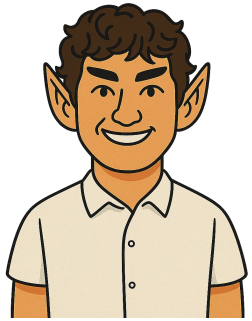
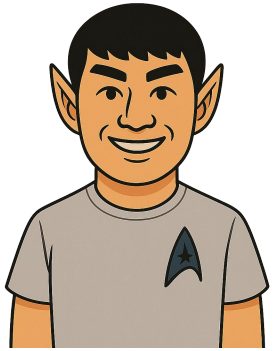
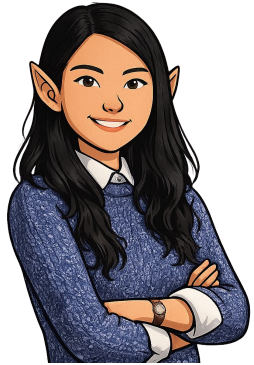
VULCAN: Vulnerable Logic Discovered with Automated Intent Analysis

Arizona State University
HARDEN PI Meeting
9/2/26

Team Introduction



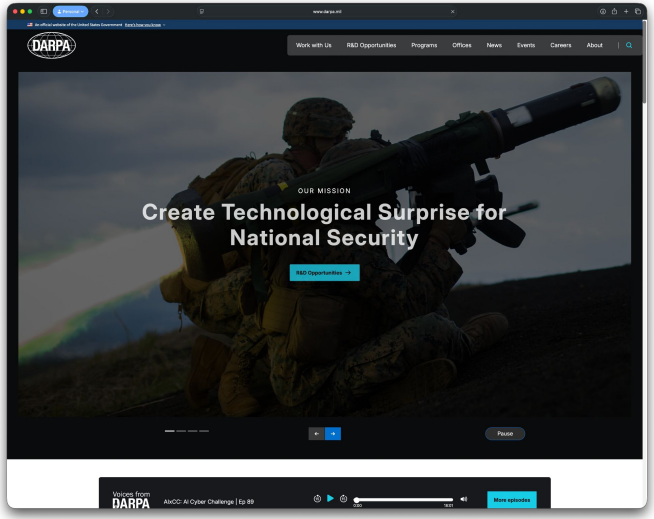


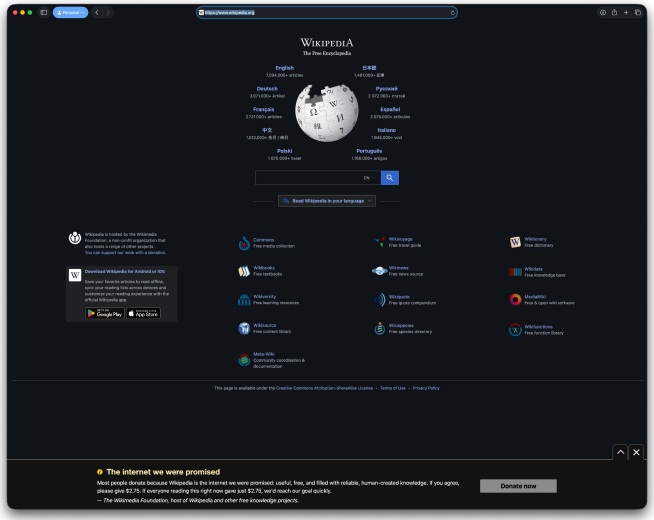


When is a bug not a bug?

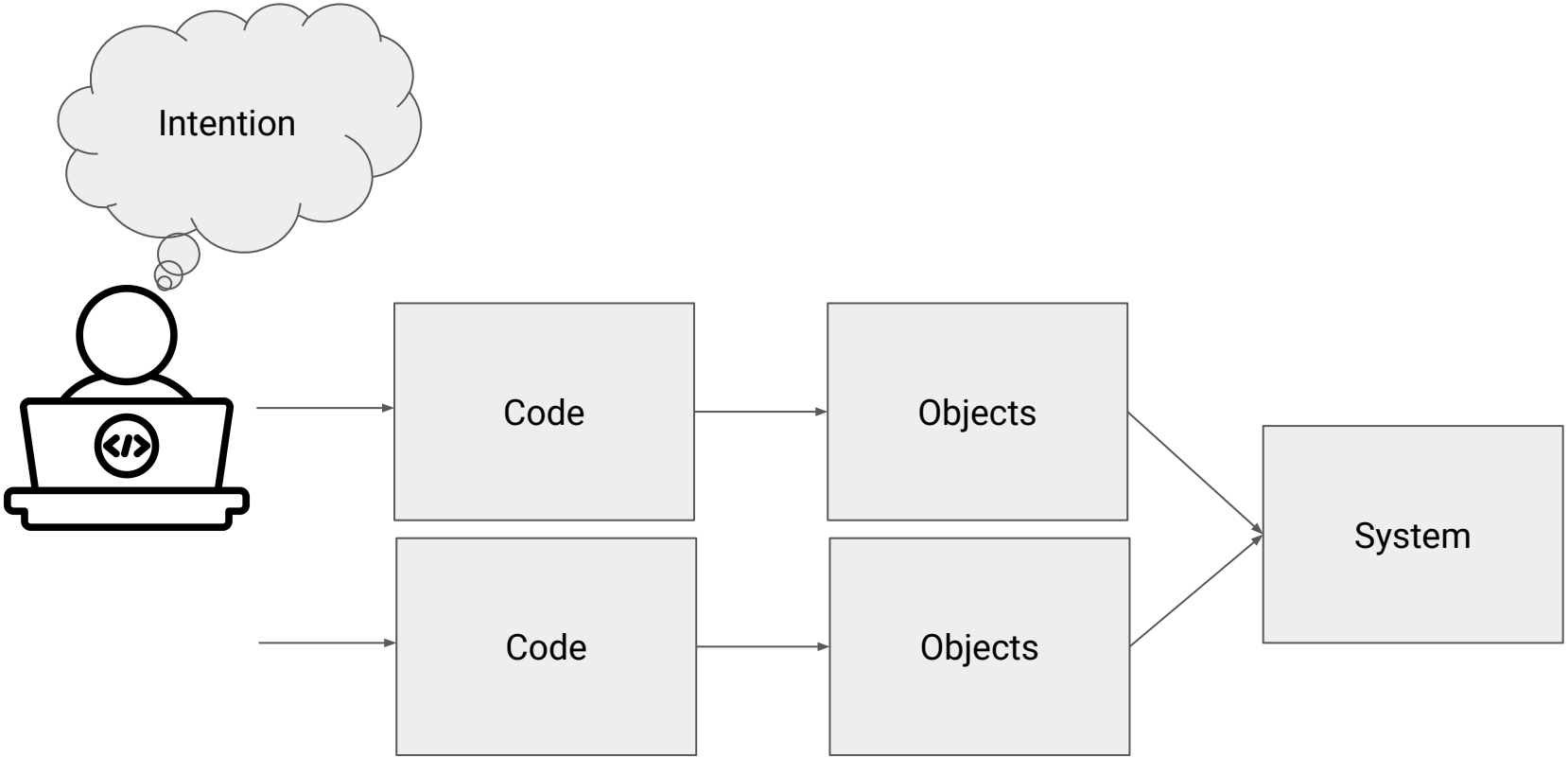
Behavior: Ability to change content of any page on a website.

Is it a bug?

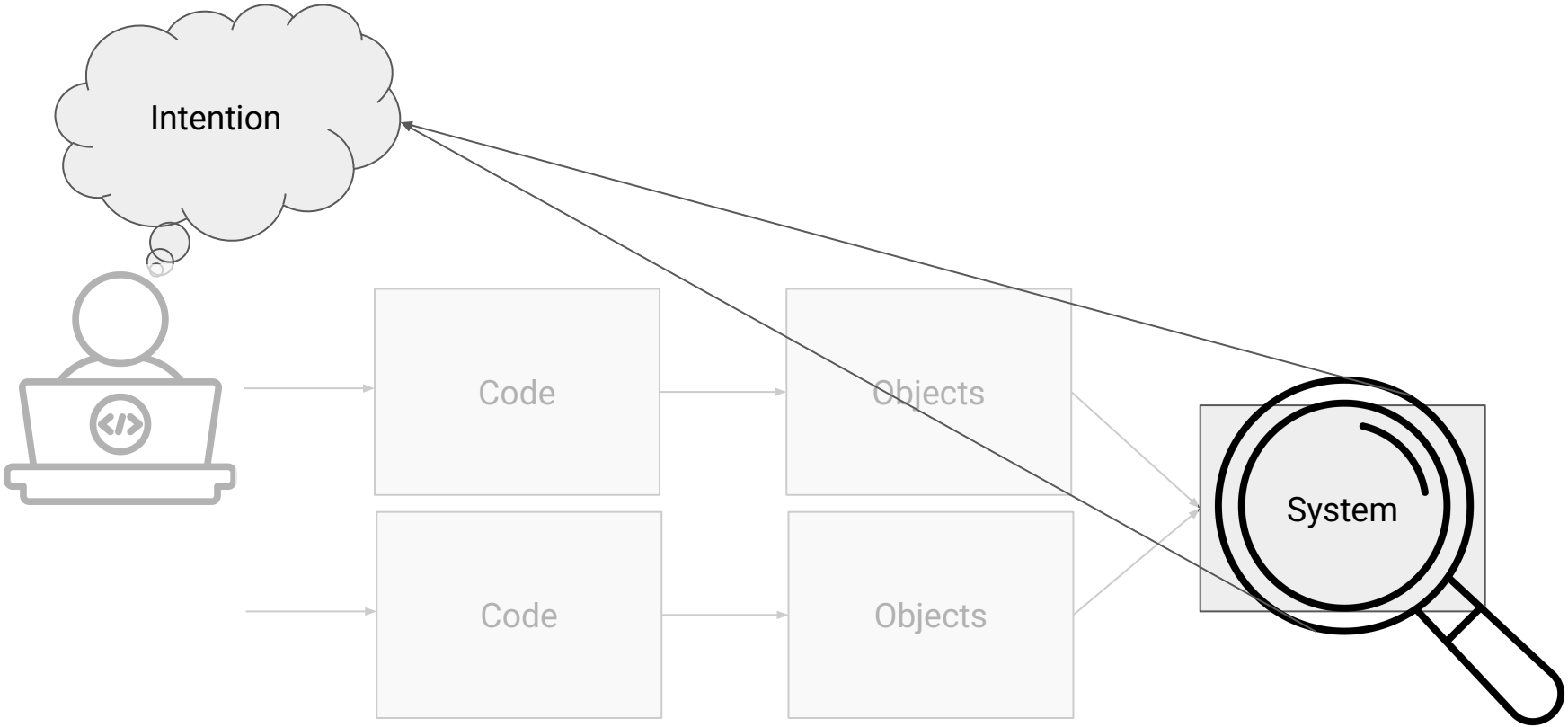




Logic Bugs



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.ed
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 vulnerabilitie
flected in the Symant
Report, which was pu
port states that, in 20
for 63% of the total n

Most recent researc
applications has focu
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
the intended meaning

One reason for the
nerabilities is that it is
general specification
teristics of these vuln
gramming environme
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 ave web. The increasing amount of sensitive data s 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 and from major corporate retailers, by writing SQ 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 respon the HTTP response can be a notification that web browser) should look elsewhere for the r source. In this case, the web application set 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 resou

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.

Why Logic Bugs?

As code moves more and more to memory-safe languages (ala TRACTOR and advanced LLMs), traditional (memory) bugs are not possible.

But logic bugs are!

Example.
In 2025, Ubuntu switched from the C-based coreutils to the rust-based uutils…
… and had them audited, resulting in **44** CVEs in April 202**6**.

But now we're done?

Why Logic Bugs?

As code moves more and more to memory-safe languages (ala TRACTOR and advanced LLMs), traditional (memory) bugs are not possible.

But logic bugs are!

Example.
In 2025, Ubuntu switched from the C-based coreutils to the rust-based uutils…
… and had them audited, resulting in **44** CVEs in April 202**6**.

But now we're done?

We'll write secure Rust!

We'll audit code as we port it!

There's not that much code to port anyways.

LLMs will write secure Rust!

A Rust implementation of Android's Binder

By **Jonathan Corbet**
November 30, 2023

LPC

The Android system was once famous for extensive, out-of-tree kernel enhancements. Many of those have been eliminated or upstreamed over the years, bringing Android much closer to the mainline kernel. One significant component in the "upstreamed" category is Binder, an interprocess communication mechanism that is used only by Android. There are a number of factors that make Binder a good candidate for rewriting in the Rust language; at the 2023 Linux Plumbers Conference, Carlos Llamas and Alice Ryhl described the motivation behind and implementation of a rewrite of Binder in Rust.

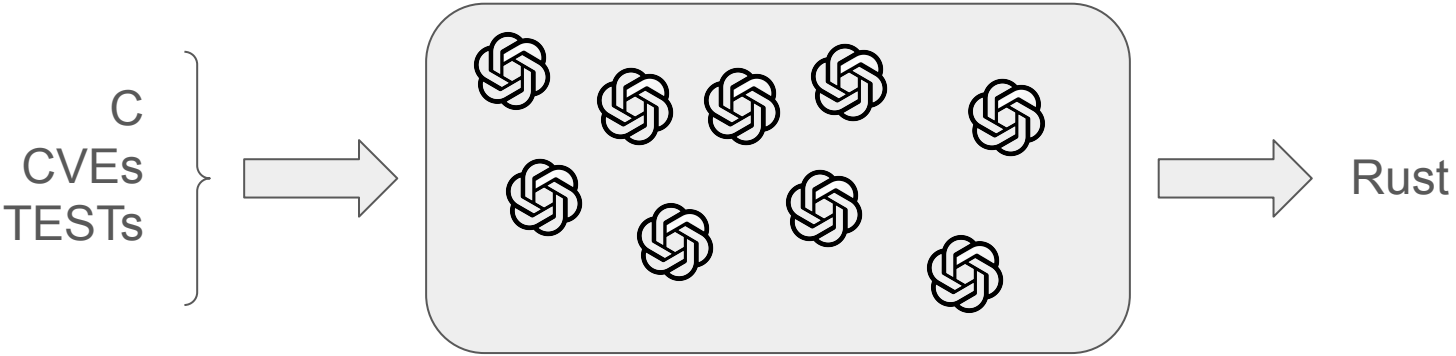
ASU ran an experiment, giving every employee **infinite Codex** usage…

SafeLibs

BETA

SafeLibs builds memory-safe (Rust) reimplementations of critical load-bearing C/C++ libraries used throughout open source infrastructure, while attempting to preserve drop-in compatibility at compile-time and runtime.

SafeLibs: Translating All C (load bearing libraries) To Rust



SafeLibs: Translating All C (load bearing libraries) To Rust

Key Insight

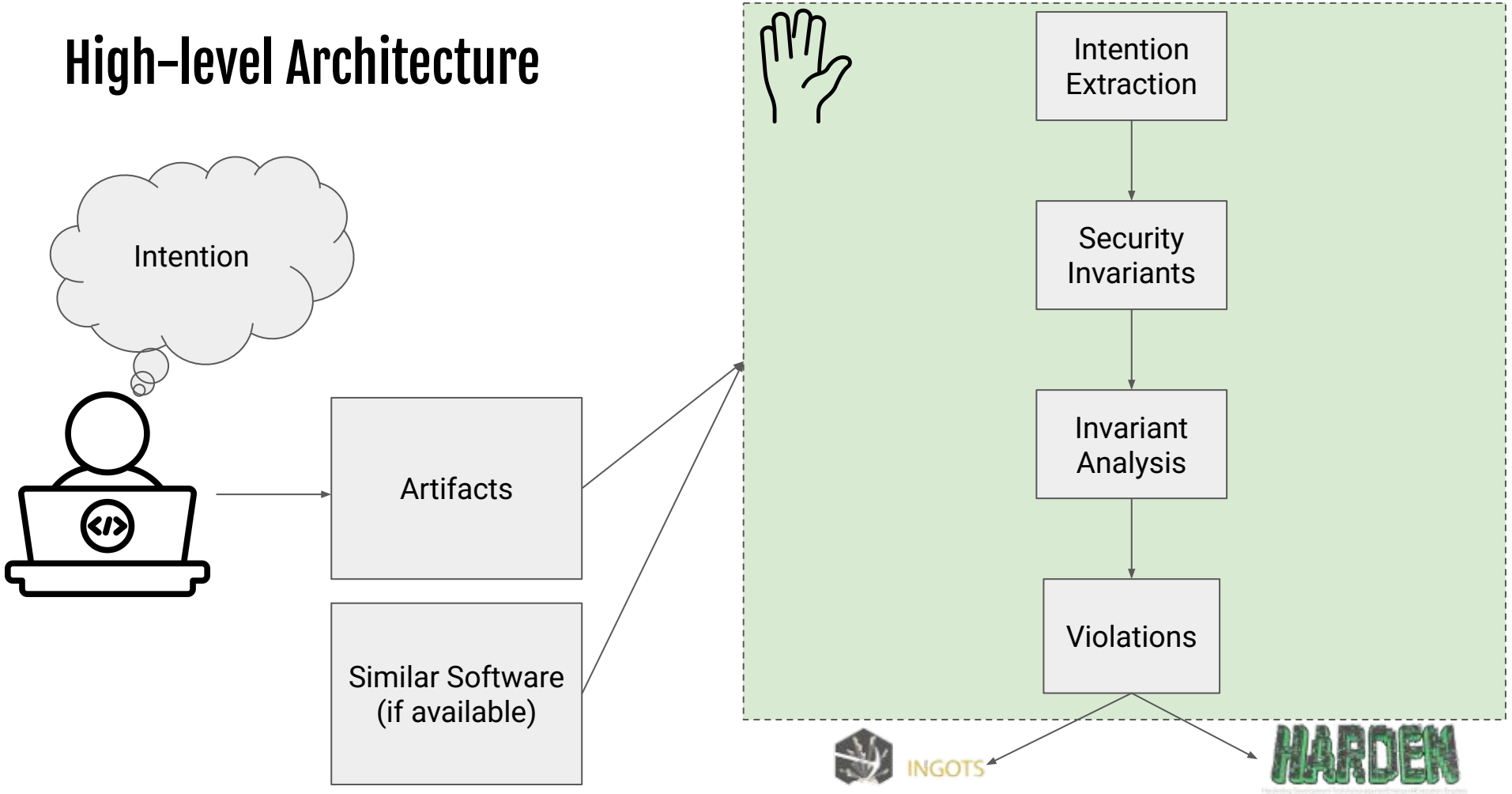
Application context is key.

Similar software shares similar intent.

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

Synthesize invariants that must hold in the system.

High-level Architecture



Test Target

Bluetooth Authorization.

Intent Communication.

Inter-process Communication.

Permission Control.

Unexpected Directory Traversal.

… and more

Ground Truth—Android and Open Harmony

SoK: History Doesn't Repeat Itself, but Android Design-Level Vulnerabilities Rhyme in OpenHarmony

Hongkai Chen[1], Yuqing Yang[2], Chao Wang[3], Arpit Nandi[1], Moritz Schloegel[2], Tiffany Bao[1], Ruoyu Wang[1], Adam Doupé[1], Zhiqiang Lin[3], Yan Shoshitaishvili[1]

[1]*Arizona State University,* [2]*CISPA Helmholtz Center for Information Security,* [3]*The Ohio State University*

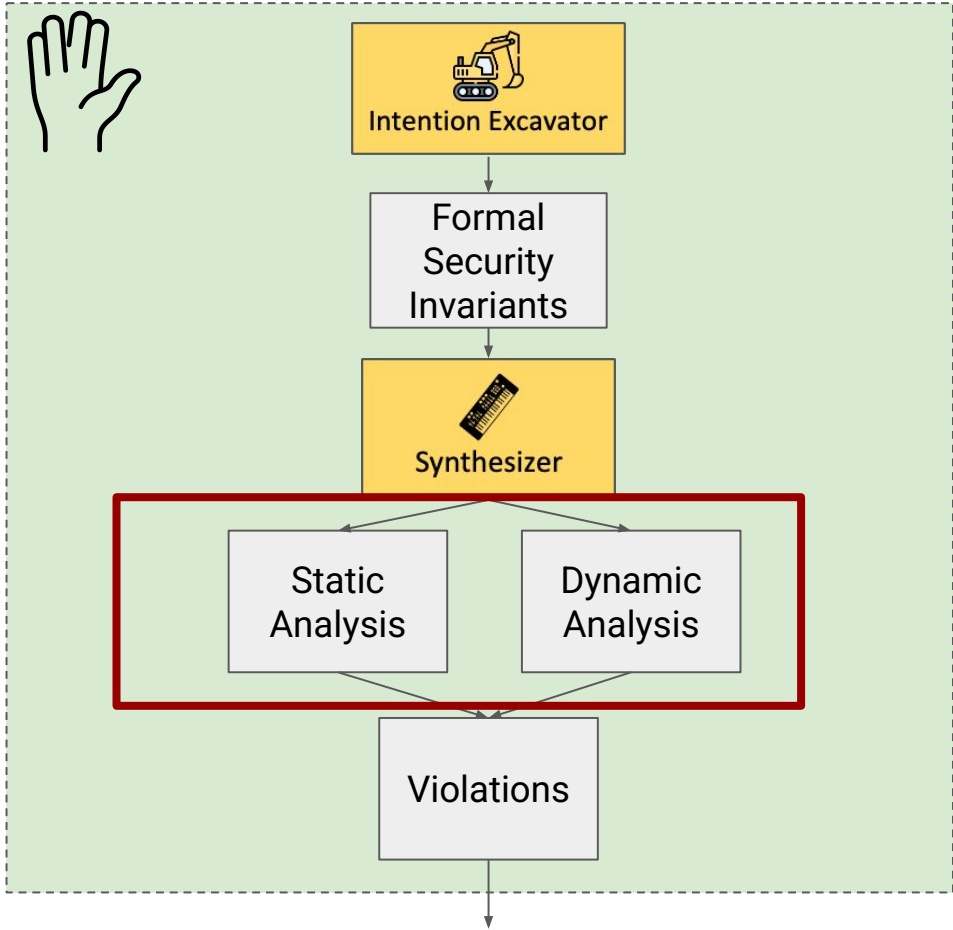
{hongkai.chen, anandi5, tbao, fishw, doupe, yans}@asu.edu, {yuqing.yang, schloegel}@cispa.de, wang.15147@osu.edu, zlin@cse.ohio-state.edu

Abstract

The dominant market share of Android across the world has led to significant scrutiny by security researchers. Over the years, many security issues were identified and remedied not only in its implementation but also in its design. Interestingly, academia has not further systematized these design flaws. This is an important gap, especially as many phone vendors have recently begun developing their own mobile operating

Recently, this fragmentation has evolved to a new stage. In 2019, due to various geopolitical, technological, and commercial factors, some Android vendors began to supersede their customizations of Android with implementations of their own mobile OSes. Many of these OSes are initially still based on Android: for example, Huawei's HarmonyOS up to version 4 was based on Android, but the developers had gradually replaced Android components with their own. Importantly, this

CWE Root	CWE Child	ID	Description of the DLV	Ad	OH	Property C	Property I	Property A	AMT
CWE-284: Improper Access Control	CWE-269: Improper Privilege Management	1	Vulnerable APIs can be exploited for state inference attacks [140]	○	◐	✗			10
		2	JS interfaces in WebView can invoke sensitive system functions [167, 181]	○	●	✗			
		3	Third-party apps can access non-resettable device IDs [121, 143, 158, 184]	○	●	✗			10
		4	Third-party apps can access device's phone number [184]	○	○	✗			6
		5	Third-party apps can access fine location with install-time permission [184]	○	◐	✗			11
		6	Third-party apps can access contacts without user consent [184]	○	○	✗			6
		7	Third-party apps can access SMS without user consent [184]	○	○	✗			6
	CWE-346: Origin Validation Error	8	Custom Tab component has critical security flaws such as cross-context state sharing [23]	○	–	✗			
		9	Exploiting autofill flaws for phishing attacks [17, 25]	○	–	✗			
		10	Deep Links allow malicious apps to hijack sensitive URLs [109, 111, 169]	○	○		✗		12
	CWE-923: Improper Restriction of Communication Channel to Intended Endpoints	11	Unix Domain Sockets can be misused due to weak security mechanisms [41, 157]	○	○	✗	✗		9
		12	Exploiting DICI mechanism for cache side-channel attacks [110]	●	–	✗			
		13	Content provider lacks access control [28, 104, 201]	○	◐	✗			
		14	Malicious apps can hijack unprotected components [191, 201]	○	○		✗		4
	CWE-300: Channel Accessible by Non-Endpoint	15	Apps can leak sensitive data by sending unprotected broadcasts [76]	○	○	✗			
		16	Implicit Intent hijacking [92, 102]	○	●		✗		11
		17	Insecure Intents can be exploited to control camera app [34]	○	○	✗	✗		
		18	Implicit PendingIntents can be hijacked for multiple attacks [32]	○	–		✗		
		19	System broadcasts lack access control, leaking sensitive information [194]	○	●	✗			12
		20	System apps' broadcasts are not protected [33]	○	○	✗			9
	CWE-295: Improper Certificate Validation	21	Improper SSL/TLS validation leads to MITM vulnerabilities [159]	○	○	✗	✗		7
		22	Apps trust any certificate by any trusted Certificate Authority [77, 134, 138]	○	●[1]	✗	✗		7
	CWE-1256: Improper Restriction of Software Interfaces to Hardware Features	23	Apps can exploit motion sensors to perform side-channel attacks [68, 118, 122, 128]	○	●	✗			12
		24	Apps can covertly access microphone in the background [197]	○	●	✗			12
		25	Third-party apps can stealthily pair Bluetooth devices [177]	○	●	✗	✗		10
		26	Third-party apps can stealthily disconnect Bluetooth devices [129]	○	●		✗		9
		27	Third-party apps can stealthily manipulate Bluetooth scan mode [29]	○	●		✗		12
		28	External Bluetooth devices can stealthily connect to the host device [31]	○	●	✗	✗		10
	CWE-552: Files or Directories Accessible to External Parties	29	Apps can exploit system files to bypass permissions and access sensitive data [148, 198]	○	◐	✗			12
		30	Apps can read packet counters to crack sequence number [145]	○	●	✗	✗		10
		31	Apps can read files in /proc to infer secrets [93, 198]	○	◐	✗			10
		32	Apps can read interruption data for inference attacks [69]	○	○	✗			10
		33	Apps can read power data for location tracking [123]	○	●	✗			9
	CWE-749: Exposed Dangerous Method or Function	34	Apps can trigger system crashes and reboots by exploiting unprotected components [156]	○	●			✗	
		35	Apps can exploit unprotected data storing operations to DoS system services [189]	○	○			✗	
		36	Apps can exploit synchronous callbacks to freeze and DoS system services [168]	○	○			✗	6
CWE-221: Information Loss or Omission	CWE-451: UI Misrepresentation of Critical Information	37	Malware can abuse accessibility services to observe and control GUI [78, 89, 94, 178]	○	○	✗	✗		13
		38	Exploiting overlay for clickjacking [139]	○	–		✗		
		39	Exploiting overlay for PHYjacking attacks [170]	○	–		✗		
		40	Exploiting overlay for GUI confusion attacks [37, 38]	○	–		✗		
		41	Exploiting overlay for phishing attacks [196]	○	–	✗			
		42	Apps can launch Activity from the background to hijack UI [44, 163]	○	◐		✗		10
		43	Picture-in-picture (PiP) can be exploited for UI hijacking attacks [27]	○	○		✗		11
		44	UI task stack flaws lead to task hijacking [149]	○	–		✗		
CWE-1038[2]	N/A	45	App process creation weakens ASLR by sharing memory layouts across processes [100]	◐	◐		✗		7
		46	Apps can exploit shared memory side channels to infer UI states [44]	○	○	✗			6
CWE-664: Improper Control of a Resource Through its Lifetime	N/A	47	System does not clean up data residues from uninstalled apps [192]	○	○	✗			6
	CWE-921: Storage of Sensitive Data in a Mechanism without Access Control	48	Apps can leak sensitive data to device logs, which other apps can access [76]	○	○	✗			4
		49	System apps can silently access global device logs [116]	○	●	✗			13
		50	Apps can access and modify other apps' data in external storage [76]	○	○	✗	✗		10
		51	Race conditions in external storage [72]	○	–		✗		
		52	Apps can covertly store identifiers in media files by appending bytes to images [71]	●	○	✗			


Intention Excavator
Formal Security Invariants
Synthesizer
Static Analysis
Dynamic Analysis
Violations

Can LLMs find Logic Bugs?

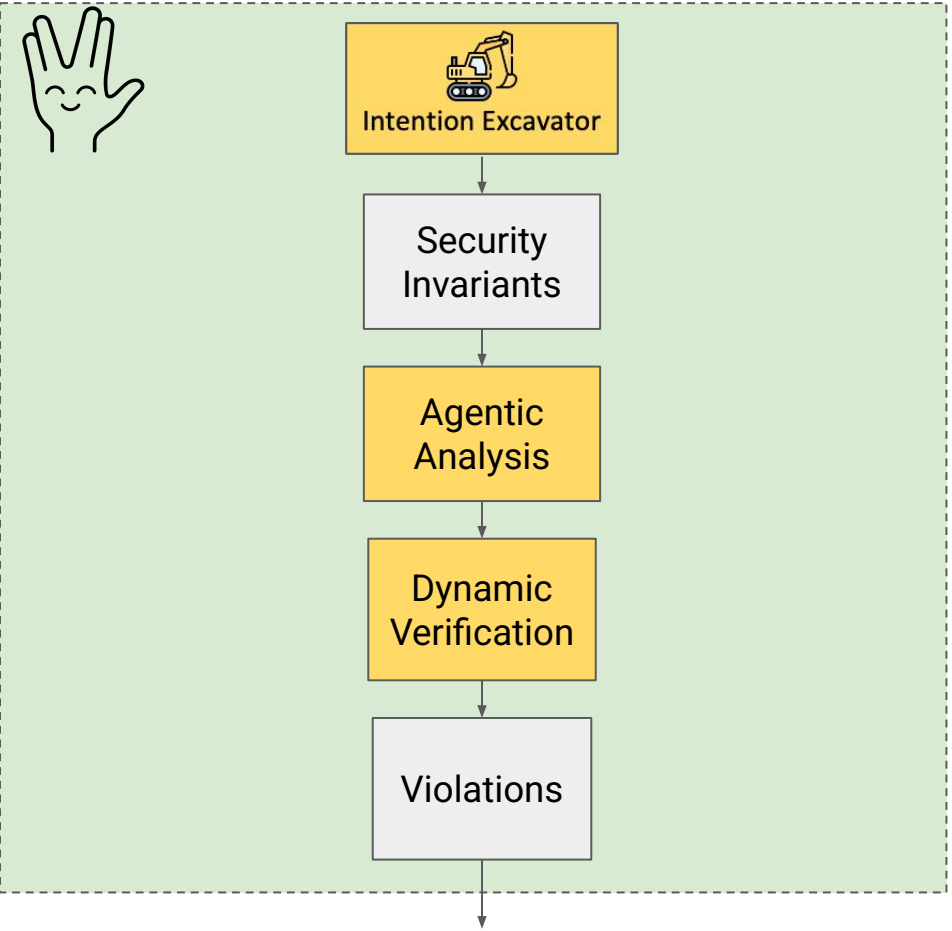
CVE-2026-31431 **100% reliable** every distro since 2017 container escape primitive 732 bytes

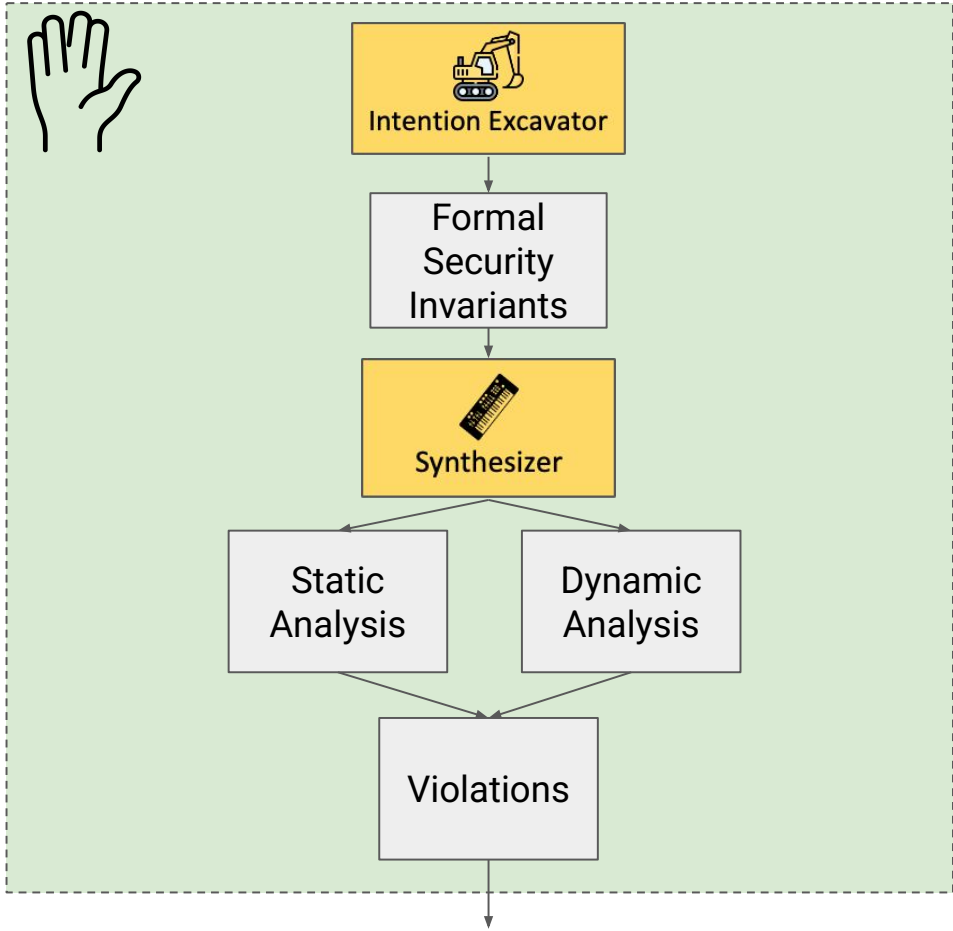
found by Xint Code

Copy Fail

Most Linux LPEs need a race window or a kernel-specific offset.
Copy Fail is a **straight-line logic flaw** — it needs neither.
The same **732-byte** Python script roots every Linux distribution shipped since 2017.

One logic bug in `authencesn`, chained through `AF_ALG` and `splice()` into a


Intention Excavator
Security Invariants
Agentic Analysis
Dynamic Verification
Violations


Intention Excavator
Formal Security Invariants
Synthesizer
Static Analysis
Dynamic Analysis
Violations

Remember SafeLibs?

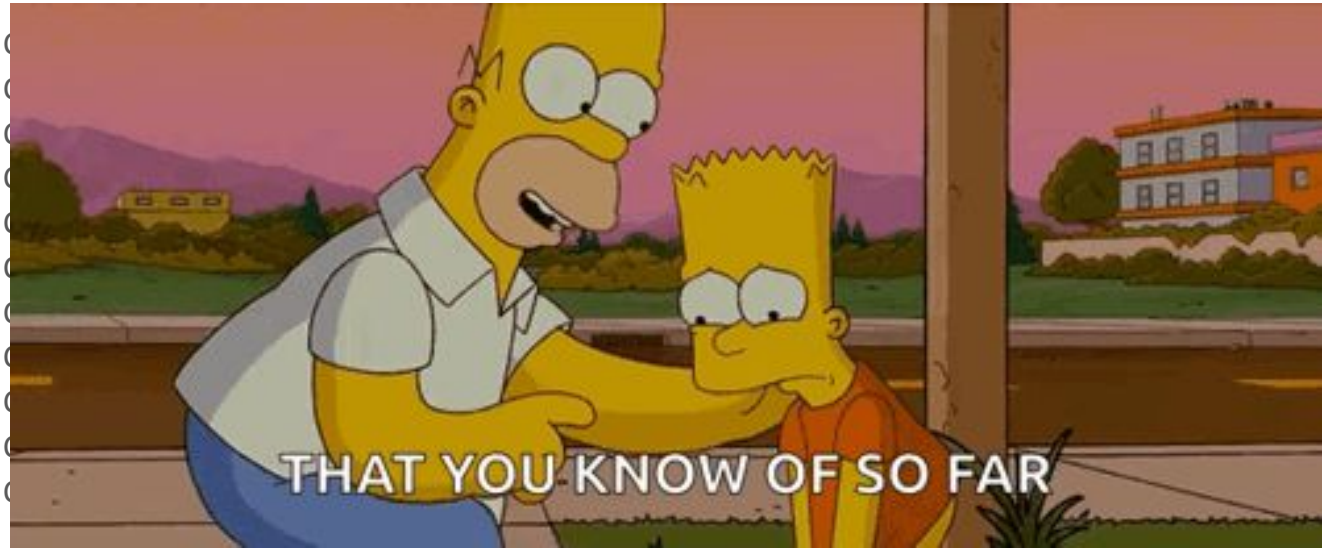
Rust reimplementation re-introduced several logic bugs!

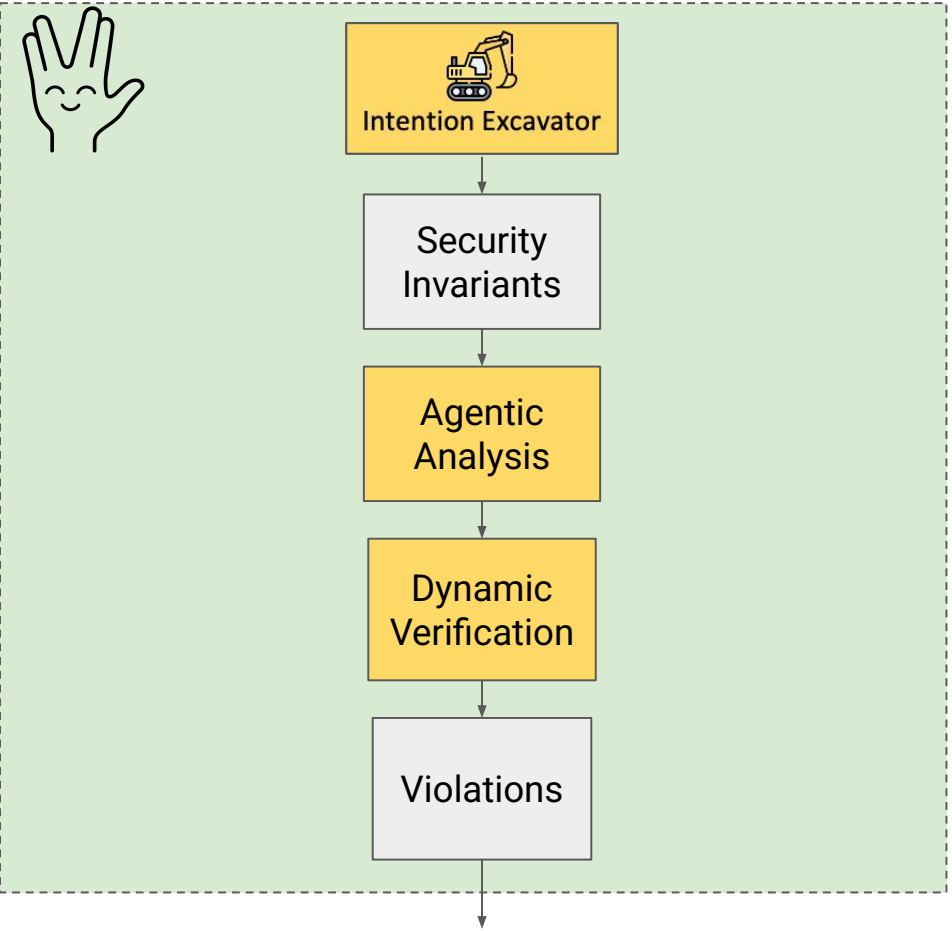
```
libgcrypt — CVE-2024-2236  — RSA PKCS#1 v1.5 Bleichenbacher oracle
libgcrypt — CVE-2018-6829  — ElGamal m=0 leaks b=0
libxml    — CVE-2024-40896 — XInclude file:// XXE under default parser flags
libtiff   — CVE-2023-6277  — StripByteCounts vec![0u8; N] aborts on OOM
libtiff   — CVE-2017-12944 — XMLPacket count vec![0u8; N] aborts on OOM
libtiff   — CVE-2018-5784  — StripOffsets count vec![0u8; N] aborts on OOM
libvips   — CVE-2026-3146  — Matrix loader Vec::with_capacity(2^60) aborts
libpng    — CVE-2014-0333  — Progressive reader never fires end_cb on zero-length IDAT
libjson   — CVE-2013-6371  — PERLLIKE hash collisions → 260× DoS (opt-in)
libc6     — CVE-2010-4051  — regcomp bounded-repetition unvalidated → OOM
libc6     — CVE-2010-4756  — glob unmatched-pattern exponential walk
```

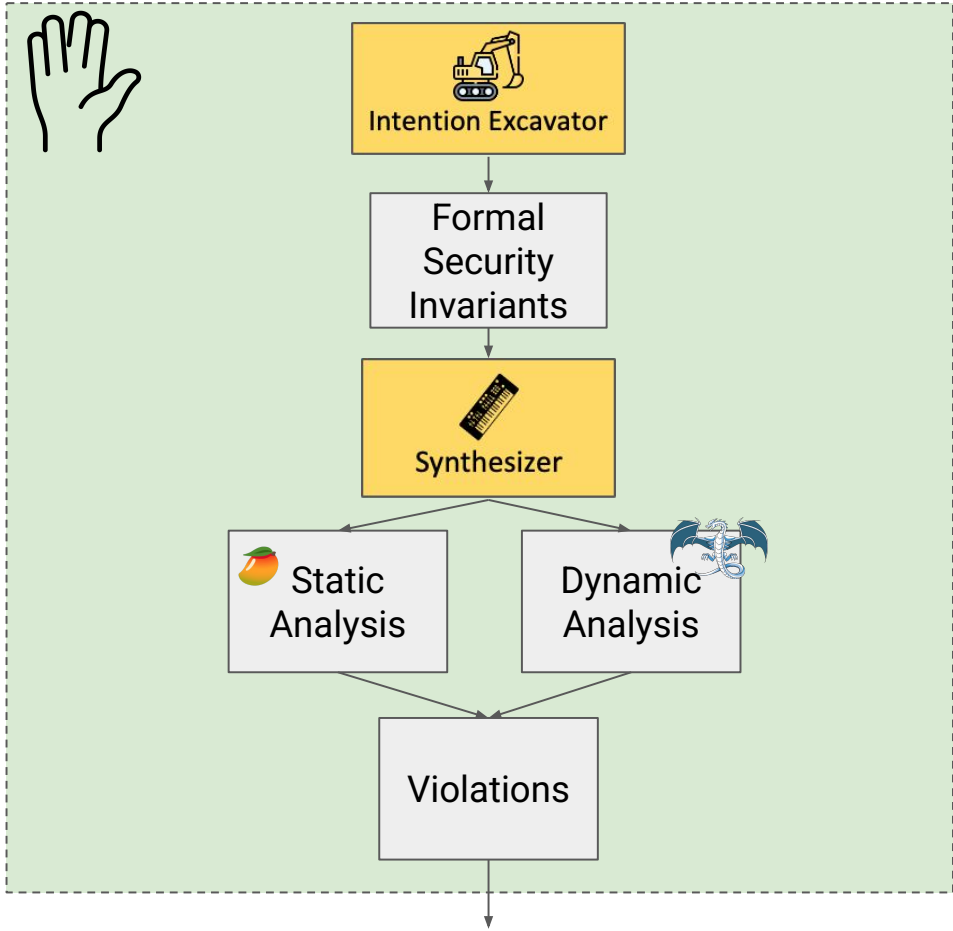
Remember SafeLibs?

Rust reimplementation re-introduced several logic bugs!

```
libgcrypt – C
libgcrypt – C
libxml    – C
libtiff   – C
libtiff   – C
libtiff   – C
libvips   – C
libpng    – C
libjson   – C
libc6     – C
libc6     – C
```




Intention Excavator
Security Invariants
Agentic Analysis
Dynamic Verification
Violations


Intention Excavator
Formal Security Invariants
Synthesizer
Static Analysis
Dynamic Analysis
Violations

OpenHarmony Results

Of the 56 DLVs that were human analyzed, Intention extractor correctly extracted 16 invariants.

9 verified via dynamic analysis.

2 verified via static analysis.

DEMO

End-to-end independent discovery of a known DLV in OpenHarmony.

Input is the OpenHarmony module.

Output is invariant, and who to trigger a violation of that invariant.

File to Reboot?

Extracted Invariant from OpenHarmony source code:

```
Every caller of musb_regdump_show must either:
   have DEBUGFS_ROOT permission
   or be a system component.
```

This invariant was synthesized into a runtime check into `musb_regdump_show`.

Fuzzing found an input that triggered the invariant.

Lessons Learned

LLMs are capable of extracting invariants from code.

Not every logic bug is exploitable… yet copy.fail shows that they might.

As LLMs get better at implementing and reimplementing systems, they clearly are introducing logic bugs.

Final Priorities

Extending evaluation to other parts of OpenHarmony (prior focus on DLVs).

Integration in UTE (confirmed that current UTE is flexible enough to support most invariants).

Apply system to Android (easy lift, nothing is Open Harmony specific).

Live Long and Prosper... Free of Logic Bugs