Rise of the HaCRS:
Augmenting Autonomous Cyber Reasoning Systems with Human Assistance

Yan Shoshitaishvili
Arizona State University
yans@asu.edu

Michael Weissbacher
Northeastern University
mw@ccs.neu.edu

Lukas Dresel
UC Santa Barbara
lukas.dresel@cs.ucsb.edu

Christopher Salls
UC Santa Barbara
salls@cs.ucsb.edu

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

Christopher Kruegel
UC Santa Barbara
chris@cs.ucsb.edu

Giovanni Vigna
UC Santa Barbara
vigna@cs.ucsb.edu

ABSTRACT

Software permeates every aspect of our world, from our homes to the infrastructure that provides mission-critical services.

As the size and complexity of software systems increase, the number and sophistication of software security flaws increase as well. The analysis of these flaws began as a manual approach, but it soon became apparent that a manual approach alone cannot scale, and that tools were necessary to assist human experts in this task, resulting in a number of techniques and approaches that automated certain aspects of the vulnerability analysis process.

Recently, DARPA carried out the Cyber Grand Challenge, a competition among autonomous vulnerability analysis systems designed to push the tool-assisted human-centered paradigm into the territory of complete automation, with the hope that, by removing the human factor, the analysis would be able to scale to new heights. However, when the autonomous systems were pitted against human experts it became clear that certain tasks, albeit simple, could not be carried out by an autonomous system, as they require an understanding of the logic of the application under analysis.

Based on this observation, we propose a shift in the vulnerability analysis paradigm, from tool-assisted human-centered to human-assisted tool-centered. In this paradigm, the automated system orchestrates the vulnerability analysis process, and leverages humans (with different levels of expertise) to perform well-defined sub-tasks, whose results are integrated in the analysis. As a result, it is possible to scale the analysis to a larger number of programs, and, at the same time, optimize the use of expensive human resources.

In this paper, we detail our design for a human-assisted automated vulnerability analysis system, describe its implementation atop an open-sourced autonomous vulnerability analysis system that participated in the Cyber Grand Challenge, and evaluate and discuss the significant improvements that non-expert human assistance can offer to automated analysis approaches.

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134069

CCS CONCEPTS

• **Security and privacy-Usability in security and privacy**; • **Security and privacy-Vulnerability scanners**;

KEYWORDS

Fuzzing, Human assistance, Cyber Reasoning Systems

1 INTRODUCTION

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) [34], 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 paper, we explore the application of this idea to vulnerability analysis. More 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 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 number 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 paper makes the following contributions:

- 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 we had open-sourced after the contest. 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.

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

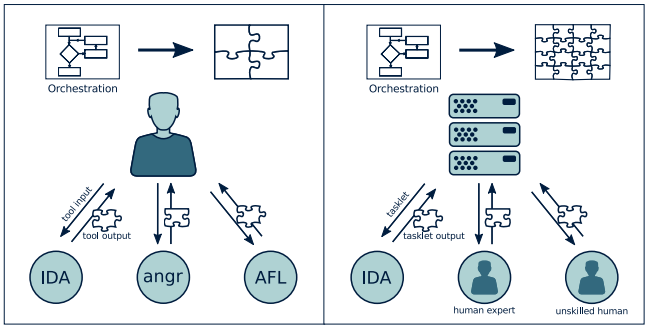

Figure 1: Tool-assisted Human-centered Analysis vs. Human-assisted Tool-centered Analysis.

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.

2 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.

2.1 Fully Automated Analysis

Individual techniques have been developed for identification of vulnerabilities [7, 12, 29], automatic exploitation [1, 13, 14], and automatic application protection [24, 35, 36]. 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 [8, 27].

2.2 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 [6], 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" [17]. 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 [18, 19, 25]. 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 [2], usually used in a CHC context. In fact, we use Mechanical Turk for many of our experiments in this paper. 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 [37], utilizes human-completed micro-tasks for the classification of astronomical images, while another, Foldit [11], aids protein folding algorithms by having humans play "folding games."

2.3 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 [16], attempt to create inputs conforming to a specification that a program is designed to process. Mutational fuzzers, such as AFL [38], 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 [10]:

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 [30].

2.4 Human-*Assisted* Automated Analysis

The Cyber Grand Challenge required a fully autonomous system (CCC, by the definitions in Section 2.2). 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 paper, we propose that limited human assistance can be used in the scope of otherwise-automated binary analysis systems.

Of course, leveraging humans for tasks that are otherwise difficult to automate is a well-explored field. Research in the field of human-computer interaction (HCI) has been focusing on effectively engaging human labor into computer systems to solve hard problems, like labeling images [31], locating objects in images [32], and recognizing characters in images [33]. One way to raise the motivation of human participants is through *gamification*, which has been adopted in security for human-assisted verification [9, 20, 22]. However, the scalability of these techniques and systems are strictly limited by the number of participants of the game, since none of them integrates the output of human users into an autonomous system. One exception has been explored in the context of generating inputs for Android applications, but this concept has never

been investigated in the context of an otherwise-autonomous Cyber Reasoning System [23].

HaCRS takes a different route: It treats its humans as optional assistants, and injects their output into an autonomous cyber reasoning system to improve an already-scalable and fully automated solution. 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.

3 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 paper, 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 [26, 27]. Shellphish designed Mechanical Phish as a set of discrete components, providing individual analysis tasks, united by a central component that handles the "overarching intelligence" [27]. 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 4.

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 paper is *input generation*. In input generation, input test cases are created through both automated and human-assisted techniques to form a base set of test cases 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.

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.

4 THE CYBER REASONING SYSTEM

We based our implementation on the Mechanical Phish, the Cyber Reasoning System developed for the Cyber Grand Challenge and open-sourced by our team (Shellphish) [26]. While Mechanical Phish is composed of modules that are spread over more than 30 different source code repositories, the core design is (or attempts to be) fairly straightforward [26].

In this section, we will describe Mechanical Phish in terms of the computation framework discussed in Section 2.2. 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 2. 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.

4.1 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 implementation 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.

4.2 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. 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 [26].

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.

4.3 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, though there are several exceptions:

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 over simple (but higher-overhead) ones.

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

4.4 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 [29].

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.

4.5 Automated Vulnerability Discovery - Fuzzing

The fuzzing approach in the Mechanical Phish is based on a mutational fuzzer known as American Fuzzy Lop [38]. 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.

4.6 Automated Vulnerability Discovery - Drilling

Driller proposed a mitigation for the stalling of the fuzzer due to the inability to satisfy complex solutions [29]. 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 test cases (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 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 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 test cases, is only a partial solution. This is because, while Driller can make major changes to the input test case it analyzes, it can only (by design and fundamental limitation) achieve 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 paper, 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 *Human-Automation 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.

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

Table 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.

5.1 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., graduate students in Computer Science) can be leveraged more readily than experts. Thus, in this paper, 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, 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 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.

5.2 Human-assisted Input Generation

HaCRS uses human assistance to break through the "semantic barriers" that limit the effectiveness of automated analyses described in Sections 4.5 and 4.6. It gives its human assistants a goal: generate an input test case that executes some amount of code in the target program that has not been reached by previously-known test cases (i.e., those previously found by automated analyses or other humans).

Human assistants interact with the target program to generate test cases, and these test cases are synchronized throughout HaCRS' components.

Human-to-automation. Human-produced test cases 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 test cases produced by other human assistants. This enables a collective effort of the understanding and leveraging of program semantics toward a higher code coverage.

Automation-to-human. The resulting automation-mutated test cases can then be shown to the human assistants (we term such a test case an "example test case"), 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 test cases.

Test case conversion. The synchronization of test cases 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 test cases look like gibberish when shown to a human, which hinders the human's ability to reason semantically about what actions the test case is causing the target program to take.

To address this issue, we use the existing `afl-tmin` utility shipped with AFL [38]. This utility is a *test case minimizer*. It takes an input test case 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.

5.3 Automation-assisted Human Assistance

Simply presenting previously-discovered test cases to human assistants enables an improvement over a base-case Cyber Reasoning System (we show this in Section 6). 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 test case to detect if input data must be provided in a specific format. It achieves this by leveraging existing techniques in protocol recovery [4, 5, 21]. 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 [27]).

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 test cases. 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 test cases 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 test case 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, discards symbolic inputs on the state, 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 test cases 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. In some simple cases (like the example in Figure 2), the symbolic tokens generated by HaCRS can directly be synchronized into Driller

as an end-to-end test case without the need for human assistance. When this holds, it is implicitly handled by the automation. Since the token recovery process discards path constraints and symbolically explores from the introduction location of the first constraint on that token, it loses coherence in other input bytes, as keeping them coherent will (for the exact reasons discussed in Section 4.6) hamper token recovery. Thus, symbolic tokens are generally not directly usable as test-cases, and instead act as inspiration for human assistants.

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 test cases 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 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 2.

The HAL interface in Figure 2 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 5.3 to present the assistant with a better idea of how to make a program behave differently than in the example test case.

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 5.3. 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.

6 EVALUATION

In this section, we evaluate the impact of 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.

6.1 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 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 test cases, 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 [28, 29, 35].

Filtering. 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 5.3) was able to facilitate a human-usable text protocol. We selected these by automatically detecting the presence of non-printable characters

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

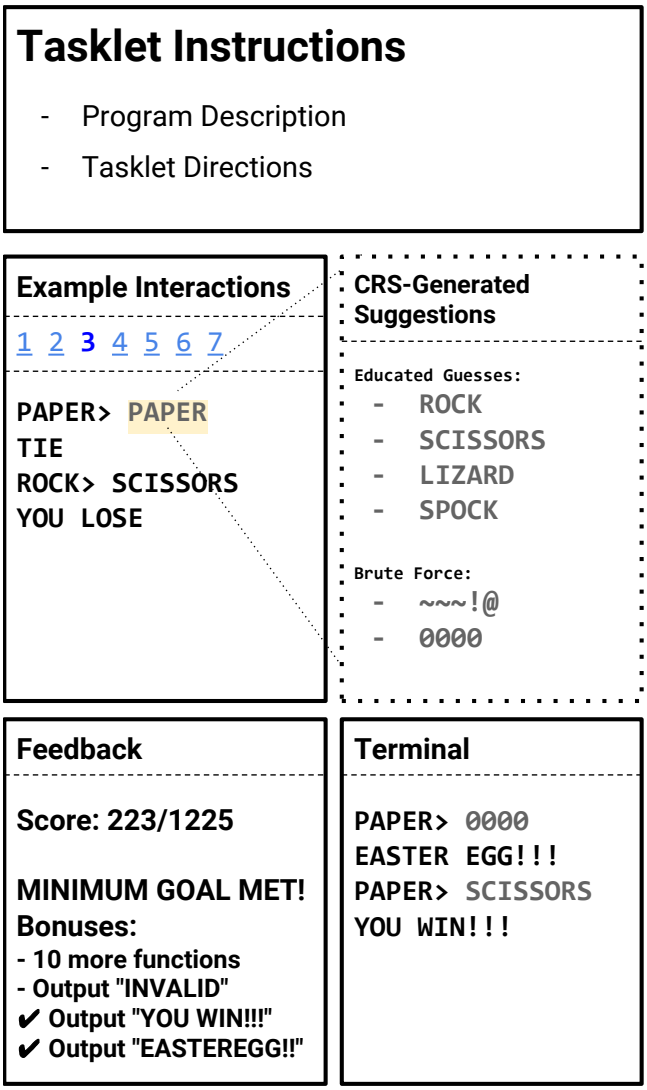

Figure 2: A diagram of the HaCRS user interface. Tasklet instructions were presented to the assistants, who could preview sample interactions (discovered by prior assistants or by the driller component) and use the terminal to create new test cases. The HaCRS provides *symbolic tokens* (the dash-bordered interface widget) to help guide the human intuition of its assistants and presents specific goals to motivate them.

in the author-provided test cases (we did not otherwise use these test cases 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 criterion.

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 [15] 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.

Classification. Even though a protocol might be text only, it might still be hard for humans to 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 3. Classifications were done by one researcher, before experiments were performed (except for binaries used during system development). For borderline cases of semantic complexity, we erred on the side of marking binaries as complex. For borderline cases of required expertise, we erred on "requiring expertise". Our reasoning for the classification of various binaries is provided alongside the classifications. Admittedly, this process is subjective. One way to address this is by having the assistants themselves rate programs by semantic and technical complexity. However, as this classification is not a core part of the system but instead a lens through which to understand its effectiveness on this specific dataset, we felt that such an undertaking would be outside of the scope of this paper.

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.

6.2 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 [2]. 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 test cases "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.

We filtered turkers by success rate (>95%), resulting in 183 turkers solving 802 tasklets at costs between $1 and 4$, scaling with the "age" of the program in the system, plus a bonus of 2 cents per extra percentage of discovered edges. Turkers demonstrated a "long tail" of performance. Our star turker completed 67 tasklets over 48 binaries and contributing to 28 assisted crashes, 8 of which automation alone didn't find, versus an average of 4.5 tasklets (across 1.8 binaries) per worker, contributing to 0.3 assisted crashes, 0.1 of which automation did not find.

Each assistant was presented with the tasklet instructions and the HAL interface. In the end, between the different experiments to fully understand our system, we spent about $1,100 on Mechanical Turk HITs, resulting in 21268 unique test cases across our experiment[3]. While this is a large amount for a research lab, it would be trivial spending for a nation state or large corporation looking to augment their analysis capabilities.

Semi-experts. We recruited five professionals in Computer Science, familiar with programming topics but not with program analysis or security, to act as our semi-expert human assistants. These professionals interacted with a random sampling of 23 binaries from our dataset, generating a total of 115 test cases.

6.3 Human-Automation Link

As we proposed a number of optimizations to the Human-Automation Link in Section 5.3, 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 5.3 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 test cases at which progress stopped being made.

Our improvements to the HAL allowed our assistants to contribute a significantly higher amount of test cases than they were previously able to. For semantically complex binaries, the number of test cases 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 test case generations. On further investigation, this makes sense – analyzing the test cases 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 test cases.

6.4 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 test case 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. Test cases produced by non-experts are presented to semi-experts as examples, and test cases 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.

Human-assisted fuzzing. In this configuration, Driller is replaced with our Human-Automation Link. Rather than symbolically tracing fuzzer-generated test cases, we present them to our human assistants and synchronize their test cases 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.

We ran each configuration for 8 hours, giving the fuzzer 4 processor cores, with 2 additional cores for Driller. The results of the experiment are presented in Table 2.

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:

[3] A preprint of this paper stated $2,000 as the cost. Amazon Mechanical Turk collects the full payment for all scheduled HITs up front, and refunds it at a later date if HITs are not completed. Confusion with the Mechanical Turk interface caused us to mistakenly leave out the refunded portion in our calculations, resulting in the higher figure.

Configuration	Semantic Complex-ity	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 2: 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 test cases and human-originated test cases, respectively, that were deemed "unique" by the Mechanical Phish's test case evaluation criteria.

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.

We analyzed the impact of the fuzzer, Driller, and human assistance on code coverage metrics and the amount of test cases for the binaries that only HaCRS was able to crash in our experiment. This is presented in Figure 3. Unsurprisingly, all of these binaries are ones that we classified as having high semantic complexity. For most of them, HaCRS achieves significantly higher code coverage, but there are several interesting exceptions where the code coverage achieved by HaCRS is very close to or even *lower* than the other techniques, despite it triggering a crash where other methods failed. Our investigation into this phenomena revealed that this is a function of humans triggering the same (or a subset) of the code that the automation does, but doing so in a different configuration more correct (or appropriately *in*correct) for the program being tested. Later in this section, we discuss one such case, `NRFIN_00005`, where automation managed to trigger all of the functionality but it took human intuition to trigger it *correctly* for a crash.

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

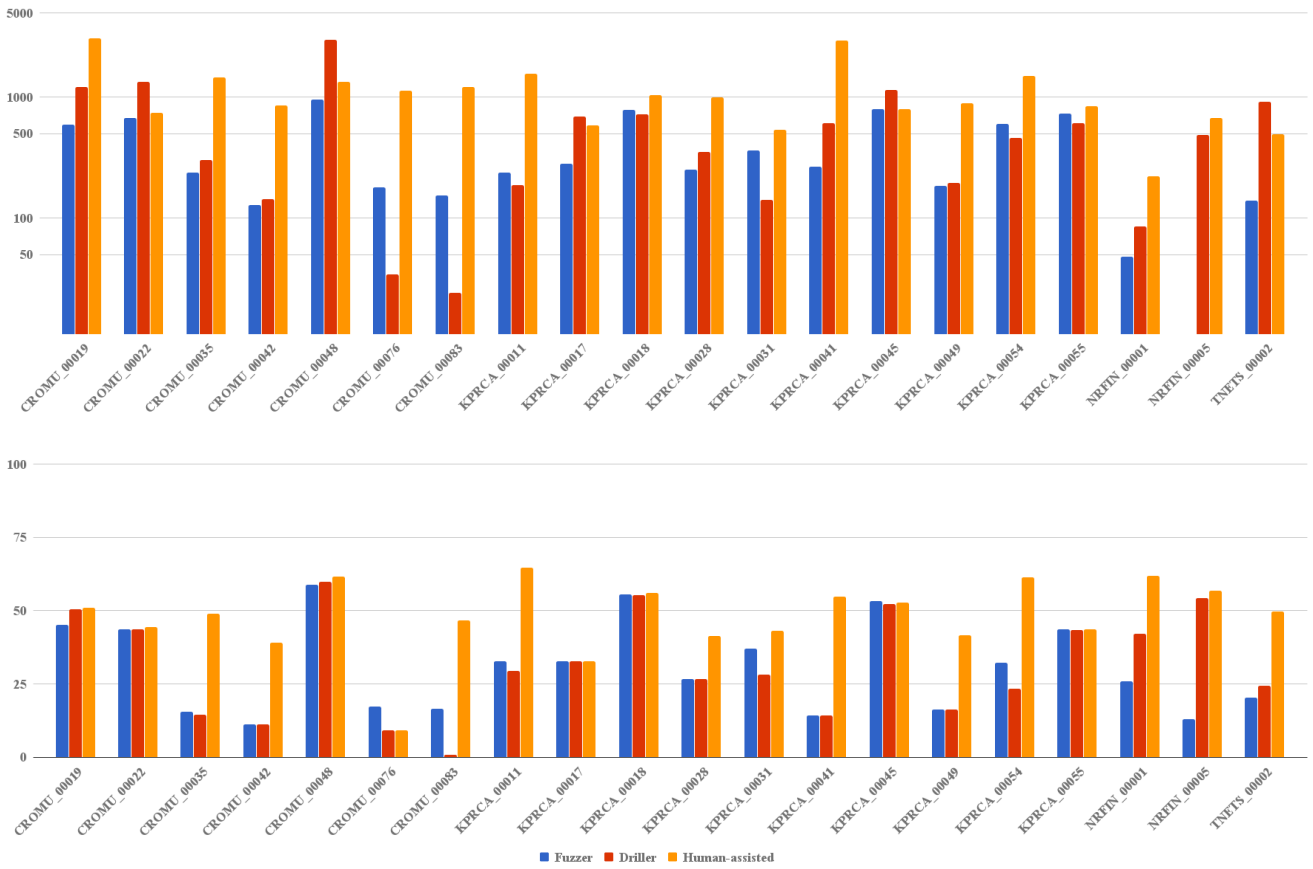

Figure 3: The binaries in which crashes were only found in the human-assisted configuration of the CRS. The number of unique (in terms of code coverage) test cases is shown in the top chart (with a logarithmic scale), and the achieved code coverage is shown in the bottom chart. The results are split into what was found by just the fuzzer alone, by including Driller, and with the addition of human assistance.

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 183 Mechanical Turk workers to assist HaCRS, we were only able to recruit five professionals, 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 automation at all. 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.

6.5 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 analysis engine 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: NRFIN_00005. This binary implements a Tic-Tac-Toe game where the human player plays against the computer. It is made harder by only displaying the moves and hiding the game board. A null pointer dereference bug can be triggered and the program is crashed by playing at least one round and then type "START OVER" to start a new game at the menu. Other options like "PLAY AGAIN" do not trigger any crash immediately.

During the game play, this binary provides a clear indication of what the format of the input should be. We observed that our assistants strictly followed the expected input format and won the first round quite fast, and then typed "PLAY AGAIN" to start a new game. HaCRS then mutated up one of the round-winning input, replaced "PLAY AGAIN" with "START OVER", and triggered the crash immediately. In comparison, neither AFL nor Driller could win a single round of the game: They could not generate input that satisfies the format requirement for sufficient number of times, and consequently they never crashed this binary.

Vulnerability-detection case study: KPRCA_00028. This binary implements a command line interpreter for expressions from an imaginary programming language called SLUR. Built-in functions include `quote`, `cons`, `equal`, `lambda`, etc., can be used inside SLUR expressions. A typical SLUR expression looks like the following: (*quote* `e`), or ((*lambda* (`v1 v2`) `e`) `e1 e2`). A null pointer dereference vulnerability exists in the lambda function.

Although this program does not provide any hint of what build-in functions there are and what form user input should have, we noticed that our assistants were able capture function names from the symbolic tokens in the guidance that HaCRS provided. Additionally, our assistants were able to correctly determine the legitimate format of SLUR expressions, and manually constructed expressions using different built-in functions that are accepted by the binary. With the help of input from our assistants, HaCRS was able to build several long SLUR expressions including one or more lambda subexpressions that triggered the crash. We scrutinized the input that AFL (without human assistance) used, and realized that it was unable to generate an expression in the correct form, which explains why it could not crash this binary.

7 DISCUSSION

In this section, we discuss implications of the Rise of the HaCRS. Specifically, we talk about the importance of our step of integrating human effort into Cyber Reasoning Systems (and specifically, *non-expert* human effort), take-aways from our evaluation, and future steps.

7.1 Human Obsolescence

As with most examples of human-dependent technique, we expect that the "intuition" that human assistants provide for HaCRS will eventually be replaced by automated techniques. However, it is not currently clear what shape such an analysis would take. While carrying out the tasklets that HaCRS requested help with, humans do not necessarily reason about code coverage (even though it is used as a goal metric), but rather about the exploration of an abstract state space of program, in a way that current automated techniques do not consider.

When automated techniques are developed that can reproduce this slice of human intuition, we expect that humans will be made redundant, similar to the relentless advance of automation in assembly lines. For now, however, it seems that we are still quite relevant, even in the simplified Cyber Grand Challenge dataset.

7.2 Assistant Skill Levels

Interestingly, judging from feedback emails sent to us when our human assistants experienced technical issues, more technically-minded assistants tended to get frustrated and quit faster. This may have been due to us attempting to simplify our assistant instructions. Combined with the relatively unimpressive performance of the semi-experts, this implies that more research is needed into presenting a correct abstraction for different skill levels, and that the non-expert interface does not necessarily scale up to expert users.

Of course, software testing specialists are quite well trained for this sort of thing – they excel at identifying corner cases in software despite being "semi-experts" in program analysis. Our future direction is the principled reintroduction of expertise into the process, with the appropriate interface support, to better understand how much of the system is impacted just by expertise-independent human "intuition" and how much is impacted by human experience, but perhaps hampered by the current "simplified" interface.

Given a pool of abundant non-experts, adequate amounts of such semi-experts, and a constrained number experts, HaCRS could strategically distribute different tasks, with interfaces of varying difficulty level, to its host of assistants. There are many open questions as to the best way to facilitate this interaction – should experts inject inputs like non-experts do, or should they function at a lower level of program paths and symbolic constraints? Once such variable levels of assistance are supported, the HaCRS will have to be taught to reason about a *budget*, in terms of the available human talent, available time, and available money to pay its assistants. This requires the potential integration of complex game theory and approach planning algorithms, which are currently relatively unexplored in the realm of Cyber Reasoning Systems.

7.3 Incentive Structures

Because Amazon Mechanical Turk is designed for quick tasks with instant payoff, we settled on the incentive model of paying assistants for triggering a pre-set amount of transitions in the program. However, this ignores, to some extent, the humans' effect on downstream automated analysis. Basically, not all transitions are created equal, and some lead to more interesting mutations than others. Thus, it would be interesting to explore an incentive structure in

which assistants are rewarded based on how much code coverage is achieved by any test cases *derived* from their tasklet solutions, not just the coverage of the solutions themselves. This could allow assistants to more carefully budget their time across different programs, as well: an assistant could put a small amount of time into program P_a, move on to P_b, and check at a later time if his inputs to P_a resulted in increased code coverage from the automation, and provide more assistance as needed.

These sorts of improved incentive structures, that allow the human to use the automation as an assistant at the same time as the automation uses the human as an assistant, may bring the two sides closer toward creating a hybrid "centaur" system.

7.4 Other Tasklets

Thus far, we have integrated human assistance into the test case creation pipeline of the Mechanical Phish. However, the HaCRS concept can be applied to other aspects of a CRS:

Test case selection. The stalling-out of the fuzzer, which HaCRS addresses by providing human-assisted test cases, represents only one side of the limitations of fuzzing-based vulnerability discovery techniques. On the other end of the spectrum is the "input explosion" that can occur when the fuzzer identifies *too many* test cases, overwhelming the evolutionary algorithm. Of course, automated techniques, such as AFLFast [3], have been developed in an attempt to help with the selection of test cases in this situation. However, the fact that human assistants augment a CRS even in the presence of techniques such as Driller suggests that exploring the use of human assistance for test case *selection*, in addition to generation, could be a promising direction of research.

Exploitation. Even though the Mechanical Phish exploited more challenges during the Cyber Grand Challenge than any of its opponents, it still had almost an 80% failure rate in converting a crash to an exploit. In many cases, this was because the specific way in which it triggered a crash did not provide it with enough control over the program's memory. Crashing test cases that the CRS fails to exploit could be dispatched to expert human assistants for "post-processing", and these assistants could modify the test-cases to achieve more control of the state, allowing the CRS to weaponize otherwise-unexploitable crashes.

Patching. One of the limitations of the Mechanical Phish is its inability to create precise patches for software, due to a lack of *root cause analysis* of vulnerabilities. This limitation forces the Mechanical Phish to exclusively adopt costly general patches that patch large swaths of code that are not vulnerable. Integrating human effort into the patch evaluation process, specifically by having experts (or maybe, with a carefully-designed interface, semi-experts) participate in the root-cause analysis of identified crashes and the evaluation of potential fixes, could significantly improve the effectiveness of this component of the Mechanical Phish.

High-level planning. Likewise, human assistance can be leveraged in the planning process – for example, during the Cyber Grand Challenge, it was not always a good idea to patch a vulnerability (in fact, the Mechanical Phish lost its chance at victory because it patched *too many* vulnerabilities [27]). An ability to integrate human advice into the system would go a long way to alleviating current limitations in the ability of the Mechanical Phish to properly respond to changing strategic situations.

We plan to explore some of these applications in our future work in this field.

8 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 2) 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 absence of these humans, these systems are able to operate fully autonomously, just at a lower effectiveness.

In this paper, 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 test cases 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.

ACKNOWLEDGMENTS

This work would not have existed without the amazing attitude of the Shellphish CTF and CGC team, and owe our team-mates enormous debts of gratitude for their support and brilliance over the years. We would also like to thank our many, many workers on Amazon Mechanical Turk, along with our amazing semi-expert volunteers: Xingcheng Chen, Tao Du, Tao Zhan, and two anonymous graduate students.

This material is based on research sponsored by the National Science Foundation under award numbers CNS-1704253 and DGE-1623246, by DARPA under agreement number FA8750-15-2-0084, and by the Office of Naval Research under grant number N00014-15-1-2948. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

REFERENCES

[1] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. 2014. Automatic Exploit Generation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS).*

[2] Jeff Barr and Luis Felipe Cabrera. 2006. AI gets a brain. *Queue* 4, 4 (2006).

[3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1032–1043.

[4] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. 2009. Dispatcher: Enabling Active Botnet Infiltration Using Automatic Protocol Reverse-engineering. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[5] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. 2007. Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[6] Rondo E Cameron. 1993. *A Concise Economic History of the World: From Paleolithic Times to the Present.* Oxford University Press, USA.

[7] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[8] DARPA. 2016. CFE File Archive. (2016). http://repo.cybergrandchallenge.com/cfe/.

[9] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Nathaniel Mote, Brian Walker, Seth Cooper, Timothy Pavlik, and Zoran Popović. 2012. Verification Games: Making Verification Fun. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*. 42–49. https://doi.org/10.1145/2318202.2318210

[10] Joshua Drake. 2015. Stagefright - Blackhat 2015 Slides. (2015). https://www.blackhat.com/docs/us-15/materials/us-15-Drake-Stagefright-Scary-Code-In-The-Heart-Of-Android.pdf.

[11] Christopher B Eiben, Justin B Siegel, Jacob B Bale, Seth Cooper, Firas Khatib, Betty W Shen, Barry L Stoddard, Zoran Popovic, and David Baker. 2012. Increased Diels-Alderase Activity through Backbone Remodeling Guided by Foldit Players. *Nature biotechnology* 30, 2 (2012).

[12] Patrice Godefroid, Michael Y Levin, and David A Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*.

[13] Sean Heelan. 2009. *Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities.* Ph.D. Dissertation. University of Oxford.

[14] Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang, Chung-Wei Lai, Han-Lin Lu, and Wai-Meng Leong. 2012. Crax: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations. In *Proceedings of the IEEE International Conference on Software Security and Reliability (SERE)*.

[15] IARPA. 2010. STONESOUP Program. (2010). https://www.iarpa.gov/index.php/research-programs/stonesoup.

[16] Peach Inc. 2013. Peach Fuzzer: Discover unknown vulnerabilities. (2013). http://peachfuzzer.com.

[17] Sam Kean. 2010. *The Disappearing Spoon: And Other True Tales of Madness, Love, and the History of the World from the Periodic Table of the Elements.* Little, Brown and Company.

[18] Alexander Kosoruko. 2000. Social Classification Structures: Optimal Decision Making in an Organization. *Late breaking papers of GECCO* (2000), 175–178.

[19] Alex Kosorukoff. 2001. Human based genetic algorithm. In *Systems, Man, and Cybernetics, 2001 IEEE International Conference on*, Vol. 5.

[20] Wenchao Li, Sanjit a Seshia, and Somesh Jha. 2012. CrowdMine: Towards Crowdsourced Human-Assisted Verification. In *Proceedings of the 49th Annual Design Automation Conference*. ACM, 1254–1255. https://doi.org/10.1145/2228360.2228590

[21] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. 2008. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*.

[22] Heather Logas, Jim Whitehead, Michael Mateas, Richard Vallejos, Lauren Scott, Dan Shapiro, John Murray, Kate Compton, Joseph Osborn, Orlando Salvatore, Zhongpeng Lin, Michael Shavlovsky, Daniel Cetina, Shayne Clementi, and Chris Lewis. 2014. Software Verification Games: Designing Xylem, The Code of Plants. In *Proceedings of the 9th International Conference on the Foundations of Digital Games (FDG)*. Society for the Advancement of the Science of Digital Games, Liberty of the Seas, Caribbean.

[23] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*.

[24] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, et al. 2009. Automatically Patching Errors in Deployed Software. In *Proceedings of the ACM SIGOPS Symposium on Operating systems principles*.

[25] Dafna Shahaf and Eyal Amir. 2007. Towards a Theory of AI Completeness. In *AAAI Spring Symposium: Logical Formalizations of Commonsense Reasoning*.

[26] Shellphish. 2016. Shellphish - The Cyber Grand Challenge. (2016). http://shellphish.net/cgc.

[27] Shellphish. 2017. Cyber Grand Shellphish. (2017). http://phrack.org/papers/cyber_grand_shellphish.html.

[28] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. SOK:(State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[29] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*.

[30] The Verge. 2016. Google rebuilt a core part of Android to kill the Stagefright vulnerability for good. (2016). http://www.theverge.com/2016/9/6/12816386/android-nougat-stagefright-security-update-mediaserver.

[31] Luis Von Ahn and Laura Dabbish. 2004. Labeling Images with a Computer Game. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 319–326.

[32] Luis Von Ahn, Ruoran Liu, and Manuel Blum. 2006. Peekaboom: A Game for Locating Objects in Images. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 55–64.

[33] Luis Von Ahn, Benjamin Maurer, Colin McMillen, David Abraham, and Manuel Blum. 2008. reCAPTCHA: Human-Based Character Recognition via Web Security Measures. *Science* 321, 5895 (2008), 1465–1468.

[34] Mike Walker. 2016. The DARPA Cyber Grand Challenge. https://www.cybergrandchallenge.com/. (2016).

[35] Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*.

[36] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable Disassembling. In *Proceedings of the USENIX Security Symposium*.

[37] Kyle W Willett, Chris J Lintott, Steven P Bamford, Karen L Masters, Brooke D Simmons, Kevin RV Casteels, Edward M Edmondson, Lucy F Fortson, Sugata Kaviraj, William C Keel, et al. 2013. Galaxy Zoo 2: detailed morphological classifications for 304,122 galaxies from the Sloan Digital Sky Survey. *Monthly Notices of the Royal Astronomical Society* (2013).

[38] Michal Zalewski. 2014. American Fuzzy Lop. (2014). http://lcamtuf.coredump.cx/afl/.

Binary	Semantic Complexity	Technical Complexity	Reasoning
CADET_00001	High	Low	Palindrome detection. Fully usable by unskilled users. ASCII protocol.
CADET_00003	High	Low	Palindrome detection. Fully usable by unskilled users. ASCII protocol.
CROMU_00001	High	Low	Messaging application, menu driven. ASCII protocol.
CROMU_00003	High	Low	Fingerd service, menu driven. ASCII protocol.
CROMU_00005	High	Low	Game vaguely based on chess allows for human intuition. ASCII protocol.
CROMU_00017	High	Low	Equation solver, menu driven. ASCII protocol.
CROMU_00019	High	Low	Text-based game. ASCII protocol.
CROMU_00029	High	Low	Temperature simulation, menu driven, simple usability. ASCII protocol.
CROMU_00031	High	Low	Game vaguely based on chess allows for human intuition. ASCII protocol.
CROMU_00037	High	Low	Messaging application, menu driven. ASCII protocol.
CROMU_00040	High	Low	Recipe Management, rich with semantic information. ASCII protocol.
CROMU_00041	High	Low	Simple e-mail program, easy to exercise. ASCII protocol.
CROMU_00044	High	Low	Messaging application, menu driven. ASCII protocol.
CROMU_00046	High	Low	Text-based game, easy to understand. ASCII protocol.
CROMU_00054	High	Low	ECM and TCM simulation, menu driven. ASCII protocol.
CROMU_00065	High	Low	Black jack game allows for human intuition, menu driven. ASCII protocol.
CROMU_00076	High	Low	Text-based game, easy to understand. ASCII protocol.
CROMU_00087	High	Low	Recipe management, rich with semantic information. ASCII protocol.
EAGLE_00005	High	Low	Hangman-style text-based game allows for human intuition. ASCII protocol.
KPRCA_00011	High	Low	Movie rental simulation, menu driven. ASCII protocol.
KPRCA_00017	High	Low	Hangman-style text-based game allows for human intuition. ASCII protocol.
KPRCA_00018	High	Low	Gambling simulation, allows for human intuition. ASCII protocol.
KPRCA_00022	High	Low	Online job application simulation, allows for human intuition. ASCII protocol.
KPRCA_00023	High	Low	Online job application simulation, allows for human intuition. ASCII protocol.
KPRCA_00026	High	Low	Memo auto-correction service, menu driven. ASCII protocol.
KPRCA_00030	High	Low	Game vaguely based on Game of Life, allows for human intuition. ASCII protocol.
KPRCA_00042	High	Low	Movie rental simulation, menu driven. ASCII protocol.
KPRCA_00043	High	Low	Lyrics submission service, allows for human intuition. ASCII protocol.
KPRCA_00049	High	Low	Note taking service, interaction described in documentation. ASCII protocol.
KPRCA_00051	High	Low	Class scheduling service, menu driven. ASCII protocol.
KPRCA_00052	High	Low	Pizza ordering simulation, allows for human intuition. ASCII protocol.
KPRCA_00053	High	Low	Text-based social network simulation, allows for human intuition. ASCII protocol.
KPRCA_00055	High	Low	Matrix calculation service, allows for human intuition. ASCII protocol.
KPRCA_00071	High	Low	Collection of online games, easy to play. ASCII protocol.
KPRCA_00079	High	Low	Space Invaders based game, easy to play. ASCII protocol.
NRFIN_00004	High	Low	Chat bot service, allows for human intuition. ASCII protocol.
NRFIN_00005+	High	Low	Text-based game based on Tic-Tac-Toe, allows for human intuition. ASCII protocol.
NRFIN_00065	High	Low	Computer game collection, allows for human intuition. ASCII protocol.
TNETS_00002	High	Low	Virtual pet management, navigation not described in README but users can explore. ASCII protocol.
YAN01_00001	High	Low	Game based on Ships, allows for human intuition. ASCII protocol.
CROMU_00002	High	High	Particle simulator, limited interaction for humans possible. ASCII protocol.
CROMU_00008	High	High	Database system similar to SQL, allows for human intuition. ASCII protocol.
CROMU_00009	High	High	Online file system and shell, allows for human intuition. ASCII protocol.
CROMU_00010	High	High	Online route calculation and map upload. ASCII protocol.
CROMU_00011	High	High	Online set operations, commands may be unclear, limited interaction possible. ASCII protocol.
CROMU_00014	High	High	Diver logging, menu driven, clear command hints, allows for human intuition. ASCII protocol.
CROMU_00015	High	High	Markup language parser, menu driven, language syntax is complex but allows for some interaction. ASCII protocol.
CROMU_00021	High	High	Integer calculator using RPN, allows for human intuition. ASCII protocol.
CROMU_00022	High	High	Diver logging, menu driven, clear command hints, allows for human intuition. ASCII protocol.
CROMU_00023	High	High	Diver logging, menu driven, clear command hints, allows for human intuition. ASCII protocol.
CROMU_00035	High	High	Online file system, menu driven, allows for human intuition. ASCII protocol.
CROMU_00042	High	High	Combination of sort, arithmetics, and a game. Menu driven, allows for human intuition. ASCII protocol.
CROMU_00048	High	High	Water treatment facility simulator, very verbose, semantic information. ASCII protocol.
CROMU_00051	High	High	Busybox inspired shell environment, verbose. ASCII protocol.
CROMU_00071	High	High	Online flight planning tool, allows for human intuition. ASCII protocol.
CROMU_00083	High	High	Online file system, interactive menu. ASCII protocol.
CROMU_00096	High	High	Text-based interactive shell. Single character commands are guessable for humans. ASCII protocol.
CROMU_00098	High	High	Sendmail inspired email server, allows for human intuition. ASCII protocol.
KPRCA_00007	High	High	Simulated router interface, little feedback, some interaction possible with no documentation. ASCII protocol.
KPRCA_00013	High	High	Spreadsheet calculator, familiarity for humans plus descriptive instructions. ASCII protocol.
KPRCA_00021	High	High	Import and pretty print of JSON based format. Little documentation, limited interaction possible. ASCII protocol.
KPRCA_00028	High	High	LISP-like evaluation engine, little documentation but usable error feedback that can lead to better input. ASCII protocol.
KPRCA_00031	High	High	Chat bot service using hidden markov chains. Sample commands available in documentation. ASCII protocol.
KPRCA_00036	High	High	String matching service, verbose clues. ASCII protocol.
KPRCA_00041	High	High	File comparison service, verbose clues. ASCII protocol.
KPRCA_00045	High	High	Template engine, clues via responses. ASCII protocol.
KPRCA_00054	High	High	Shell, documentation plus help menu. ASCII protocol.
KPRCA_00068	High	High	Audio importing and editing service, menu is verbose. ASCII protocol.
LUNGE_00002	High	High	FAR section lookup service, hard to understand without documentation, some interaction possible. ASCII protocol.
NRFIN_00001+	High	High	SNMP inspired service, verbose menu, allows for human intuition. ASCII protocol.
NRFIN_00009+	High	High	Music store client, limited documentation and feedback, basic exploration possible. ASCII protocol.
NRFIN_00054	High	High	Path tracing service, easily navigable, verbose menu instructions. ASCII protocol.
NRFIN_00055	High	High	Integer calculator, limited documentation, commands can be explored intuitively. ASCII protocol.
YAN01_00002	High	High	Tennis Ball Motion Calculator, little documentation, limited interaction possible. ASCII protocol.
YAN01_00007	High	High	Password wallet, little documentation, limited interaction possible. ASCII protocol.
YAN01_00011	High	High	Word completion game, verbose clues, allows for human intuition. ASCII protocol.
CROMU_00025*	Low	High	Image uploading service using custom file formats, binary data.
CROMU_00030*	Low	High	Game requiring specific format that is described in README. ASCII protocol.
CROMU_00034*	Low	High	File storage for specifically formatted files. Format is described in README.
KPRCA_00010*	Low	High	File storage and processing of PCM data format. Basic interaction requires no knowledge of the format.
KPRCA_00064*	Low	High	Secure compression service. Basic interaction (menu) requires no knowledge of the format.
NRFIN_00008*	Low	Low	ASCII video streaming with video upload. ASCII protocol.
NRFIN_00064	Low	Low	Text-based game, the grid is not displayed to the user, increasing difficulty but still navigable. ASCII protocol.
NRFIN_00069+	Low	Low	Text-based game, the grid is not displayed to the user, increasing difficulty but still navigable. ASCII protocol.
YAN01_00015	Low	Low	Text-based game with fixed hardcoded responses that need to be given. ASCII protocol.

Table 3: The binaries in our dataset, grouped by semantic complexity of their operation and the required technical (Computer Science) expertise, along with justifications for their classification. 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-interactive. 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.