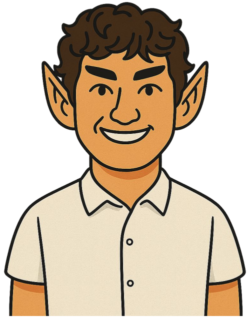
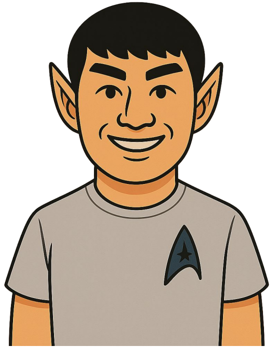
VULCAN: Vulnerable Logic Discovered with Automated Intent Analysis

Arizona State University
Kickoff Meeting
11/25/25

Team Introduction



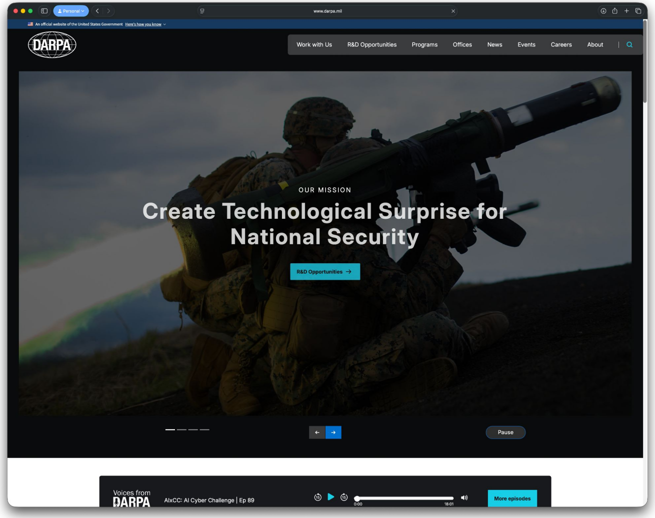



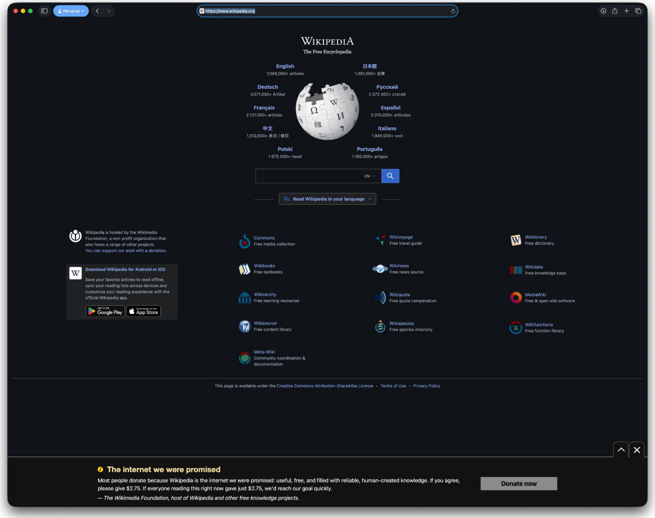


When is a bug not a bug?

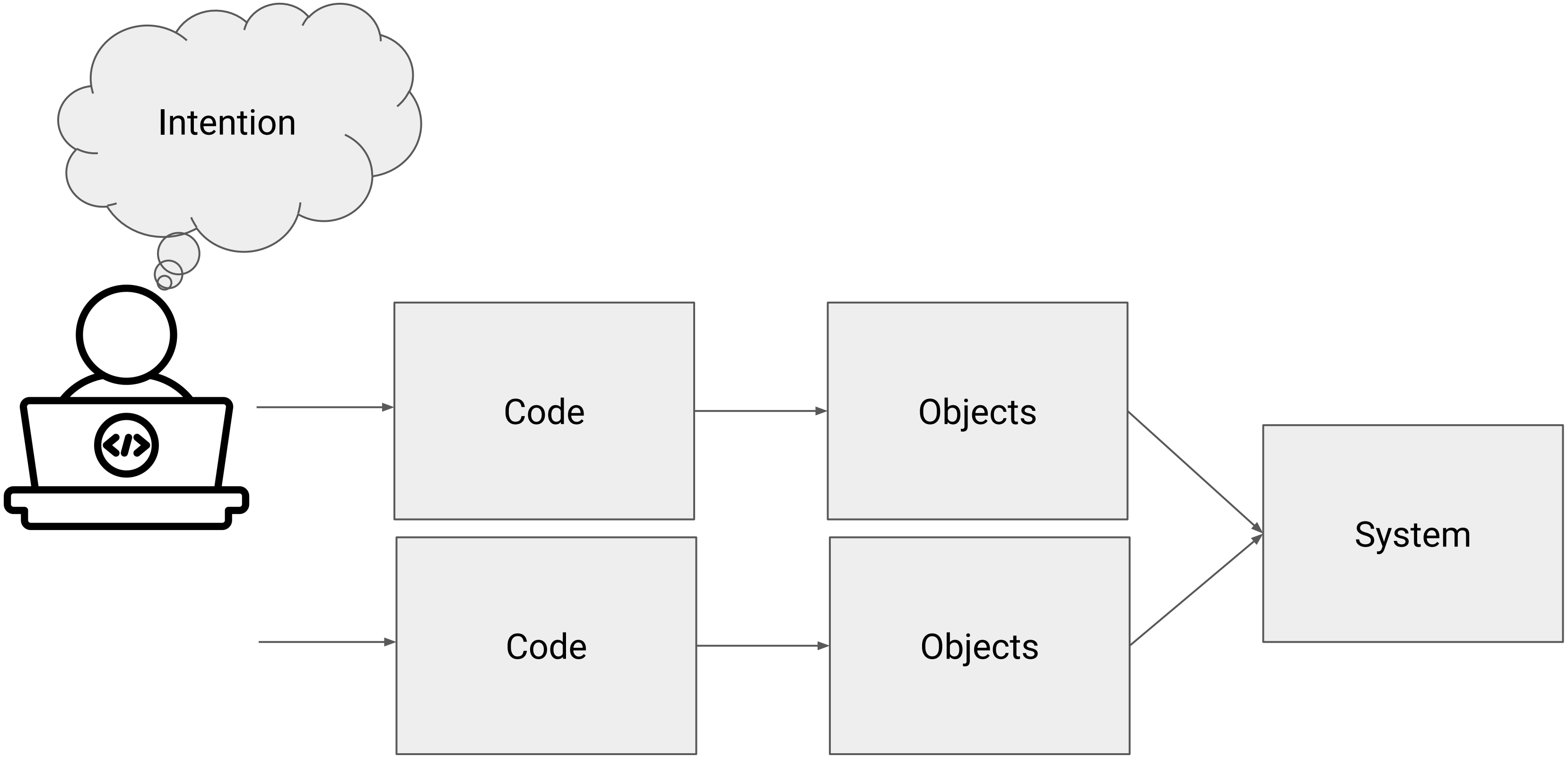
Behavior: Ability to change content of any page on a website.

Is it a bug?

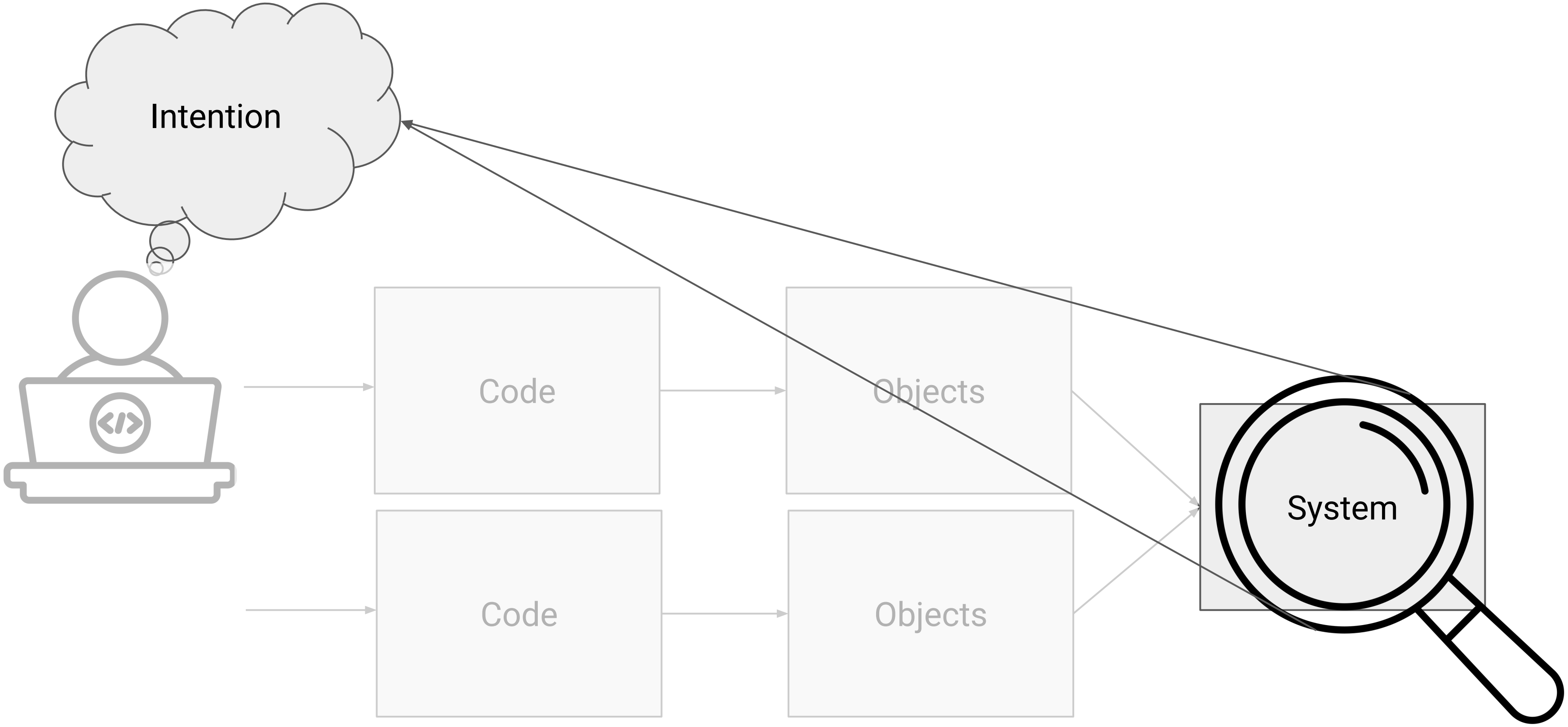




Logic Bugs



Detecting Logic Bugs



Active Research Area... For Web Applications

Toward Automated Detection of
Logic Vulnerabilities in Web Applica

Viktoria Felmetsger Ludovico Cavedon Christopher Kru
[rusvika,cavedon,chris,vigna]@cs.ucsb.ec
Computer Security Group
Department of Computer Science
University of California, Santa Barbara

Abstract

Web applications are the most common way to make services and data available on the Internet. Unfortunately, with the increase in the number and complexity of these applications, there has also been an increase in the number and complexity of vulnerabilities. Current techniques to identify security problems in web applications have mostly focused on input validation flaws, such as cross-site scripting and SQL injection, with much less attention devoted to application logic vulnerabilities.

Application logic vulnerabilities are an important class of defects that are the result of faulty application logic. These vulnerabilities are specific to the functionality of particular web applications, and, thus, they are extremely difficult to characterize and identify. In this paper, we propose a first step toward the automated detection of application logic vulnerabilities. To this end, we first use dynamic analysis and observe the normal operation of a web application to infer a simple set of behavioral specifications. Then, leveraging the knowledge about the typical execution paradigm of web applications, we filter the learned specifications to reduce false positives, and we use model checking over symbolic input to identify program paths that are likely to violate these specifications under specific conditions, indicating the presence of a certain type of web application logic flaws. We developed a tool, called Waler, based on our ideas, and we applied it to a number of web applications, finding previously-unknown logic vulnerabilities.

financial constraints. .
plication vulnerabiliti
flected in the Symant
Report, which was pul
port states that, in 20
for 63% of the total nu

Most recent researc
applications has focus
gation of input valida
bilities is characterize
uses external input as
out first checking or
examples of input val
ing (XSS) [20] and SQ
With XSS, an applica
not sufficiently check
ject malicious JavaSc
then executed on the c
injection, an attacker p
the intended meaning

One reason for the p
nerabilities is that it is
general specification
teristics of these vuln
gramming environmen
functions that read inp
tions that represent se
sinks), and a set of fu
cious content. Then,

Fear the EAR: Discovering and Mitigating Execution After Redirect Vulnerabilities

Adam Doupé, Bryce Boe, Christopher Kruegel, and Giovanni Vigna
University of California, Santa Barbara
{adoupe, bboe, chris, vigna}@cs.ucsb.edu

ABSTRACT

The complexity of modern web applications makes it difficult for developers to fully understand the security implications of their code. Attackers exploit the resulting security vulnerabilities to gain unauthorized access to the web application environment. Previous research into web application vulnerabilities has mostly focused on input validation flaws, such as cross-site scripting and SQL injection, while logic flaws have received comparably less attention.

In this paper, we present a comprehensive study of a relatively unknown logic flaw in web applications, which we call Execution After Redirect, or EAR. A web application developer can introduce an EAR by calling a redirect method under the assumption that execution will halt. A vulnerability occurs when server-side execution continues after the developer's intended halting point, which can lead to broken/insufficient access controls and information leakage. We start with an analysis of how susceptible applications written in nine web frameworks are to EAR vulnerabilities. We then discuss the results from the EAR challenge contained within the 2010 International Capture the Flag Competition. Finally, we present an open-source, white-box, static analysis tool to detect EARs in Ruby on Rails web applications. This tool found 3,944 EAR instances in 18,127 open-source applications. Finally, we describe an approach to prevent EARs in web frameworks.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]

General Terms

Security

Keywords

static analysis, web applications, execution after redirect

1. INTRODUCTION

An increasing number of services are being
line. For example, banking, shopping, socializ
the news, and enjoying entertainment are all ava
web. The increasing amount of sensitive data st
applications has attracted the attention of cyb
who break into systems to steal valuable infor
as passwords, credit card numbers, social secur
and bank account credentials.

Attackers use a variety of vulnerabilities to
applications. In 2008, Albert Gonzalez was
later convicted of stealing 40 million credit anc
from major corporate retailers, by writing S(
attacks [20, 30]. Another common vulnerabili
scripting (XSS), is the second highest-ranked
OWASP top ten security risks for web applicat
injection attacks like SQL injection [29]. Th
jection and XSS have received a large amount
by the security community. Other popular wel
vulnerabilities include cross site request forgery
HTTP parameter pollution (HPP) [3, 12], HT
splitting [27], and clickjacking [2, 21].

In this paper, we present an in-depth study of a
real-world web application logic flaw; one we ar
ecution After Redirect (EAR). An EAR occur
a developer's misunderstanding of how the wel
framework operates. In the normal workflow of
cation, a user sends a request to the web appl
web application receives this request, performs
side processing, and returns an HTTP respor
the HTTP response can be a notification that
web browser) should look elsewhere for the r
source. In this case, the web application sets
response code to `301`, `302`, `303`, or `307`, and a
`tion` header [32]. These response codes instruct
to look for the resource originally requested at
specified by the web application in the HTT
header [31]. This process is known as redirecti
application redirects the user to another resour

Intuitively, one assumes that a redirect shoul

Toward Black-Box Detection of Logic Flaws in Web Applications

Giancarlo Pellegrino
EURECOM, France
SAP Product Security Research, France
giancarlo.pellegrino@eurecom.fr

Davide Balzarotti
EURECOM, France
davide.balzarotti@eurecom.fr

***Abstract*—Web applications play a very important role in many critical areas, including online banking, health care, and personal communication. This, combined with the limited security training of many web developers, makes web applications one of the most common targets for attackers.**

In the past, researchers have proposed a large number of white- and black-box techniques to test web applications for the presence of several classes of vulnerabilities. However, traditional approaches focus mostly on the detection of input validation flaws, such as SQL injection and cross-site scripting. Unfortunately, logic vulnerabilities specific to particular applications remain outside the scope of most of the existing tools and still need to be discovered by manual inspection.

In this paper we propose a novel black-box technique to detect logic vulnerabilities in web applications. Our approach is based on the automatic identification of a number of behavioral patterns starting from few network traces in which users interact with a certain application. Based on the extracted model, we then generate targeted test cases following a number of common attack scenarios.

We applied our prototype to seven real world E-commerce web applications, discovering ten very severe and previously-unknown logic vulnerabilities.

I. Introduction

Web applications play a very important role in many critical areas, and are currently trusted by billions of users to perform financial transactions, store personal information, and communicate with their friends. Unfortunately, this makes web applications one of the primary targets for attackers interested in a wide range of malicious activities.

To mitigate the existing threats, researchers have proposed a large number of techniques to automatically test web applications for the presence of several classes of vulnerabilities. Existing solutions span from black-box fuzzers and pentesting tools to static analysis systems that parse the source code of an application looking for well-defined vulnerability patterns. However, traditional approaches focus mostly on the detection of input validation flaws, such as SQL injection and cross-site scripting. To date, more subtle vulnerabilities specific to the logic of a particular application are still discovered by manual inspection [33].

Logic vulnerabilities still lack a formal definition, but, in general, they are often the consequence of an insufficient validation of the business process of a web application. The resulting violations may involve both the control plane (i.e., the navigation between different pages) and the data plane (i.e., the data flow that links together parameters of different pages). In the first case, the root cause is the fact that the application fails to properly enforce the sequence of actions performed by the user. For example, an application may not require a user to log in as administrator to change the database settings (authentication bypass), or it may not check that all the steps in the checkout process of a shopping cart are executed in the right order. Logic errors involving the data flow of the application are caused instead by failing to enforce that the user cannot tamper with certain values that propagate between different HTTP requests. As a result, an attacker can try to replay expired authentication tokens, or mix together the values obtained by running several parallel sessions of the same web application.

Formal specifications describing the evolution of the internal state and of the expected user behavior are almost never available for web applications. This lack of documentation makes it very hard to find logic vulnerabilities. For example, while being able to add several times the same product to a shopping cart is a common feature, being able to add several times the same discount code is likely a logic vulnerability. A human can easily understand the difference between these two scenarios, but for an automated scanner without the proper application model it is very hard to tell the two behaviors apart.

Key Insight

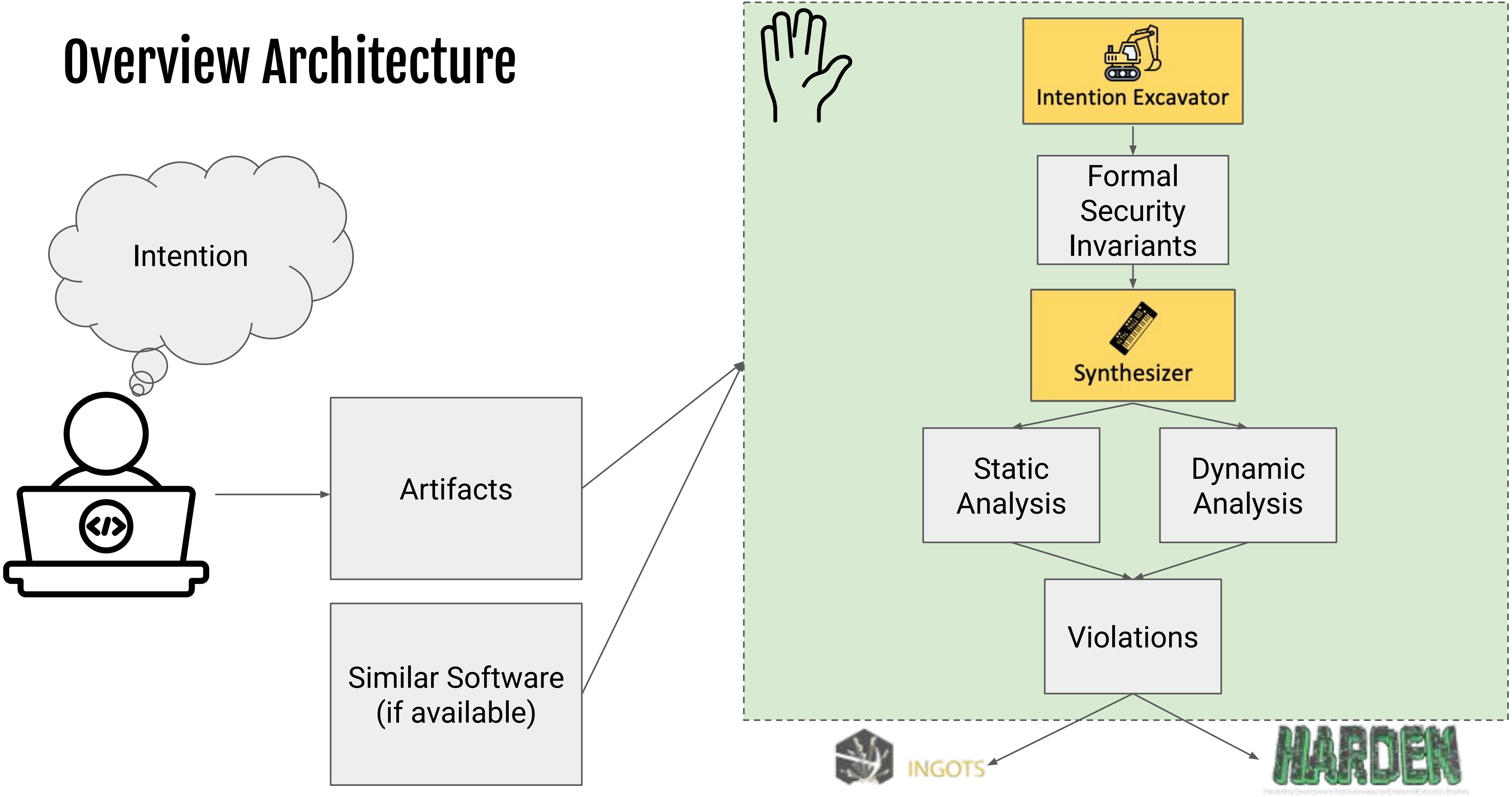
Application context is key.

Similar software shares similar intent.

LLMs contain *world knowledge* that can help identify *expected* behaviors.

Synthesize invariants that must hold in the system.

Overview Architecture



Test Target

Bluetooth Authorization.

Intent Communication.

Inter-process Communication.

Permission Control.

Unexpected Directory Traversal.

... and more

Thrust 1—Intent Recovery

Goal. Extract target-specific intent of a given target.

Risks. Extracted intents are (1) too general, or (2) cannot be synthesized. LLMs are unable to synthesize intents.

Mitigations. Extract using different variants of the target; Resort to human experts.

Deliverables. Techniques that can recover intents with preliminary intent specification and examples.

Thrust 2—Target-Specific Security Invariant Synthesis

Goal. Automatically synthesize target-specific, run-time machine-verifiable security invariants that Thrust 1 generates.

Risks. Some intents cannot be synthesized into invariants; Certain types of invariants may be too costly to verify during runtime.

Mitigations. Co-development of an invariant language for synthesizing invariants from intents; Developing novel sanitizers using new hardware features on AArch64.

Deliverables. Technique that synthesize invariants to run-time monitors and sanitizers.

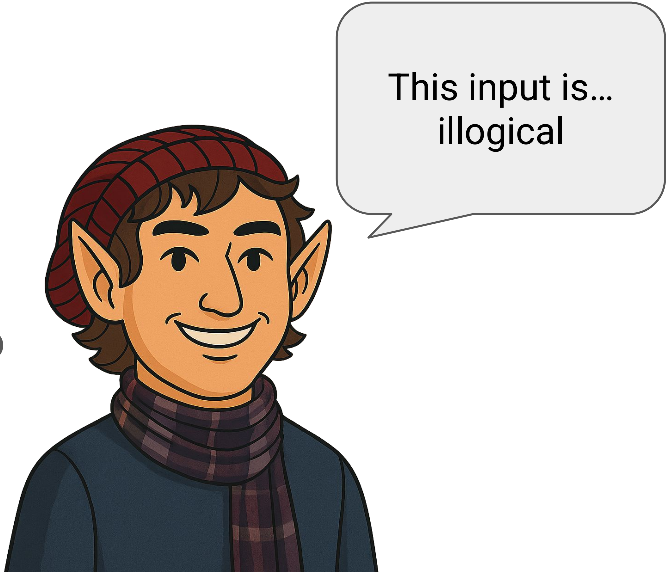
Thrust 3—Counterexample Synthesis and Triage

Goal. Automatically find counterexamples that violate the synthesized invariants.

Risks. Dynamic analysis techniques for Android are hard to use. Symbolic execution does not scale. Difficulty in collecting traces.

Mitigations. Apply Java-based fuzzers. Develop scalable symbolic execution techniques.

Deliverables. Dynamic and static analysis techniques to identify inputs that violate invariants.



Thrust 4—Weird-Machine Instruction Modeling and Program Synthesis

Goal. Lift and abstract the target-specific security invariants into a weird machine instruction, and combine them with other weird machine instructions and synthesize a weird machine program.

Risks. Unable to express security invariants in UTE.

Mitigations. Model primitives of security invariants and express security invariants as combinations of these primitives.

Deliverables. Language to describe target-specific security invariants in the context of weird machines, and UTE models for target-specific security invariants.

Success Criteria

Each aspect of the system has different criteria for success, in this way we will know we are progressing.

Metrics

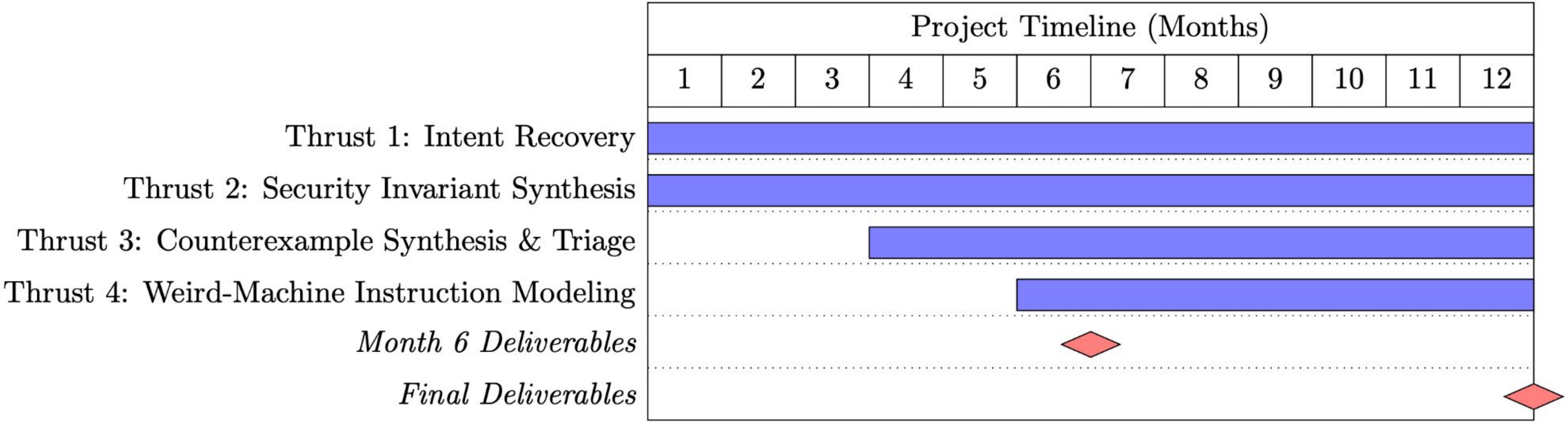
- Explainability of intent.
- Explainability of Bug and bug classes.
- Applicability of bug classes.
- Cost of resources: machine, human, and AI.

Success Criteria

- Month 6: One logic bug class, one Android component.
- Month 12: Two logic bug classes, three Android components.

All the logic bugs will be ingested by UTE.

Schedule



Project Timeline (Months)
1
2
3
4
5
6
7
8
9
10
11
12
Thrust 1: Intent Recovery
Thrust 2: Security Invariant Synthesis
Thrust 3: Counterexample Synthesis & Triage
Thrust 4: Weird-Machine Instruction Modeling
Month 6 Deliverables
Final Deliverables

Live Long and Prosper... Free of Logic Bugs