiments involve human assistants, we were careful to ensure that ethical procedures were followed. We worked with the Institutional Review Board of our institution to evaluate our testing protocol. The IRB approved our experiments, and we were careful to follow ethical guidelines throughout our work.

5.5.3 Human-Automation Link

As we proposed a number of optimizations to the Human-Automation Link in Section 5.4.3, it is important to understand whether this actually enhances the effectiveness of human assistances. To determine this, we performed two separate experiments in having non-experts interact with programs in the HAL, with our optimizations in Section 5.4.3 disabled in the first and enabled in the second.

For each binary, we dispatched tasklets to the human assistants until they were unable to make further progress in code coverage, given an hour-long timeout. We collated the results by the semantic complexity of the binaries involved, and computed the median number of testcases at which progress stopped being made.

Our improvements to the HAL allowed our assistants to contribute a significantly higher amount of testcases than they were previously able to. For semantically complex binaries, the number of testcases was roughly double, but for binaries that were not semantically complex, the improvement was considerably higher, approach a three-fold increase in the number of successful testcase generations. On further investigation, this makes sense – analyzing the testcases generated by the human assistants, we were able to see them quickly guess how to interact with semantically-complex programs, but struggle with less complex ones. However, with the improved HAL interface, they were given extra information that they could leverage to provide high-quality testcases.

5.5.4 Comparative Evaluation

HaCRS improves the vulnerability detection process by injecting human intuition into the Cyber Reasoning System. To understand how effective this is, we analyze the impact that non-expert and semi-expert assistance has on CRS effectiveness. To explore these questions, we ran several different experiment configurations:

Non-expert humans. As a baseline to understand the ability of humans to generate inputs for binary code, we disabled the automated components of the Mechanical Phish and relied solely on human assistants for testcase creation.

Semi-expert and non-expert humans. With the amount of semi-experts at our disposal, it did not make sense to have them work alone. As such, we ran an integrated semi- and non-expert experiment. To understand the impact of expertise, we added the semi-experts to our assistant pool and reran the human-only experiment. Testcases produced by non-experts are presented to semi-experts as examples, and testcases created by the semi-experts are synchronized into the system and eventually presented to the non-experts.

Unassisted fuzzing (AFL). This configuration, with both symbolic and human assistance disabled, achieves a baseline for comparing the other experiments to understand the relative gains in code coverage and crashes.

Symbolic-assisted fuzzing (Driller). This is the reference configuration of the Mechanical Phish: a fuzzer aided by a dynamic symbolic execution engine, as proposed by Driller. We consider this as the prior state-of-the-art configuration.

Configuration	Semantic Complexity	Expertise Required	Median Code Coverage	Median #AT	Median #HT	Binaries Crashed	Median Time-to-Crash
Non-expert Humans	High	Low	46.68%	0	137	0	N/A
	High	High	48.83%	0	150	0	N/A
	Low	Low	48.69%	0	168	0	N/A
	Low	High	16.81%	0	297	0	N/A
		Total	47.19%	0	151	0	N/A
All Humans	High	Low	46.83%	0	137	0	N/A
	High	High	48.83%	0	150	1	2815
	Low	Low	48.69%	0	168	0	N/A
	Low	High	17.39%	0	298	0	N/A
		Total	47.19%	0	151	1	2815
Unassisted Fuzzing	High	Low	41.82%	410	0	12	807
	High	High	43.32%	526	0	14	1278
	Low	Low	56.17%	187	0	1	143
	Low	High	17.46%	211	0	1	7
		Total	42.87%	361	0	28	897
Symbolic-assisted Fuzzing	High	Low	42.90%	663	0	14	1302
	High	High	48.85%	764	0	17	1426
	Low	Low	56.07%	156	0	2	62
	Low	High	41.88%	1500	0	3	390
		Total	44.91%	649	0	36	1298
Human-assisted Fuzzing	High	Low	49.70%	326	136	21	1378
	High	High	60.45%	472	126	24	1442
	Low	Low	64.03%	125	35	2	48
	Low	High	17.46%	207	9	1	10
		Total	52.38%	308	84	48	1334
Human-assisted Symbolic-assisted Fuzzing	High	Low	48.98%	369	69	23	1140
	High	High	59.68%	485	11	28	1855
	Low	Low	64.03%	121	46	2	47
	Low	High	48.52%	641	5	3	584
		Total	53.45%	403	34	56	1301

Table 5.3: The crashes found and code coverage achieved by different configurations of the automated and human components of HaCRS. The full HaCRS configuration includes human non-expert, human semi-expert, and automated innovation agents. #AT, and #HT are the numbers of automation-originated testcases and human-originated testcases, respectively, that were deemed "unique" by the Mechanical Phish's testcase evaluation criteria.

Human-assisted fuzzing. In this configuration, Driller is replaced with our Human-Automation Link. Rather than symbolically tracing fuzzer-generated testcases, we present them to our human assistants and synchronize their testcases back into the fuzzer. This configuration, together with the Driller and AFL configurations, allow us to understand the relative effectiveness of Drilling versus Human Assistance.

Human-assisted Symbolic-assisted fuzzing. This is the "complete" configuration of HaCRS, all components, representing the new state-of-the-art in Cyber Reasoning System.

The results of the experiment are presented in Table 5.3.

End-to-end system. The most obvious result is the improvement in the number of vulnerabilities that were identified with the full HaCRS configuration. By iteratively combining human assistance and symbolic assistance to its internal fuzzer, the HaCRS was able to identify an additional *twenty* bugs in different binaries over symbolically-assisted fuzzing (a whopping 55% improvement) and *twice* as much as the base-case fuzzer alone. This result is significant: non-expert humans, overwhelmingly likely to have no security or program analysis training, are able to make real contributions toward the analysis of binary software.

Comparison to Driller. In HaCRS, human assistants take on a very similar role to Driller: they provide extra inputs that the fuzzer can leverage to avoid stalling in its exploration of the target program. Rather than making small control-flow diversions, human assistants make *semantic* divergences based on their understanding of the operation of the target program. This is reflected in the results – for semantically-complex programs, the human assistants significantly beat out Driller, achieving an improvement of up to 11.6% improvement in coverage. However, for binaries that did not have semantic complexity but required computing expertise, the human assistants suffered, being unable to understand the concepts presented by the program and intuit how to interact with it. This is where the combination of human and automated analysis shines – Driller picks up the slack in these binaries, and the combination of human and symbolic assistance achieves higher code coverage than either alone.

Impact of expertise. Interestingly, the inclusion of semi-experts in our analysis did not seriously impact the achieved code coverage. This is an example of the different scale achievable for experts and semi-experts. While we were able to get just over 300 Mechanical Turk workers to assist HaCRS, we were only able to recruit five undergraduate students, and they could not make a strong impact on the results (in fact, because the results are presented in aggregate, there was almost no impact on the median measurements). However, they did have localized success: due to their ability to intelligently interact with more complex binaries, the experts were able to identify a bug in one of the applications without any human assistance. Specifically, they triggered a bug in *CROMU_00021*, which implements a simple variable storage and arithmetic engine, but contains an exploitable bug when a variable with a blank name is created.

5.5.5 Case Studies

In the course of our experiments, our human assistants achieved some results that are interesting to explore more in-depth. This was despite the fact that the human assistants were completely unskilled in program analysis, and were recruited with absolutely no training. Here, we delve deeper into these bugs, and discuss why human effort helped with these specific binaries.

Coverage case study: CROMU_00008. This binary implements a database with a SQL-inspired interaction interface. Proper use of this binary required understanding the concepts of storing and retrieving data records. Interestingly, our human assistants quickly developed an understanding for how to do this, taking the suggested keywords from the CRS suggestions and combining them into expressions the program understood. They achieved a code coverage of 55.5%, compared with 12.1% for the automated analyses. Manual investigation into the delta between automation and human assistance revealed that, as expected, the humans produced inputs that were meaningful for the program, while the symbolic seed synthesis attempted to optimize for code coverage, triggering many meaningless states (such as incorrect commands) without ever getting to the actual operation of the program.

Coverage case study: KPRCA_00052. This binary is surprisingly complicated: the assistant is presented with a pizza order menu system. To properly navigate this system, the assistant must understand how a pizza is made: the crust is chosen first, then the cheese, then the toppings. This makes it very hard for the automated system to explore this binary and, in fact, our automation achieved a 19% code coverage over the course of the experiment, as opposed to 52% achieved by human assistants.

Vulnerability-detection case study: KPRCA_00043. This binary includes a lyrical storage engine that disallows certain words from being provided as lyrics. Furthermore, these words are checked by a *bloom filter*. Because this filter is implemented as a hash map, the resulting symbolic memory references make it difficult for symbolic execution to produce these words. The vulnerability consists of an overflow in the lyrics buffer if enough words are entered that trigger the bloom filter but then pass the secondary check against the blacklist. For example, enough misspellings of the blacklisted words can overflow the filter.

Interestingly, the binary includes a semantic hint – a depressing message is printed when the program starts. We observed that our assistants quickly picked up on this hint and produced inputs containing blacklisted words, whereas the symbolic seed synthesis produced gibberish and failed to trigger the blacklist. While both approaches actually had a similar level of code coverage, we hypothesized that the human-assisted inputs would be a better seed set for a fuzzer to find the vulnerability. We verified this by running AFL for an hour with the human-provided seeds, and for an hour with the automation-created seeds. As expected, the former triggered the vulnerability, while the latter did not.

Vulnerability-detection case study: NRFIN_00055. This binary implements a complex mathematical calculator. In this case, the assistants were given no guidance – simply an input prompt. Over time, they figured out that mathematical expressions led to the output of mathematical solutions, and their apparent knowledge of mathematics came into play. Assistants submitted a number of equations, exploring different operations in semantically-valid inputs (i.e., proper use of parenthesis and so forth). By comparison, the symbolic seed synthesis became overwhelmed by the number of paths in the parsing engine, resulting in an inability to produce reasonable inputs.

Again, we suspected that the seeds provided by humans would be more useful to a fuzzer, and tested this by an hour-long execution of AFL. As with the previous case, AFL was able to mutate the human-produced testcases into a crashing input by introducing a modifier symbol missed by the assistants into one of their testcases, resulting in a type confusion. The automatically-generated testcases were less useful, and no crash was found from them within an hour.

5.6 Conclusion

The use of principled human-assistance in Cyber Reasoning Systems constitutes a paradigm shift in our view of how binary analysis is done. Instead of the dichotomy between human-led, semi-automated systems (HCH, as discussed in Section 5.1) and fully automated systems (CCC), we propose a C(H|C)C system, where computers, which scale beyond human ability, make organizational calls and humans, whose intuition has not yet been replicated, assist when able. This system can utilize the insight of *non-expert humans*, who are more abundant than expert humans and thus scale better. In the absense of these humans, these systems are able to operate fully autonomously, just at a lower effectiveness.

In this report, we have taken a first look at how non-experts impact the automated vulnerability discovery pipeline. The results are significant: humans, with no security training, were able to seriously improve the bug detection rate of a state-of-the-art vulnerability analysis engine. Further exploration is warranted. For example, humans can confirm or repudiate results of static analysis, combine behavior observed in different testcases into one, and help verify automatically-generated patches. All of this is challenging or simply infeasible with modern techniques, but the use of human assistance can greatly augment Cyber Reasoning Systems with these capabilities regardless.

Chapter 6

What Now?

Through my graduate studies, I have led the creation of a base upon which cyber-autonomy can be built. More importantly, I had contributed this base as an open source platform for the community. Going forward, it is my hope that this base will spur research in the area, and that a PhD student working on their dissertation thirty years from now will look back and be shocked at how far the field has come.

For now, as I discussed in Chapter 4, the problem is not solved. While we have achieved a lot of capability in the autonomous analysis of binary software, we have not made much progress in the autonomous *understanding* of it. This is almost more of a philosophical problem than a technical one, but we can speculate on technical implications and techniques that might address it.

For example, it is interesting that the large increase in the number of crashes found in CGC binaries by HaCRS (a 55% improvement compared to a CRS with no human assistance) does not have an associated increase in code coverage (which is only improved by 8%). It is possible to interpret this in several ways, but I suspect, based on initial examination of possible causes of this phenomenon, that this has to do with the fact that humans optimize the exploration of the *state space* of the program, while autonomous techniques prioritize the increase of code coverage. Thus far, there is no clear path to making automated techniques understand program state on an abstract level (i.e., that a given bit X signifies that the program has entered phase Y).

As HaCRS demonstrates, we can continue to use the crutch of human assistance until automated techniques can address this problem. In the meantime, a better semantic understanding of programs would make a great research direction.

If someone reads this dissertation and is inspired to solve the problem, make sure to send a pull request!

Bibliography

[1] Зарецкий, А и Труханов, А и Зарецкая, Л, Энциклопедия профессора Фортрана. Просвещение, 1991.

[2] A. C. Lovelace, *Translator's notes to an article on babbage's analytical engine*, *Scientific Memoirs* **3** (1842) 691–731.

[3] A. Turing, *Checking a large routine*, in *The early British computer conferences*, pp. 70–72, MIT Press, 1989.

[4] G. Weinberg, "Fuzz testing and fuzz history." `http://secretsofconsulting.blogspot.com/2017/02/fuzz-testing-and-fuzz-history.html`.

[5] R. S. Boyer, B. Elspas, and K. N. Levitt, *SELECT-a formal system for testing and debugging programs by symbolic execution*, *ACM SigPlan Notices* **10** (1975), no. 6 234–245.

[6] P. Cousot and R. Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 238–252, ACM, 1977.

[7] J. W. Duran and S. Ntafos, *A report on random testing*, in *Proceedings of the 5th international conference on Software engineering*, pp. 179–183, IEEE Press, 1981.

[8] Indefinite Studies, "The Halting Problem for Reverse Engineers." `http://indefinitestudies.org/2010/12/19/the-halting-problem-for-reverse-engineers/`.

[9] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, *Modeling and discovering vulnerabilities with code property graphs*, in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, pp. 590–604, 2014.

[10] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, *Automatic Inference of Search Patterns for Taint-style Vulnerabilities*, in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, vol. 2015-July, pp. 797–812, 2015.

[11] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, *Cross-architecture bug search in binary executables*, in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, vol. 2015-July, pp. 709–724, 2015.

[12] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum, *WYSINWYX: What You See Is Not What You eXecute*, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* **4171 LNCS** (2008) 202–213.

[13] B. P. Miller, L. Fredriksen, and B. So, *An empirical study of the reliability of UNIX utilities*, *Communications of the ACM* **33** (1990), no. 12 32–44.

[14] C. Miller, "Fuzzing With Code Coverage By Example." `https://fuzzinginfo.files.wordpress.com/2012/05/cmiller_toorcon2007.pdf`.

[15] "American Fuzzy Lop." `http://lcamtuf.coredump.cx/afl/`.

[16] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, *A Taint Based Approach for Smart Fuzzing* , in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pp. 818–825, IEEE, 2012.

[17] M. Zalwski, "Bunny the Fuzzer Documentation." `http://code.google.com/p/bunny-the-fuzzer/wiki/BunnyDoc`.

[18] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, *Dowser: a guided fuzzer to find buffer overflow vulnerabilities*, in *Proceedings of the 22nd USENIX Security Symposium*, pp. 49–64, 2013.

[19] M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos, *The BORG: Nanoprobing Binaries for Buffer Overreads*, in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pp. 87–97, ACM, 2015.

[20] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, *Prospex: Protocol Specification Extraction*, in *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, pp. 110–125, IEEE, 2009.

[21] E. J. Schwartz, T. Avgerinos, and D. Brumley, *All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)*, in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, (Washington, DC, USA), pp. 317–331, IEEE Computer Society, 2010.

[22] C. Cadar, D. Dunbar, and D. Engler, *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*, in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, vol. 8, pp. 209–224, 2008.

[23] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, *EXE: Automatically Generating Inputs of Death*, in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pp. 322–335, 2006.

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

[25] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea, *Selective Symbolic Execution*, in *Proceedings of the 5th Workshop on Hot Topics in System Dependability*, 2009.

[26] P. Boonstoppel, C. Cadar, and D. Engler, *RWset: Attacking Path Explosion in Constraint-Based Test Generation*, in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 4963 LNCS, pp. 351–366. Springer Berlin Heidelberg, 2008.

[27] Y. Li, Z. Su, L. Wang, and X. Li, *Steering Symbolic Execution to Less Traveled Paths*, in *Proceedings of the 2013 ACM SIGPLAN international conference on Object Oriented Programming Systems Languages & Applications*, pp. 19–32, 2013.

[28] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, *Enhancing Symbolic Execution with Veritesting*, pp. 1083–1094, 2014.

[29] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, *Efficient State Merging in Symbolic Execution*, in *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*, p. 193, 2012.

[30] P. Saxena, P. Poosankam, S. McCamant, and D. Song, *Loop-Extended Symbolic Execution on Binary Programs*, in *Proceedings of the 18th International Symposium on Software Testing and Analysis*, p. 225, 2009.

[31] P. Godefroid, M. Y. Levin, and D. Molnar, *SAGE: Whitebox Fuzzing for Security Testing*, *ACM Queue* **10** (2012), no. 1 20.

[32] P. Godefroid, N. Klarlund, and K. Sen, *DART: Directed Automated Random Testing*, in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, vol. 40, pp. 213–223, 2005.

[33] G. Campana, *Fuzzgrind: un outil de fuzzing automatique*, in *Actes du 7ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, pp. 213–229, 2009.

[34] D. Caselden, A. Bazhanyuk, M. Payer, L. Szekeres, S. McCamant, and D. Song, *Transformation-aware Exploit Generation using a HI-CFG*, tech. rep., DTIC Document, 2013.

[35] S. K. Cha, M. Woo, and D. Brumley, *Program-Adaptive Mutational Fuzzing*, in *Proceedings of IEEE Symposium on Security and Privacy*, vol. 2015-July, pp. 725–741, 2015.

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

[37] D. Engler and D. Dunbar, *Under-constrained Execution: Making Automatic Code Destruction Easy and Scalable*, in *Proceedings of the 2007 international symposium on Software testing and analysis*, pp. 1–4, 2007.

[38] D. a. Ramos and D. Engler, *Under-Constrained Symbolic Execution: Correctness Checking for Real Code*, in *Proceedings of the 24th USENIX Security Symposium*, pp. 49–64, 2015.

[39] DAPRA, "DARPA Cyber Grand Challenge." `http://www.cybergrandchallenge.com/`.

[40] DARPA, "CyberGrandChallenge samples git repository." `https://github.com/CyberGrandChallenge/samples`.

[41] "OWASP Top 10 Project." `http://http://www.owasp.org`.

[42] Bloomberg Business, "Hospital Gear Could Save Your Life or Hack Your Identity." `http://www.bloomberg.com/features/2015-hospital-hack/`.

[43] K. Thompson, *Reflections on Trusting Trust*, *Communications of the ACM* **27** (Aug., 1984) 761–763.

[44] "The XcodeGhost malware." `http://www.apple.com/cn/xcodeghost/#english`.

[45] L. Szekeres, M. Payer, T. Wei, and D. Song, *SoK: Eternal War in Memory*, in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 48–62, 2013.

[46] F. Pérez and B. E. Granger, *IPython: A System for Interactive Scientific Computing*, *Computing in Science and Engineering* **9** (May, 2007) 21–29. http://ipython.org.

[47] T. Avgerinos, S. K. Cha, B. L. Tze Hao, and D. Brumley, *AEG: Automatic Exploit Generation*, in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, 2011.

[48] E. Schwartz, T. Avgerinos, and D. Brumley, *Q: Exploit hardening made easy*, in *Proceedings of the 20th USENIX Security Symposium*, vol. 8, p. 25, 2011.

[49] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, *Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware*, in *Proceedings of Network and Distributed System Security Symposium*, no. February, pp. 8–11, Internet Society, 2015.

[50] L. De Moura and N. Bjørner, *Z3: An Efficient SMT Solver*, in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, (Berlin, Heidelberg), pp. 337–340, Springer-Verlag, 2008.

[51] C. Barrett, L. De Moura, and A. Stump, *SMT-COMP: Satisfiability modulo theories competition*, in *Computer Aided Verification*, pp. 20–23, Springer, 2005.

[52] T. Dullien and S. Porst, *REIL: A platform-independent intermediate representation of disassembled code for static code analysis*, *CanSecWest* (2009).

[53] C. Lattner and V. Adve, *LLVM: A compilation framework for lifelong program analysis & transformation*, in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pp. 75–86, IEEE, 2004.

[54] N. Nethercote and J. Seward, *Valgrind: a framework for heavyweight dynamic binary instrumentation*, *ACM Sigplan Notices* **42** (2007), no. 6 89–100.

[55] Y. Shoshitaishvili, "PyVEX - Python bindings for VEX IR." `http://github.com/zardus/pyvex`.

[56] L. Xu, F. Sun, and Z. Su, *Constructing Precise Control Flow Graphs from Binaries*, *University of California, Davis, Tech. Rep* (2009).

[57] C. Cifuentes and M. Van Emmerik, *Recovery of Jump Table Case Statements from Binary Code*, in *Proceedings of the Seventh International Workshop on Program Comprehension*, pp. 192–199, IEEE, 1999.

[58] J. Troger and C. Cifuentes, *Analysis of Virtual Method Invocation for Binary Translation*, in *Proceedings of Ninth Working Conference on Reverse Engineering, 2002*, pp. 65–74, IEEE, 2002.

[59] B. Schwarz, S. Debray, and G. Andrews, *Disassembly of Executable Code Revisited*, in *Proceedings of Ninth working conference on Reverse engineering, 2002*, pp. 45–54, IEEE, 2002.

[60] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, *Static Disassembly of Obfuscated Binaries*, in *Proceedings of the 13th USENIX Security Symposium*, vol. 13, pp. 18–18, 2004.

[61] J. Kinder and H. Veith, *Jakstab: A Static Analysis Platform for Binaries*, in *Proceedings of the 20th international conference on Computer Aided Verification*, (Berlin), pp. 423–427, Springer-Verlag, 2008.

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

[63] M. Müller-Olm and H. Seidl, *Precise Interprocedural Analysis Through Linear Algebra*, in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, vol. 39, pp. 330–341, 2004.

[64] J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey, *Signedness-Agnostic Program Analysis: Precise Integer Bounds for Low-Level Code*, in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7705 LNCS, pp. 115–130, 2012.

[65] J. Lee, T. Avgerinos, and D. Brumley, *TIE: Principled Reverse Engineering of Types in Binary Programs*, in *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS'11)*, 2011.

[66] J. Newsome, D. Brumley, J. Franklin, and D. Song, *Replayer: Automatic Protocol Replay by Binary Analysis*, in *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 311–321, 2006.

[67] S.-K. Huang, M.-H. Huang, P.-Y. Huang, C.-W. Lai, H.-L. Lu, and W.-M. Leong, *CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations*, in *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*, pp. 78–87, IEEE, 2012.

[68] D. K. Sean Heelan, *Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities*, 9, 2009.

[69] H. Shacham, *The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)*, in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, vol. 22, pp. 552–561, 2007.

[70] P. Chen, X. Xing, B. Mao, L. Xie, X. Shen, and X. Yin, *Automatic construction of jump-oriented programming shellcode (on the x86)*, in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pp. 20–29, ACM, 2011.

[71] V. Chipounov, V. Kuznetsov, and G. Candea, *S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems*, in *Proceedings of the sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 265–278, 2011.

[72] ForAllSecure, “Unleashing the Mayhem CRS.”
`http://blog.forallsecure.com/2016/02/09/unleashing-mayhem/`.

[73] Trail of Bits Blog, “How We Fared in the Cyber Grand Challenge.”
`http://blog.trailofbits.com/2015/07/15/how-we-fared-in-the-cyber-grand-challenge/`.

[74] P. Mag, *U.S. Barely Cracks List of Countries With Top Wi-Fi Penetration*, 2012.
`http://www.pcmag.com/article2/0,2817,2402672,00.asp`.

[75] Forbes, *When “Smart Homes” Get Hacked: I Haunted A Complete Stranger’s House Via The Internet*, 2013. `http://www.forbes.com/sites/kashmirhill/2013/07/26/smart-homes-hack/`.

[76] D. Davidson, B. Moench, S. Jha, and T. Ristenpart, *FIE on firmware: finding vulnerabilities in embedded systems using symbolic execution*, in *Proceedings of the USENIX Security Symposium*, pp. 463–478, USENIX Association, 2013.

[77] Zaddach, Jonas and Bruno, Luca and Francillon, Aurelien and Balzarotti, Davide, *AVATAR: A framework to support dynamic security analysis of embedded systems’ firmwares*, in *Proceedings of the Network and Distributed System Security Symposium*, 2014. http://www.eurecom.fr/publication/4158.

[78] /dev/ttyS0, *From China, With Love*, 2013.
`http://www.devttys0.com/2013/10/from-china-with-love/`.

[79] /dev/ttyS0, *Reverse Engineering a D-Link Backdoor*, 2013. `http://www.devttys0.com/2013/10/reverse-engineering-a-d-link-backdoor/`.

[80] Tripwire, *SOHO Wireless Router (In)security*, 2014.
`http://www.tripwire.com/register/soho-wireless-router-insecurity/`.

[81] Arstechnica, *Bizarre Attack Infects Linksys Routers With Self-Replicating Malware*, 2014. `http://arstechnica.com/security/2014/02/bizarre-attack-infects-linksys-routers-with-self-replicating-malware/`.

[82] F. Schuster and T. Holz, *Towards reducing the attack surface of software backdoors*, in *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security*, (CCS), (New York, NY, USA), pp. 851–862, ACM, 2013.

[83] /dev/ttypS0, *Finding and Reverse Engineering Backdoors in Consumer Firmware*, 2014. `http://www.devttys0.com/wp-content/uploads/2014/04/FindingAndReversingBackdoors.pdf`.

[84] R. Santamarta, *Here be backdoors: A journey into the secrets of industrial firmware*, in *BlackHat*, 2012.

[85] "CVE-2012-4964."
`http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4964`.

[86] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley, *AEG: Automatic Exploit Generation*, in *Proceedings of the network and Distributed System Security Symposium*, Feb., 2011.

[87] L. Xu, F. Sun, and Z. Su, *Constructing precise control flow graphs from binaries*, 2010.

[88] Akos Kiss, Judit Jasz, Gabor Lehotai, and Tibor Gyimothy, *Interprocedural static slicing of binary executables*, in *Source Code Analysis and Manipulation*, pp. 118–127, IEEE, 2003.

[89] Tok, Teck Bok and Guyer, Samuel Z and Lin, Calvin, *Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers*, in *Compiler Construction*, pp. 17–31, Springer, 2006.

[90] Babić, Domagoj and Martignoni, Lorenzo and McCamant, Stephen and Song, Dawn, *Statically-directed dynamic automated test generation*, in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pp. 12–22, ACM, 2011.

[91] L. De Moura and N. Bjørner, *Z3: An efficient smt solver*, in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340. Springer, 2008.

[92] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, *TaintEraser: protecting sensitive data leaks using application-level taint tracking*, *ACM SIGOPS Operating Systems Review* **45** (2011), no. 1 142–154.

[93] A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, and S. Antipolis, *A large-scale analysis of the security of embedded firmwares*, .

[94] "Re: ION Meter Security." `http://www.powerlogic.com/literature/IONMeterCyberSecurityApril2012.pdf`.

[95] Craig Heffner, "Finding and Reversing Backdoors in Consumer Firmware."
`http://www.devttys0.com/wp-content/uploads/2014/04/FindingAndReversingBackdoors.pdf`.

[96] "86,800 network printers open to the whole Internet." `http://nakedsecurity.sophos.com/2013/01/29/86800-printers-open-to-internet/`.

[97] "Vetting Commodity IT Software and Firmware (VET)."
`http://www.darpa.mil/Our_Work/I2O/Programs/Vetting_Commodity_IT_Software_and_Firmware_(VET).aspx`.

[98] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, *BitBlaze: A new approach to computer security via binary analysis*, in *Proceedings of the International Conference on Information Systems Security*, ICISS, (Berlin, Heidelberg), Springer-Verlag, 2008.

[99] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, *Pin: Building customized program analysis tools with dynamic instrumentation*, in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, (PLDI), 2005.

[100] V. Chipounov, V. Kuznetsov, and G. Candea, *S2E: A platform for in-vivo multi-path analysis of software systems*, *SIGPLAN Not.* **47** (Mar., 2011) 265–278.

[101] P. Godefroid, M. Y. Levin, D. A. Molnar, *et. al.*, *Automated whitebox fuzz testing.*, in *NDSS*, vol. 8, pp. 151–166, 2008.

[102] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, *Automatic exploit generation*, *Communications of the ACM* **57** (2014), no. 2 74–84.

[103] R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, *Ramblr: Making reassembly great again.*, in *NDSS*, 2017.

[104] S. Wang, P. Wang, and D. Wu, *Reassembleable disassembling.*, in *USENIX Security*, pp. 627–642, 2015.

[105] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, *et. al.*, *Automatically patching errors in deployed software*, in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 87–102, ACM, 2009.

[106] Shellphish, *Cyber grand shellphish*, January, 2017. `http://phrack.org/papers/cyber_grand_shellphish.html`.

[107] DARPA, "CFE File Archive." `http://repo.cybergrandchallenge.com/CFE/`.

[108] R. E. Cameron, *A concise economic history of the world: from Paleolithic times to the present.* Oxford University Press, USA, 1993.

[109] A. R. de Castro, *âĂIJthe dissappearing spoon and other true tales from the periodic tableâĂİ, sam kean*, *Revista de Química* **26** (2013), no. 1-2 45.

[110] D. Shahaf and E. Amir, *Towards a theory of ai completeness.*, in *AAAI Spring Symposium: Logical Formalizations of Commonsense Reasoning*, pp. 150–155, 2007.

[111] A. Kosorukoff, *Human based genetic algorithm*, in *Systems, Man, and Cybernetics, 2001 IEEE International Conference on*, vol. 5, pp. 3464–3469, IEEE, 2001.

[112] A. Kosoruko, *Social classification structures. optimal decision making in an organization*, .

[113] J. Barr and L. F. Cabrera, *Ai gets a brain*, *Queue* **4** (2006), no. 4 24.

[114] K. W. Willett, C. J. Lintott, S. P. Bamford, K. L. Masters, B. D. Simmons, K. R. Casteels, E. M. Edmondson, L. F. Fortson, S. Kaviraj, W. C. Keel, *et. al.*, *Galaxy zoo 2: detailed morphological classifications for 304 122 galaxies from the sloan digital sky survey*, *Monthly Notices of the Royal Astronomical Society* (2013) stt1458.

[115] C. B. Eiben, J. B. Siegel, J. B. Bale, S. Cooper, F. Khatib, B. W. Shen, B. L. Stoddard, Z. Popovic, and D. Baker, *Increased diels-alderase activity through backbone remodeling guided by foldit players*, *Nature biotechnology* **30** (2012), no. 2 190–192.

[116] P. Inc, "Peach fuzzer: Discover unknown vulnerabilities." `http://peachfuzzer.com`.

[117] J. Drake, "Stagefright - Blackhat 2015 Slides." `https://www.blackhat.com/docs/us-15/materials/us-15-Drake-Stagefright-Scary-Code-In-The-Heart-Of-Android.pdf`.

[118] T. Verge, "Google rebuilt a core part of android to kill the stagefright vulnerability for good." `http://www.theverge.com/2016/9/6/12816386/android-nougat-stagefright-security-update-mediaserver`.

[119] A. Machiry, R. Tahiliani, and M. Naik, *Dynodroid: An input generation system for android apps*, in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 224–234, ACM, 2013.

[120] Shellphish, "Shellphish - the cyber grand challenge." `http://shellphish.net/cgc`.

[121] J. Caballero, H. Yin, Z. Liang, and D. Song, *Polyglot: Automatic extraction of protocol message format using dynamic binary analysis*, in *Proceedings of the 14th ACM conference on Computer and communications security*, pp. 317–329, ACM, 2007.

[122] Z. Lin, X. Jiang, D. Xu, and X. Zhang, *Automatic protocol format reverse engineering through context-aware monitored execution.*, in *NDSS*, vol. 8, pp. 1–15, 2008.

[123] 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*, pp. 621–634, ACM, 2009.

[124] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, *et. al.*, *SOK:(State of) The Art of War: Offensive Techniques in Binary Analysis*, in *Security and Privacy (SP), 2016 IEEE Symposium on*, pp. 138–157, IEEE, 2016.

[125] IARPA, "STONESOUP Program." `https://www.iarpa.gov/index.php/research-programs`.