

A few notes on hacker math

“Fundamental research challenges hidden in plain sight”

Sergey's disclaimers



- Obligatory: **Any opinions are mine alone and don't represent any of my employers past or present.**
- Substantive:
 - This is a personal perspective on other people's amazing work. All credit goes to them, not me.
 - This is a tiny, biased sample of a great domain. Please tell me what I am missing!
- Trivial: I am a former mathematician. I tend to see math everywhere :)

Hard hacking problems mean math

“Math pwns”

Hypothesis:

Biggest advances in hacking/cybersecurity come from nifty machine-readable mathematical representations of data & code— which are friendly to efficient algorithms.

Right representation => Math => Algorithm => Tool => Pwnage

Binary Diffing = Graph Isomorphism

Halvar Flake: BinDiff

Representation => algorithms => pwnage

- Diffing binaries is useful (e.g., for patches) but hard.
 - Heuristics work, but only up to a point
- **Insight:** Graph isomorphism for basic block graphs!
 - Basic blocks make graphs, matching graphs (“graph isomorphism”) is a hard algorithmic problem, but has efficient subcases
- Cf. Joxean Koret’s *Diaphora* (<https://github.com/joxeankoret/diaphora>); QuarksLab’s Diffing portal (<https://diffing.quarkslab.com/>)
- Also: Trail of Bits’ *PolyFile* and *GraphTage* for diffing and merging arbitrary binary formats (<https://blog.trailofbits.com/2020/08/28/graphtage/>)

Decompilation: graph structuring

Cristina Cifuentes, Mother of decompilation

- Going back from compiled binary is hard, heuristics only get so far

- **Insight:**

Reverse Compilation Techniques

by

Cristina Cifuentes

Bc.App.Sc – Computing Honours, QUT (1990)

Bc.AppSc – Computing, QUT (1989)

Submitted to the School of Computing Science
in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

| | | |
|-------|---|----|
| 1 | Introduction to Decompiling | 1 |
| 1.1 | Decompilers | 1 |
| 1.2 | Problems | 1 |
| 1.2.1 | Recursive Undecidability | 2 |
| 1.2.2 | The von Neumann Architecture | 3 |
| 1.2.3 | Self-modifying code | 3 |
| 1.2.4 | Idioms | 3 |
| 1.2.5 | Virus and Trojan “tricks” | 4 |
| 1.2.6 | Architecture-dependent Restrictions | 6 |
| 1.2.7 | Subroutines included by the compiler and linker | 6 |
| 1.3 | The Phases of a Decompiler | 7 |
| 1.3.1 | Syntax Analysis | 8 |
| 1.3.2 | Semantic Analysis | 9 |
| 1.3.3 | Intermediate Code Generation | 10 |
| 1.3.4 | Control Flow Graph Generation | 10 |
| 1.3.5 | Data Flow Analysis | 10 |
| 1.3.6 | Control Flow Analysis | 11 |
| 1.3.7 | Code Generation | 11 |
| 1.4 | The Grouping of Phases | 12 |
| 1.5 | The Context of a Decompiler | 13 |
| 1.6 | Uses of Decompilation | 15 |
| 1.6.1 | Legal Aspects | 15 |

Decompilation, 30 years later

Still making progress: Hex-Rays, Binary Ninja, Ghidra, FoxDec, ...

**Ahoy SAILR! There is No Need to DREAM of C:
A Compiler-Aware Structuring Algorithm for Binary Decompilation**

Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O’Kain, Derron Miao,
Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, Ruoyu Wang

Arizona State University

{zbasque,atipriya,wfgibbs,judeo,derronm,tbao,doupe,yans,fishw}@asu.edu

In contrast, SAILR does not blindly eliminate gotos and instead **treats the cause: compiler transformations.** SAILR precisely reverts compiler transformations found to be the cause of unstructurable code, which manifest as gotos in decompilation. Of these transformations, certain compiler optimizations and the gap between the decompiler and the compiler play a significant role in unstructurability. SAILR approaches a solution to both of these problems by improving the knowledge of the decompiler and reverting certain optimizations.

- More: <https://mahaloz.re/dec-history-pt2>

Reverse Engineering ~ Math

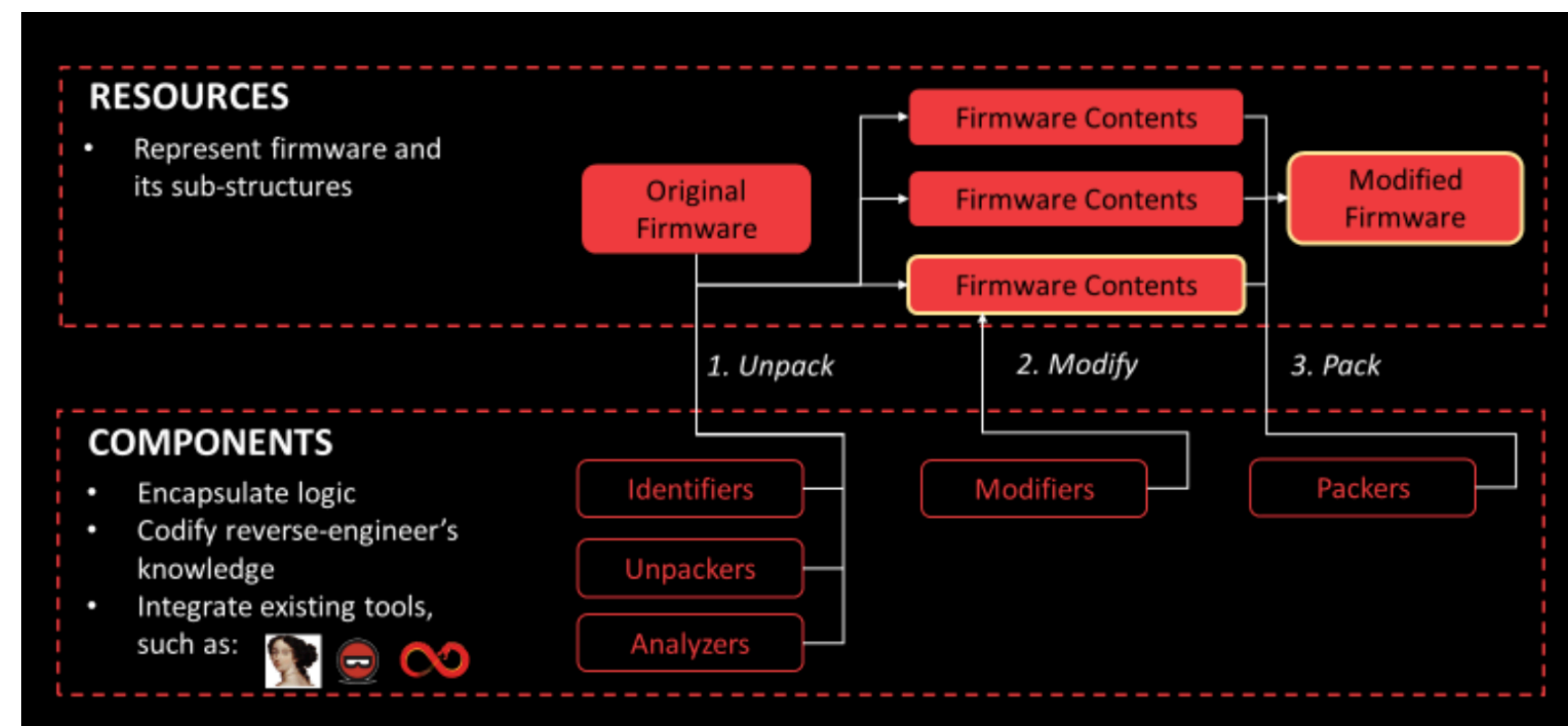
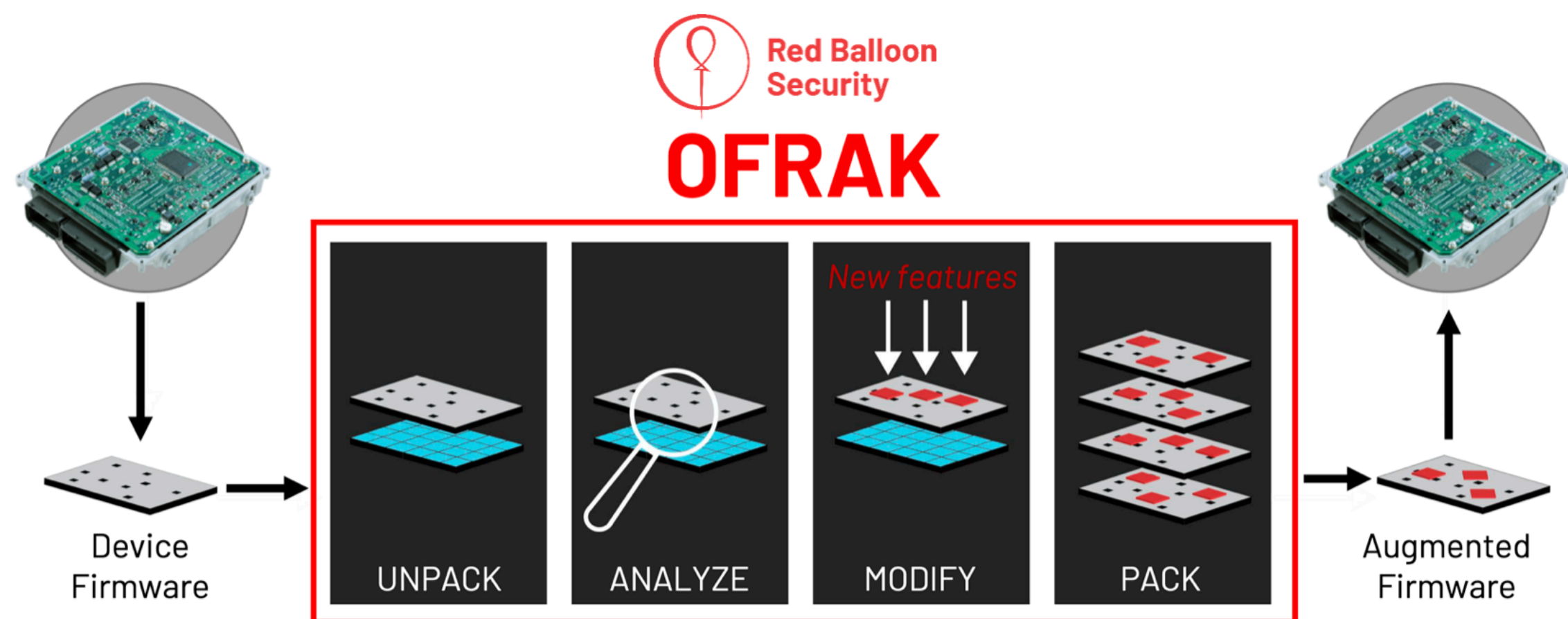
Halvar Flake's "RE 2006: New Challenges Need Changing Tools"

- #1 and #2: Automated data structure recovery; building UML inheritance diagrams from binaries.
- Coupling the above with a debugger to allow run-time object inspection and editing.
- #3: Automated modularization of binaries (decomposing binaries to recover library structure / groupings).
- #4: De-templating of heavily templated C++ code.
- #7: "Normal forms" for sequences of code (a Groebner-base equivalent?)
- #8: A visualization for callgraphs that shows each node as a Poset to make sure the order of outgoing edges is visualized, too.
- 9#: Recovery of the internal state machine of a target.
- 10#: Semantics-based FLIRT-style library identification.

Interestingly, challenge #5 - automated input data creation - is the one where most progress has happened since the talk. To my great amusement, this talk suggests the use of SAT solvers to do it. At that time, I was obviously unaware at the time of the research on SMT that is happening and will lead to Vijay Ganesh's great 2007 thesis (and the release of STP).

Modular framework for decompilation research

A new generation of tools for maintaining binaries



<https://github.com/redballoonsecurity/ofrak>

Modular framework for binary research

A new generation of tools for maintaining binaries

- CodeCut: <https://github.com/JHUAPL/CodeCut>
- Codehawk: <https://github.com/static-analysis-engineering/codehawk>
- VIBES: <https://github.com/draperlaboratory/VIBES>
- Remill, Anvil, Relic LLVM lifters: <https://github.com/lifting-bits/remill>, <https://github.com/lifting-bits/anvill>, <https://github.com/lifting-bits/relic>
- PATE binary patch verifier: <https://github.com/GaloisInc/pate>
- MCTrace code release: <https://github.com/GaloisInc/mctrace>
- Binary analysis and rewriting tools used by PATE and MCTrace: <https://github.com/GaloisInc/macaw>, reopt, what4, crucible, elf-edit, renovate, etc.

“Weaponizing the Chomsky syntax hierarchy

Kaminsky/Sassaman/Patterson: Breaking X.509 => LangSec

- Why so many “input validation/sanitization” bugs in everything?
 - What are programmers doing that they can’t ever get right?
- **Insight:** Inputs have grammars. Complex grammars are hard to parse. Ambiguous grammars are impossible to validate. Sanitization is an anti-pattern.
 - Many bugs uncovered, hardened/correct parsers built



Safe Documents: Safely intake electronic data by creating tools to build machine-readable unambiguous format definitions and secure verified parsers

“Recognizer doesn't match the input language”

Regex match open tags except XHTML self-contained tags

Asked 12 years, 4 months ago Modified 4 months ago Viewed 3.4m times



You can't parse [X]HTML with regex. Because HTML can't be parsed by regex. Regex is not a tool that can be used to correctly parse HTML. As I have answered in HTML-and-regex questions here so many times before, the use of regex will not allow you to consume HTML. Regular expressions are a tool that is insufficiently sophisticated to understand the constructs employed by HTML. HTML is not a regular language and hence cannot be parsed by regular expressions. Regex queries are not equipped to break down HTML into its meaningful parts. so many times but it is not getting to me. Even enhanced irregular regular expressions as used by Perl are not up to the task of parsing HTML. You will never make me crack. HTML is a language of sufficient complexity that it cannot be parsed by regular expressions. Even Jon Skeet cannot parse HTML using regular expressions. Every time you attempt to parse HTML with regular expressions, the unholy child weeps the blood of virgins, and Russian hackers pwn your webapp. Parsing HTML with regex summons tainted souls into the realm of the living. HTML and regex go together like love, marriage, and ritual infanticide. The <center> cannot hold it is too late.

lord help us how can anyone survive this scourge using regex to parse HTML has doomed humanity to an eternity of dread torture and security holes using regex as a tool to process HTML establishes a breach between this world and the dread realm of corrupt entities (like SGML entities, but more corrupt) a mere glimpse of the world of regex parsers for HTML will instantly transport a programmer's consciousness into a world of ceaseless screaming, he comes, the pestilent slithy regex-infection will devour your HTML parser, application and existence for all time like Visual Basic only worse he comes he comes do not fight he comes, his unholy radiance destroying all enlightenment, HTML tags leaking from your eyes like liquid pain, the song of regular expression parsing will extinguish the voices of mortal man from the sphere I can see it can you see it it is beautiful the final snuffing of the lies of Man ALL IS LOST ALL IS LOST the pony he comes he comes he comes the ichor permeates all MY FACE MY FACE oh god no NO NOOOO NO stop the angles are not real ZALGO IS TONY THE PONY HE COMES

A LangSec view of data languages

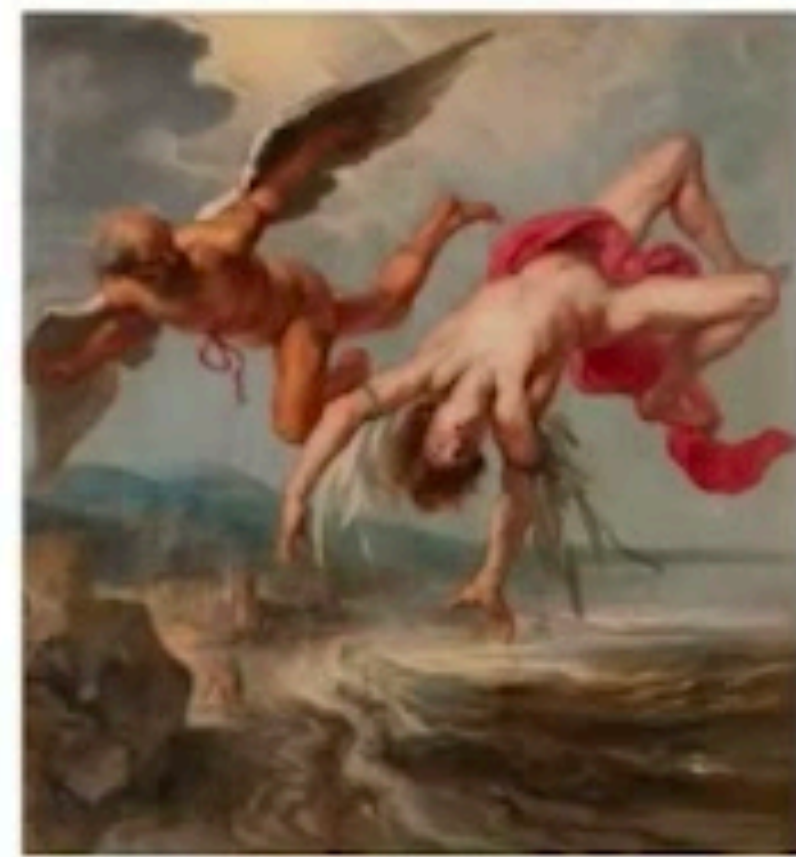
| Complexity class | Examples | Model needed to correctly parse/accept | Complexity to <u>verify</u> implementation | Security |
|--------------------------------------|---|--|--|--------------------|
| Recursively enumerable | Javascript, Flash, "benign programs"* | Turing machine | In general, impossible | Gift to attackers |
| Recursive | Some limited programs | Always-stopping Turing machines | In general, impossible | |
| Context-sensitive | Document & image formats, PDF, MPEG, DNS, SMB, ASN.1, X.509, actual XML | Linear-bounded automata (random-access memory) | Likely not safe or securable in general | |
| Mildly context-sensitive | Subsets of document & protocol formats | Embedded pushdown automata "stack of stacks" | "Research needed" | |
| Context-free | HTML*, JSON*, XML* | Pushdown automata "stacks" | "Feasible, with challenges" | |
| CALC-regular "regular+length fields" | Many TCP/IP* protocols | Finite state machines with accumulators | Feasible | Safe if done right |
| Regular | IPv4 | Finite state machines | Known & efficient | |

A LangSec view of data languages

| Complexity class | Examples | Model needed to correctly parse/accept | Complexity to <u>verify</u> implementation | Security |
|--------------------------------------|---|--|--|--------------------|
| Recursively enumerable | Javascript, Flash, "benign programs"* | Turing machine | In general, impossible | Gift to attackers |
| Recursive | Some limited programs | Always-stopping Turing machines | In general, impossible | |
| Context-sensitive | Document & image formats, PDF, MPEG, DNS, SMB, ASN.1, X.509, actual XML | Linear-bounded automata (random-access memory) | Likely not safe or securable in general | |
| Mildly context-sensitive | Subsets of document & protocol formats | Embedded pushdown automata "stack of stacks" | "Research needed" | |
| Context-free | HTML*, JSON*, XML* | Pushdown automata "stacks" | "Feasible, with challenges" | |
| CALC-regular "regular+length fields" | Many TCP/IP* protocols | Finite state machines with accumulators | Feasible | Safe if done right |
| Regular | IPv4 | Finite state machines | Known & efficient | |

Good news: this work has started!

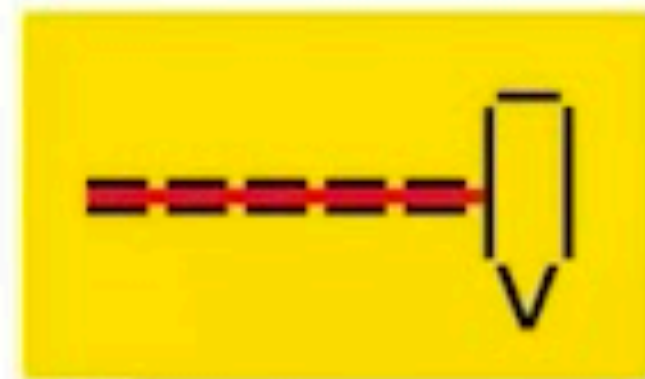
Data Definition Languages (DDLs):



DaeDaLus (Galois, Inc.)

Parsley

Parsley DDL (SRI)



Parser combinators for binary formats, in C. Yes, in C. What? Don't look at me like that.

Hammer/VALARIN
(Special Circumstances, LLC/BAE)



Arlington PDF Model
(PDF Association)

Image credits:
[1] <https://en.wikipedia.org/wiki/Icarus#/media/File:Gowy-icaro-prado.jpg>
[2] Natarajan Shankar, SRI
[3] Meredith L. Patterson, Special Circumstances LLC
[4] PDF Association

Microsoft Research Summit
2021



pdf-association/
safedocs



Artifacts from the DARPA-funded SafeDocs
research program



- “**Demystifying PDF through a machine-readable definition,**” Peter Wyatt, CTO of PDF Association
- “**Building a File Observatory for Secure Parser Development,**” Tim Allison et al., **NASA Jet Propulsion Lab**
- “**Accessible Formal Methods for Verified Parser Development,**” Letitia Li et al., BAE Systems
- “**RL-GRIT: Reinforcement Learning for Grammar Inference,**” Walt Woods et al., Galois Inc.

A few resources by



- First ever **machine-readable object model for PDF** 1.6 through 2.0
500+ objects, 3,500+ keys, 5000 rules, 40 relationship predicate types
- Exposed multiple bugs in existing validators and parsers, 600+ deviations
- 100+ disambiguating candidate edits proposed and adopted into ISO PDF 2.0 standard (32000-2:2020, 1000 pages, 79 normative references)

Map of PDFs in the July 2020 Common Crawl Data --
GeoLocation of URL/IPs via MaxMind's GeoIP City Database



© 2021 California Institute of Technology. Government sponsorship acknowledged.

jpl.nasa.gov

- First ever Internet-scale observatory for a major document format, global coverage
- Based on Apache CommonCrawl, AWS
- Automatic identification of malformations, with attribution and estimated impact
- <https://digitalcorpora.org/corpora/file-corpora/cc-main-2021-31-pdf-untruncated/> (sponsored by AWS)

Exploitation = proofs + programming

Exploits are proofs, exploitation is verification

The idea is to identify security-critical software bugs so they can be fixed first.

BY THANASSIS AVGERINOS, SANG KIL CHA, ALEXANDRE REBERT, EDWARD J. SCHWARTZ, MAVERICK WOO, AND DAVID BRUMLEY

Automatic Exploit Generation

ATTACKERS COMMONLY EXPLOIT buggy programs to break into computers. Security-critical bugs pave the way for attackers to install trojans, propagate worms, and use victim computers to send spam and launch denial-of-service attacks. A direct way, therefore, to make computers more secure is to find security-critical bugs before they are exploited by attackers.

How would you go about finding the unknown exploitable ones that still lurk?

Given a program, the automatic exploit generation (AEG) research challenge is to both automatically find bugs and generate working exploits. The generated exploits unambiguously demonstrate a bug is security-critical. Successful AEG solutions provide concrete, actionable information to help developers decide which bugs to fix first.

Our research team and others cast AEG as a program-verification task but with a twist (see the sidebar “History of AEG”). Traditional verification takes a program and a specification of safety as inputs and verifies the program satisfies the safety specification. The twist is we replace typical safety properties with an “exploitability” property, and the “verification” process becomes one of finding a program path where the exploitability property holds. Casting AEG in a verification framework ensures AEG techniques are based on a firm theoretic foundation. The verification-based approach guarantees sound analysis, and automatically generating an exploit provides proof that the reported bug is security-critical.

Verification involves many well-known scalability challenges, several of which are exacerbated in AEG. Each

Our research team and others cast AEG as a program-verification task but with a twist (see the sidebar “History of AEG”). Traditional verification takes a program and a specification of safety as inputs and verifies the program satisfies the safety specification. The twist is we replace typical safety properties with an “exploitability” property, and the “verification” process becomes one of finding a program path where the exploitability property holds. Casting AEG in a verification framework ensures AEG techniques are based on a firm theoretic foundation. The verification-based

“Verification .. becomes finding a program path [with] the **exploitability property**”

Automated Exploitation Grand Challenge

Julien Vanegue's Challenge Problems

The Automated Exploitation Grand Challenge

Tales of Weird Machines

Julien Vanegue

julien.vanegue@gmail.com

H2HC conference, Sao Paulo, Brazil

October 2013

A Program for Automated Exploitation

- Inspired by David Hilbert and many ones after him, we define a list of problems whose solutions pave the way for years to come in the realm of automated low-level software analysis.
- The Grand Challenge consists of a set of 11 problems in the area of vulnerability discovery and exploitation that vary in scope and applicability.
- Most problems relate to discovering and combining exploit primitives to achieve elevation of privilege.

https://openwall.info/wiki/media/people/jvanegue/files/aegc_vanegue.pdf

What are 'exploitability properties'?

Hypothesis: primitives add up to generic programmability

https://seclists.org/bugtraq/2000/Nov/32

[Home page logo](#)

Site Search

[Bugtraq](#) mailing list archives

◀ [By Date](#) ▶ ▶ [By Thread](#) ▶

List Archive Search

Re: Future of buffer overflows ?

From: Gerardo Richarte <core.lists.bugtraq () CORE-SDI COM>
Date: Mon, 30 Oct 2000 21:53:26 -0300

Thomas Dullien wrote:

So it is possible to have readable, non-executable memory pages, at a not too bad performance hit of up to 10%. This is very cool.

This is not a new concept. It's been out there for a while now...

Does this mean buffer overflows and format string vulnerabilities are dead ?

I'll hung from here.

As you said, this is not true, it's just a little trickier to exploit a "return address on the stack" buffer overflow bug. You showed us a way to do it (by returning to exec() with "pre pushed" arguments)

here I'll show two different approaches to exploit this bugs in "protected" systems.

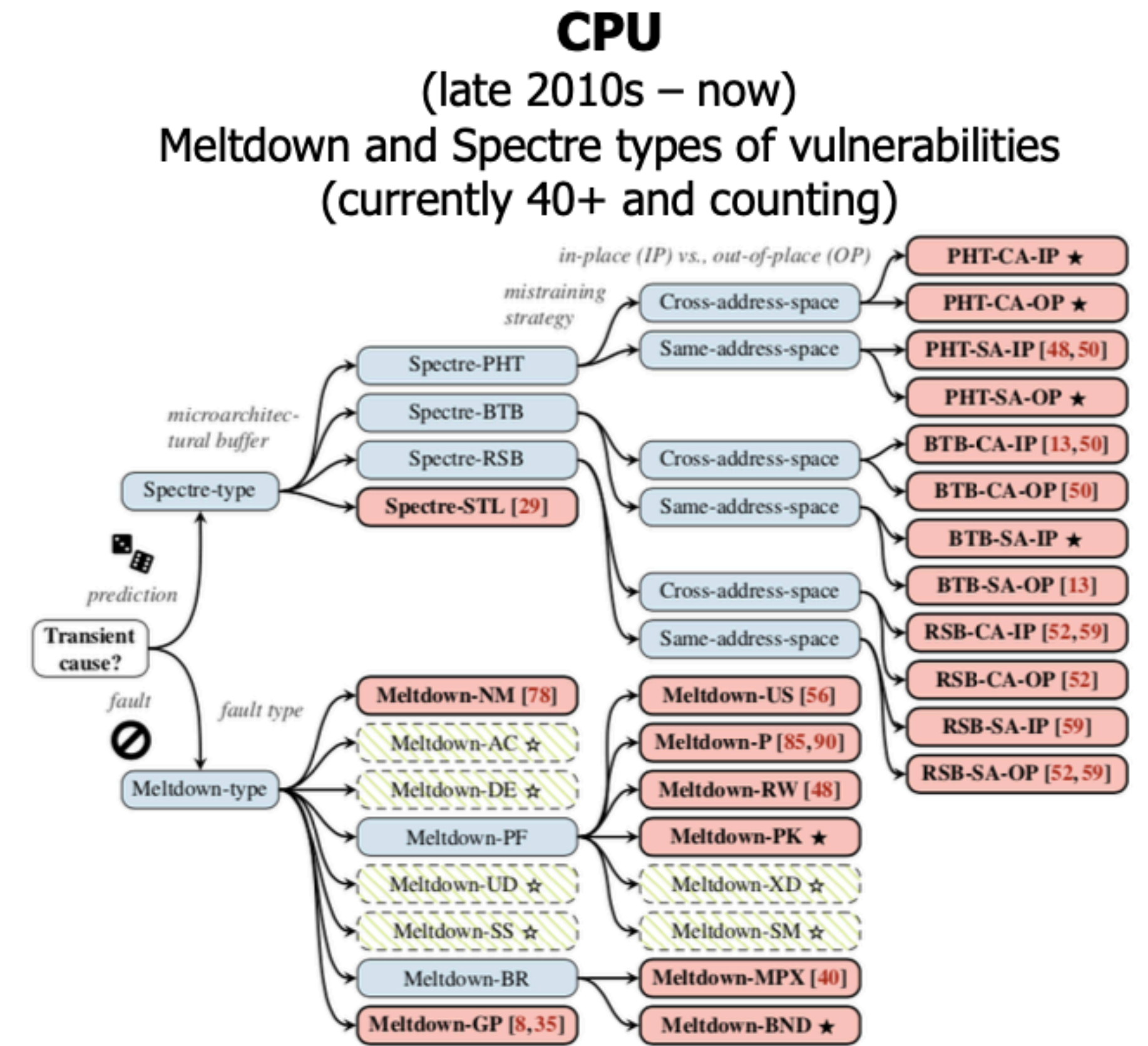
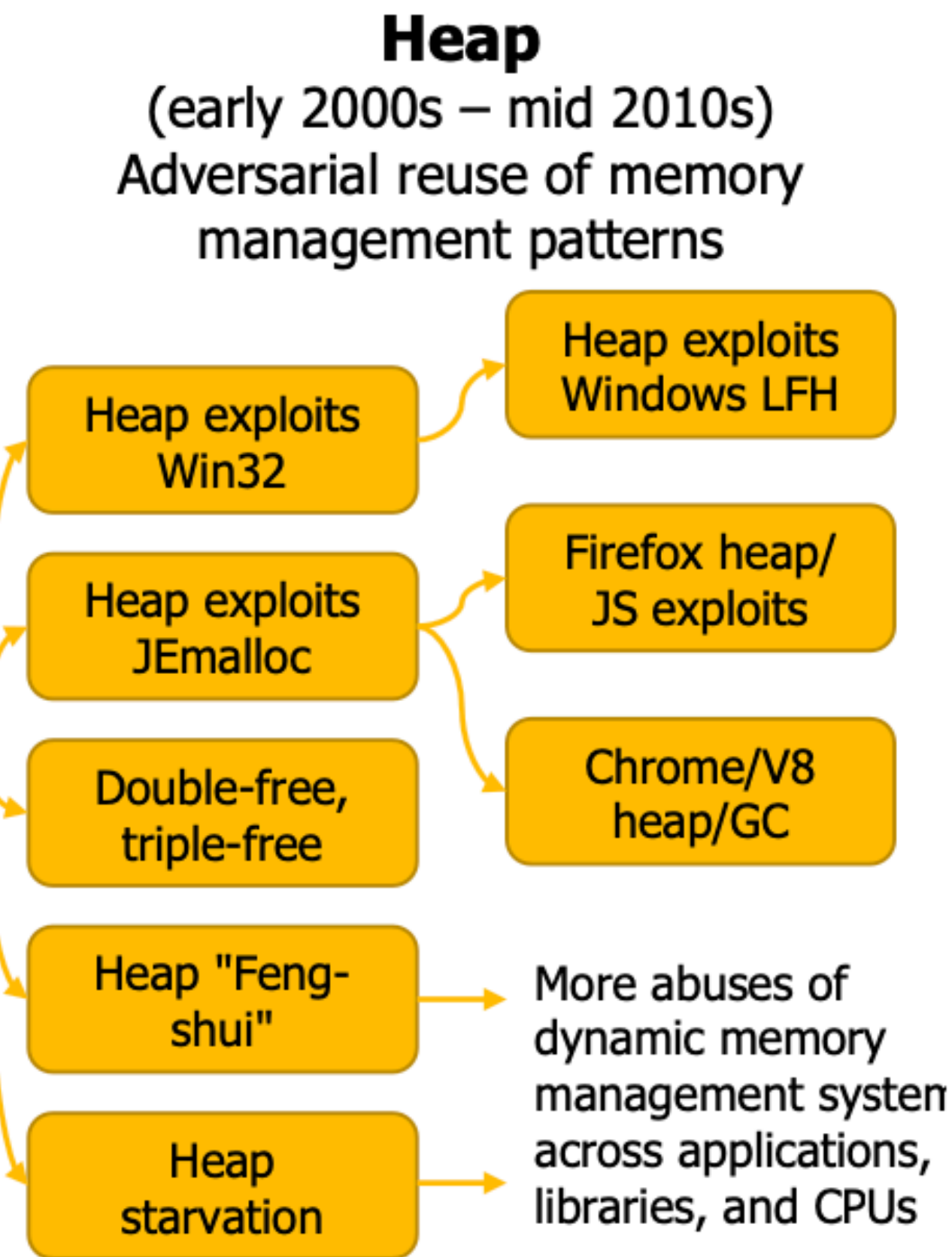
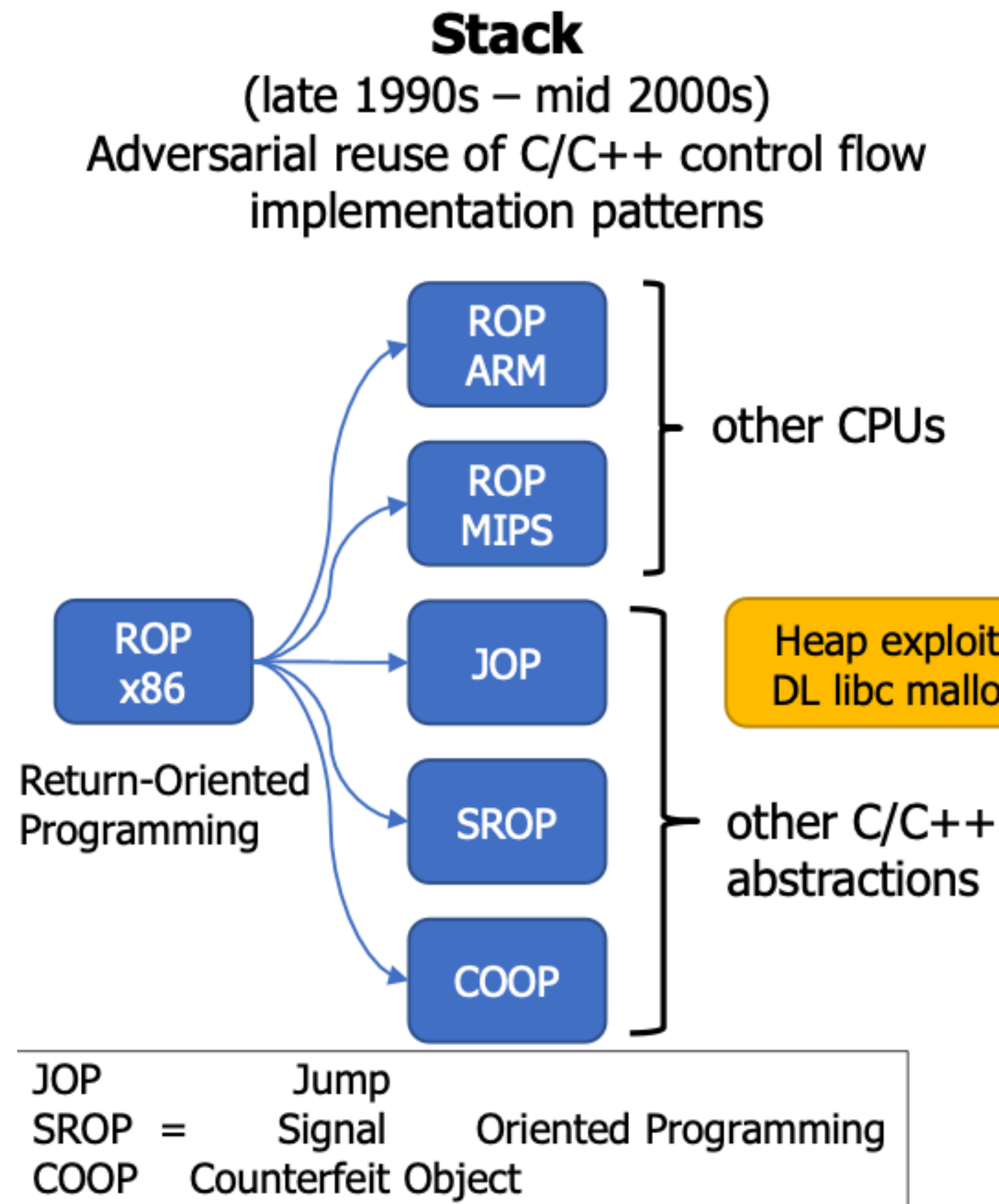
Here I present a way to code any program, or almost any program, in a way such that it can be fetched into a buffer overflow in a platform where the stack (and any other place in memory, but libc) is executable:

In 7-8 more years, we will know this as ROP, JOP, and many other *OP

<https://seclists.org/bugtraq/2000/Nov/32>

Exploitation = programming + proofs

General-purpose programmability via exploit primitives is a recurring pattern



Weird machines: mostly harmless?

USENIX WOOT 2013

“Weird Machines” in ELF: A Spotlight on the Underappreciated Metadata

Rebecca Shapiro
Dartmouth College

Sergey Bratus
Dartmouth College

Sean W. Smith
Dartmouth College

USENIX WOOT 2011

Exploiting the hard-working DWARF: Trojan and Exploit Techniques With No Native Executable Code

James Oakley, Sergey Bratus

USENIX WOOT 2013

The Page-Fault Weird Machine: Lessons in Instruction-less Computation

Julian Bangert, Sergey Bratus, Rebecca Shapiro, Sean W. Smith

Core OS mechanisms are unexpectedly Turing-complete as attacker’s input-driven agents

- ELF loader/relocator is T.-c.
 - PE and Mach-O are too (cf. LOCREATE, *Uninformed 6:3*)
- So is the DWARF exception handler VM, helpfully linked into C/C++ programs
- So is the **x86 MMU** on its configs (GDT + IDT + TSS + PTEs)

Defining the common exploitability pattern

IEEE TRANSACTIONS ON
EMERGING TOPICS
IN COMPUTING

Received 15 August 2017; revised 3 November 2017; accepted 11 December 2017.
Date of publication 19 December 2017; date of current version 9 June 2020.

Digital Object Identifier 10.1109/TETC.2017.2785299

Weird Machines, Exploitability, and Provable Unexploitability

THOMAS DULLIEN , (Member, IEEE)

The author is with the Google's Project Zero, Zurich 8002, Switzerland
CORRESPONDING AUTHOR: T. F. DULLIEN (thomas.dullien@gmail.com)

A brief history:

<https://weirdmachines.gitlab.io/>

ABSTRACT The concept of *exploit* is central to computer security, particularly in the context of *memory corruptions*. Yet, in spite of the centrality of the concept and voluminous descriptions of various exploitation techniques or countermeasures, a good theoretical framework for describing and reasoning about exploitation has not yet been put forward. A body of concepts and folk theorems exists in the community of exploitation practitioners; unfortunately, these concepts are rarely written down or made sufficiently precise for people outside of this community to benefit from them. This paper clarifies a number of these concepts, provides a clear definition of exploit, a clear definition of the concept of a *weird machine*, and how programming of a weird machine leads to exploitation. The paper also shows, somewhat counterintuitively, that it is feasible to design some software in a way that even powerful attackers—with the ability to corrupt memory once—cannot gain an advantage. The approach in this paper is focused on *memory corruptions*. While it can be applied to many security vulnerabilities introduced by other programming mistakes, it does not address *side channel attacks*, *protocol weaknesses*, or security problems that are present *by design*.

Not so harmless: Spectre is more than a side-channel

NDSS 2019

ExSpectre: Hiding Malware in Speculative Execution

Jack Wampler
University of Colorado Boulder
jack.wampler@colorado.edu

Ian Martiny
University of Colorado Boulder
ian.martiny@colorado.edu

Eric Wustrow
University of Colorado Boulder
ewust@colorado.edu

ExSpectre shares many properties with *weird machines*—a machine which takes advantage of bugs or unexpected idiosyncracies in existing systems to perform arbitrary computation [6], [7]. In particular ExSpectre showcases the ability to use CPU speculation to compute.

- Modern CPU optimization layers contain **enough shared state and logic** to form a transient, mostly unobservable emergent computing/execution environment
- **Interactions** between different CPU optimizations' internal states can serve as logical **gates and circuits** adding up to a virtual CPU
 - Programmed by seemingly meaningless series of memory reads and writes
 - Results are read off as timings of races

ASPLOS 2021, Distinguished paper

Computing with Time: Microarchitectural Weird Machines

Dmitry Evtushkin
devtyushkin@wm.edu
William & Mary
United States

Thomas Benjamin
tbenjamin@perspectalabs.com
Perspecta Labs
United States

Jesse Elwell
jelwell@perspectalabs.com
Perspecta Labs
United States

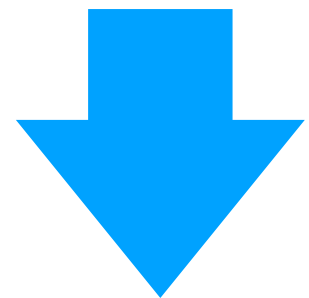
Jeffrey A. Eitel
jeitel@perspectalabs.com
Perspecta Labs
United States

Angelo Sapello
asapello@perspectalabs.com
Perspecta Labs
United States

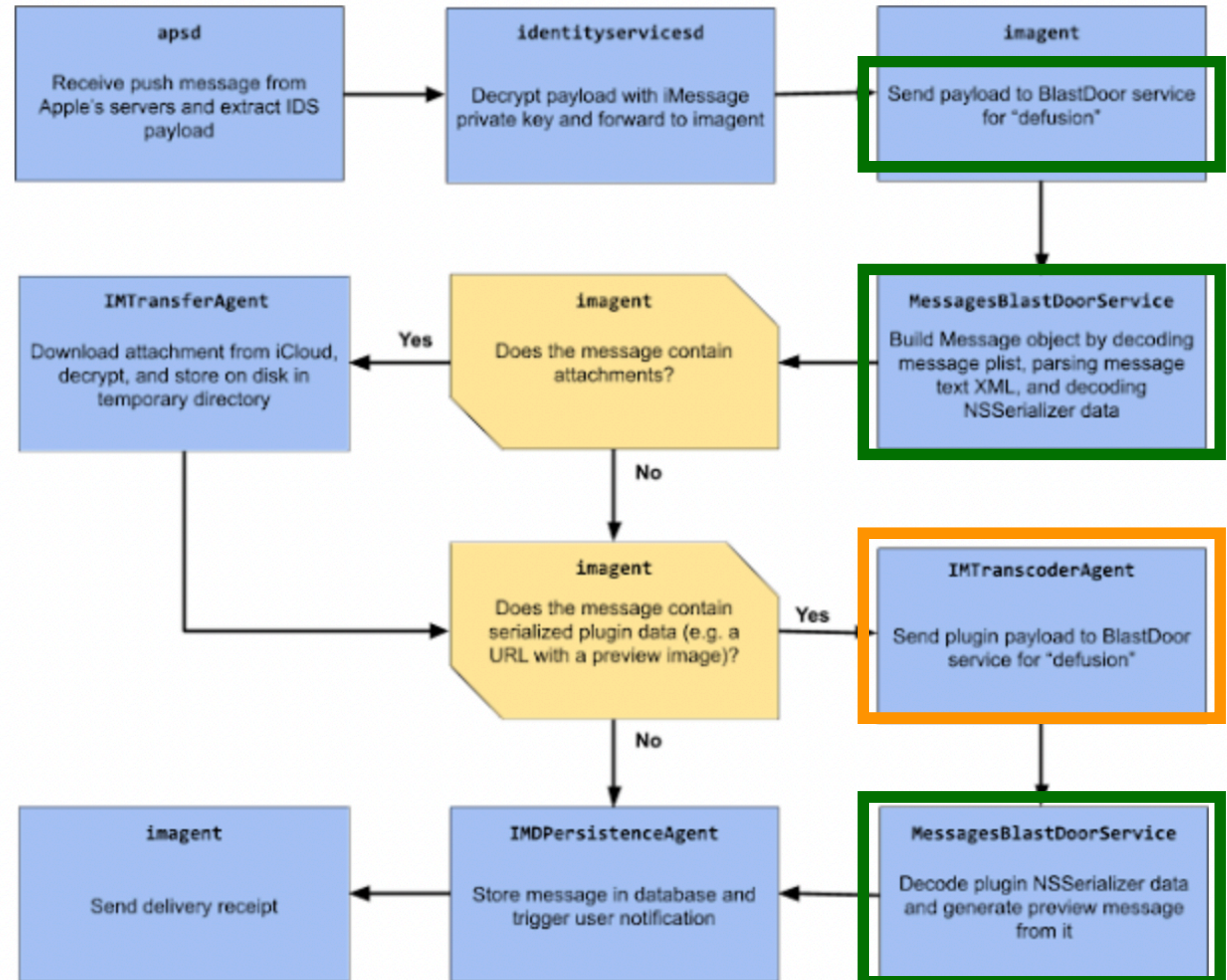
Abhrajit Ghosh
aghosh@perspectalabs.com
Perspecta Labs
United States

No longer just theory: iMessage exploitation

From single click to zero-click



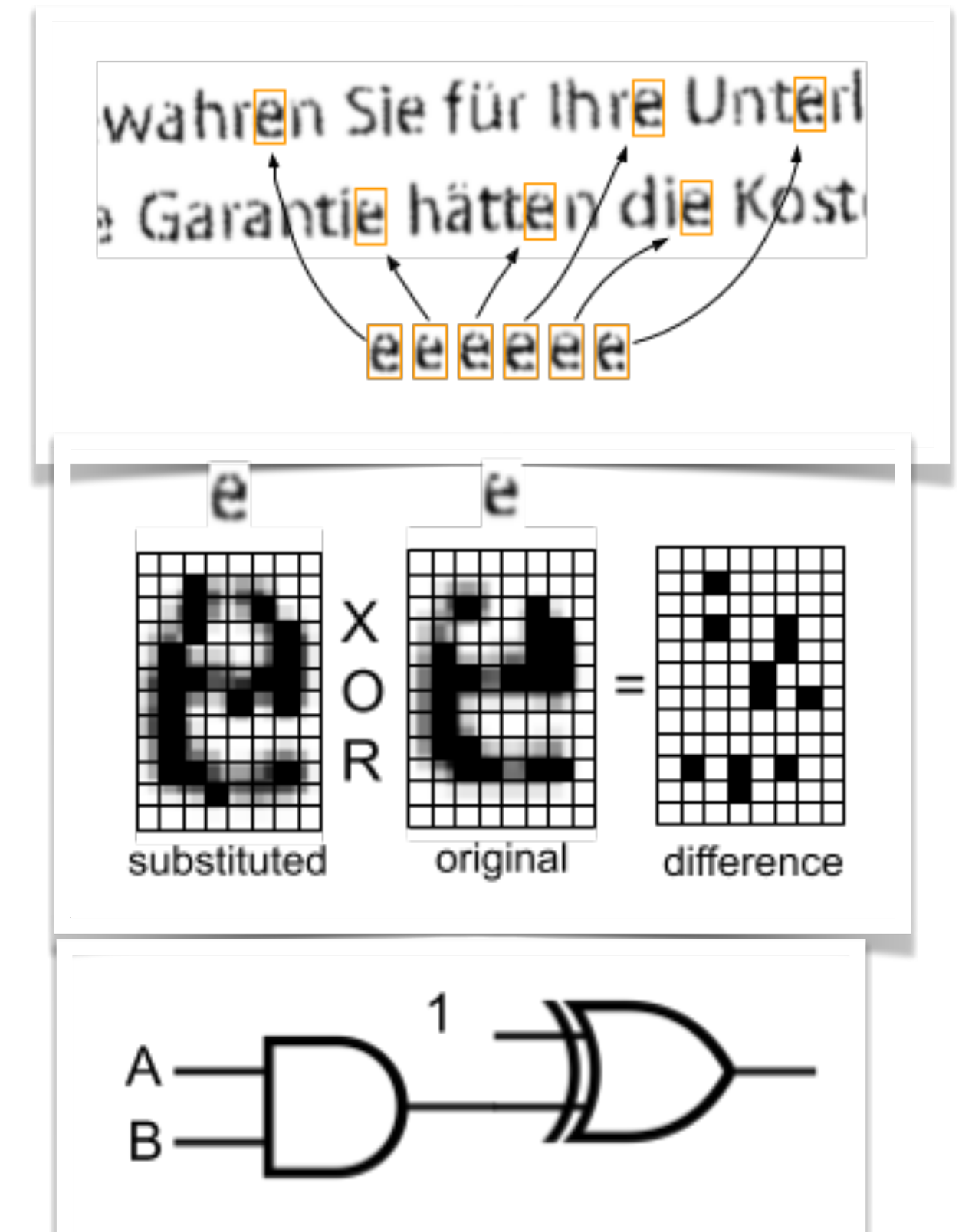
Enter the BlastDoor sandbox



WM in iMessage's looping GIFs

<https://googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html>

- Pass received files with **.gif** extension to **ImageIO** library
- ImageIO ignores extension and guesses file type **[exploit passes a PDF file]**
- **CoreGraphics PDF** parser has an integer overflow, but no JavaScript to exploit it **[are we just going to safely crash? No.]**
- ...but it has **JBIG2** decompression of glyphs, with **XOR** logic over memory areas
 - ...which will apply **out of buffer's bounds**, thanks to the overflow;
 - this makes **logical gates**. Logical gates make a **virtual machine** as fast and reliable as JavaScript!



You can now provide as input a sequence of JBIG2 segment commands which implement a sequence of logical bit operations to apply to the page. And since the page buffer has been unbounded those bit operations can operate on arbitrary memory.

Fully automated exploitation & patching?



DARPA and ARPA-H's Artificial Intelligence Cyber Challenge (AIXCC) brings together the foremost experts in AI and cybersecurity to safeguard the software critical to all Americans.

AIXCC is excited to have Anthropic, Google, Microsoft, OpenAI, the Linux Foundation, the Open Source Security Foundation, Black Hat USA, and DEF CON as collaborators in this effort.

**Where else is hacker math
hiding?**

“IRs are magic”

Intermediate Representations make analysis go



PaX - gcc plugins galore

PaX Team

H2HC 2013.10.05

GCC Overview

- ▶ GCC AST: language frontends produce GENERIC
 - ▶ Data structure: tree
 - ▶ Plugins can implement new attributes and pragmas, inspect structure declarations and variable definitions (gcc 4.6+)
- ▶ GCC IR #1: GIMPLE
 - ▶ Static Single Assignment (SSA) based representation
 - ▶ First set of optimization/transformation passes runs on GIMPLE (`-fdump-ipa-all`, `-fdump-tree-all`)
 - ▶ Data structures: `cgraph_node`, `function`, `basic_block`, `gimple`, `tree`
- ▶ GCC IR #2: RTL
 - ▶ GIMPLE is lowered to RTL (pre-SSA gcc had only this)
 - ▶ Second set of optimization passes runs on RTL (`-fdump-rtl-all`)
 - ▶ Data structures: `rtx`, `tree`

- GrSecurity/PaX made revolutionary Linux kernel hardening with compiler plugins — operating over GCC IRs

Towers of Intermediate Representations

“IRs are useful. What’s an IR?”

- IRs are everywhere
 - LLVM passes ~ IRs, MLIR
 - Ghidra uses P-code
 - Angr uses VEX
 - Binary Ninja has 3 public IRs
- But what is an IR?
 - Trail of Bits: *why only one?*

Finding bugs in C code with Multi-Level IR and VAST

POST JUNE 15, 2023 1 COMMENT

Intermediate languages (IRs) are what reverse engineers and vulnerability researchers use to see the forest for the trees. IRs are used to view programs at different abstraction layers, so that analysis can understand both low-level code aberrations and higher levels of flawed logic mistakes. The setback is that **bug-finding tools are often pigeonholed into choosing a specific IR**, because **bugs don’t uniformly exist across abstraction levels**.

We developed a new tool called **VAST** that solves this problem by providing a **“tower of IRs,”** allowing a program analysis to start at the best-fit representation for the analysis goal, then work upwards or downwards as needed. For instance, an analyst may want to do one of three things with a stack-based buffer overflow. (1) Identify it. (2) Classify it. (3) Remediate it.

Bugs span the semantic gap, and so should analyses!

Move up and down the tower of IRs as needed

Now comes choosing the right IR. Some bug properties are only apparent at certain abstraction levels. A buffer overflow is easily identified in LLVM IR, because stack buffers in LLVM IR are highly characteristic (i.e., created via the `alloca` instruction). This is the “best-fit” IR for identification.

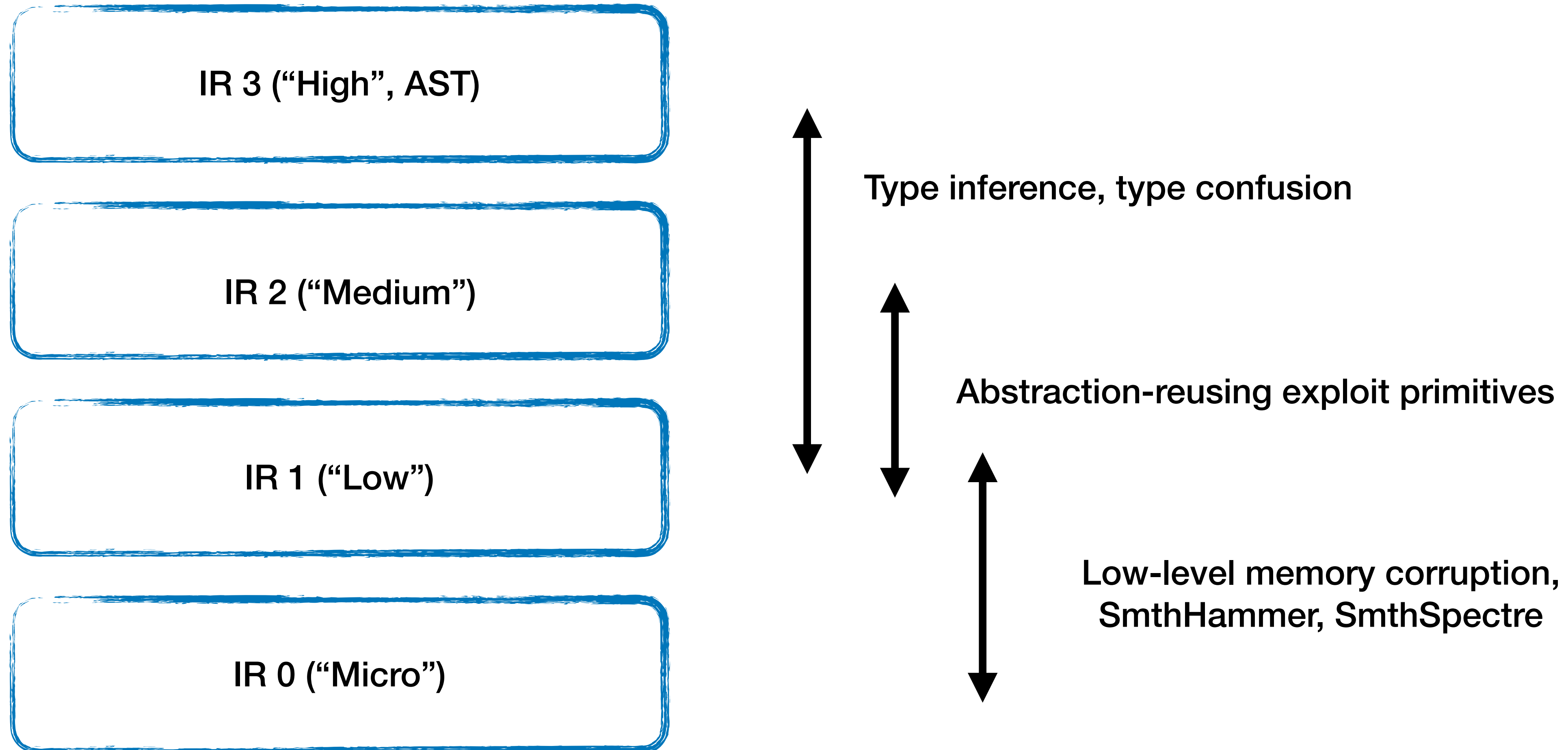
For *classification*, a buffer overflow can go from a common bug to a security threat if the buffer sits near sensitive data in program memory. This only becomes clear below the LLVM IR level, near or at the machine code level, where buffers are **fused together** with other sensitive information, forming a **“stack frame.”**

The last part of the story is *communication* and *remediation*. The reason why the buffer overflowed in the first place can be a **side-effect of a type conversion on a buffer index that was self-evident in the program’s abstract syntax tree (AST)**, the highest level IR. **Connecting these facts together used to be impossible, but VAST’s tower of IRs is changing this.** **Bugs span the semantic gap, and so should analyses.**

- Buffer overflows: LLVM IR
- Adjacency: below LLVM IR
- Root causes like out-of-type references: AST
- ToB solution: **VAST/Multiplier**
 - Get all the IRs (as dialects of MLIR)
 - “Move up or down as needed”

A tower of IRs

A sequence of compatible, interoperating IRs



“For my analyses, I’d rather use DSLs”

Hard problems remain hard, but scalability increases

Program Analysis for Domain Specific Language Extraction of Legacy Software

DARPA · Jul 2021

As part of the DARPA V-SPELLS program, this project aims to automate domain specific program analysis. There are inherently hard challenges in general program analysis, such as handling pointers, indirect calls, constructing loop invariants, and decompilation, despite the steady progress the community has been making. Fuzzing techniques and bug finding tools are still limited to finding low level bugs such as memory bugs, and formal methods often require substantial human efforts to translate domain specific and application specific properties down to annotations to implementation artifacts. The project focuses on lifting implementation to post hoc domain specific models, providing a new

Prof. Xiangyu Zhang, Purdue U.

<https://www.cs.purdue.edu/homes/xyzhang/>

implementation to post hoc domain specific models, providing a new perspective to these hard problems. Instead of dealing with the low level implementation details, we abstract them away such that their high-level semantics become clean and easy to reason. With lifted domain models, domain specific properties can be easily checked. This allows existing fuzzers to find complex logical bugs, formal methods can be substantially simplified and automated. We are interested in lifting implementations in various domains such as parsers, network protocols, robotic systems, smart contracts, and even binary executables.

Lifting gets CVEs that fuzzing misses

50+ new CVEs in network protocols

Lifting Network Protocol Implementation to Precise Format Specification with Security Applications

Qingkai Shi
Purdue University
West Lafayette, USA
shi553@purdue.edu

Junyang Shao
Purdue University
West Lafayette, USA
shao156@purdue.edu

Yapeng Ye
Purdue University
West Lafayette, USA
ye203@purdue.edu

Mingwei Zheng
Purdue University
West Lafayette, USA
zheng618@purdue.edu

Xiangyu Zhang
Purdue University
West Lafayette, USA
xyzhang@cs.purdue.edu

We implement our method as a tool, namely Netlifter **to lift packet formats from source code in C.** It is implemented on top of the LLVM (12.0.0) compiler infrastructure [50] and the Z3 (4.8.12) SMT solver [37]. The source code of a protocol is compiled into the LLVM bitcode, where we perform our static analysis. In the analysis, Z3 is used to represent abstract values as symbolic expressions and solve path constraints. All experiments are run on a Macbook Pro

Extracting Protocol Format as State Machine via Controlled Static Loop Analysis

Qingkai Shi
Purdue University

Xiangzhe Xu
Purdue University

Xiangyu Zhang
Purdue University

Our new technique **extracts a sound state machine** by regarding each loop iteration as a state and the dependency between loop iterations as state transitions. To achieve high, i.e., path-sensitive, precision but avoid path explosion, the analysis is controlled to merge as many paths as possible based on carefully-designed rules. The evaluation results show that we can infer a state machine and, thus, the message formats, in five minutes with over 90% precision and recall.

Table 1: Protocols and Their Codebases for Evaluation

| Name | Codebase | Size (kloc) | Time (sec.) | Description |
|---------|---------------------|-------------|-------------|---|
| L2CAP | linux/bluetooth [7] | 38 | 12 | logical link ctrl and adaptation proto. |
| SMP | linux/bluetooth [7] | 12 | 2 | low energy security manager proto. |
| APDU | opensc [14] | 3 | 3 | application proto. data unit |
| OSDP | libosdp [5] | 14 | 27 | open supervised device proto. |
| SSQ | libssq [8] | 8 | 1 | source server query proto. |
| TCP/IP | lwip [6] | 41 | 53 | transport control & internet proto. |
| IGMP/IP | lwip [6] | 17 | 16 | internet group mgmt. & internet proto. |
| QUIC | ngtcp2 [4] | 59 | 11 | general-purpose transport layer proto. |
| BABEL | frouting [3] | 7 | 9 | a distance-vector routing proto. |
| IS-IS | frouting [3] | 22 | 6 | intermediate system (IS) to IS proto. |

Codebases chosen by recent activity, presence of fuzzing harness

50+ new CVEs claimed

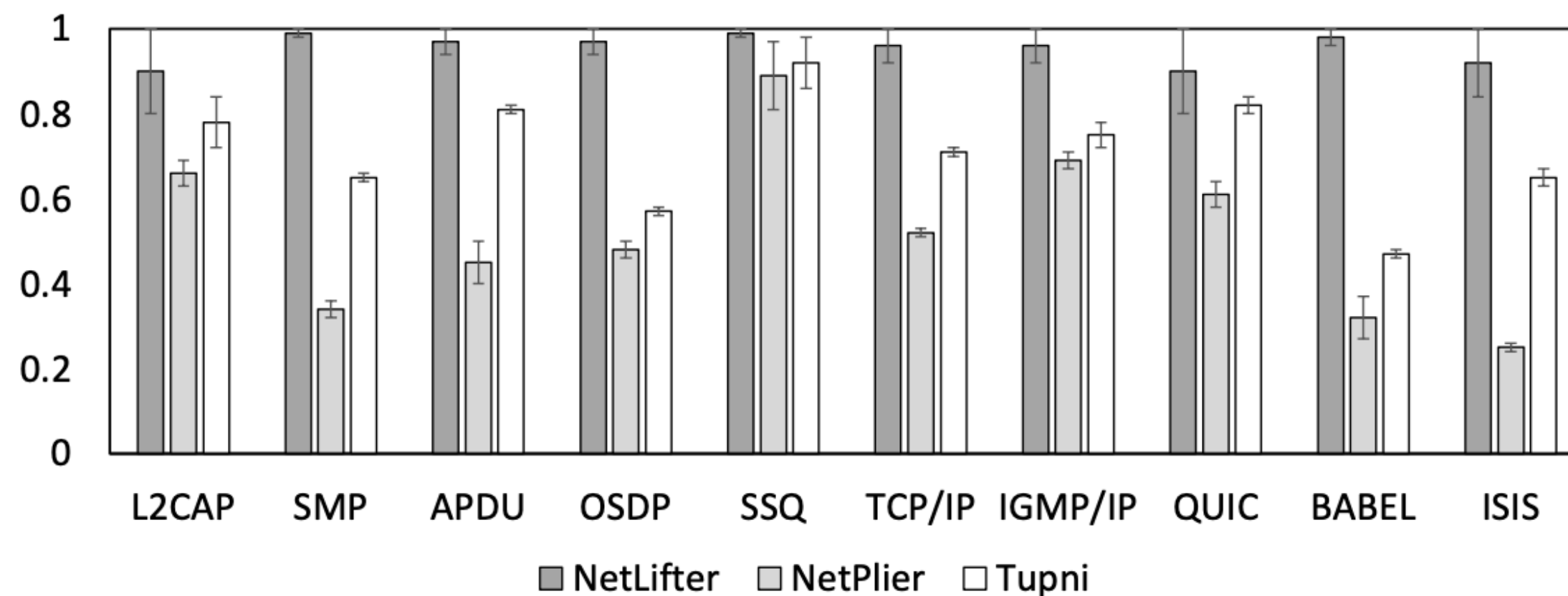
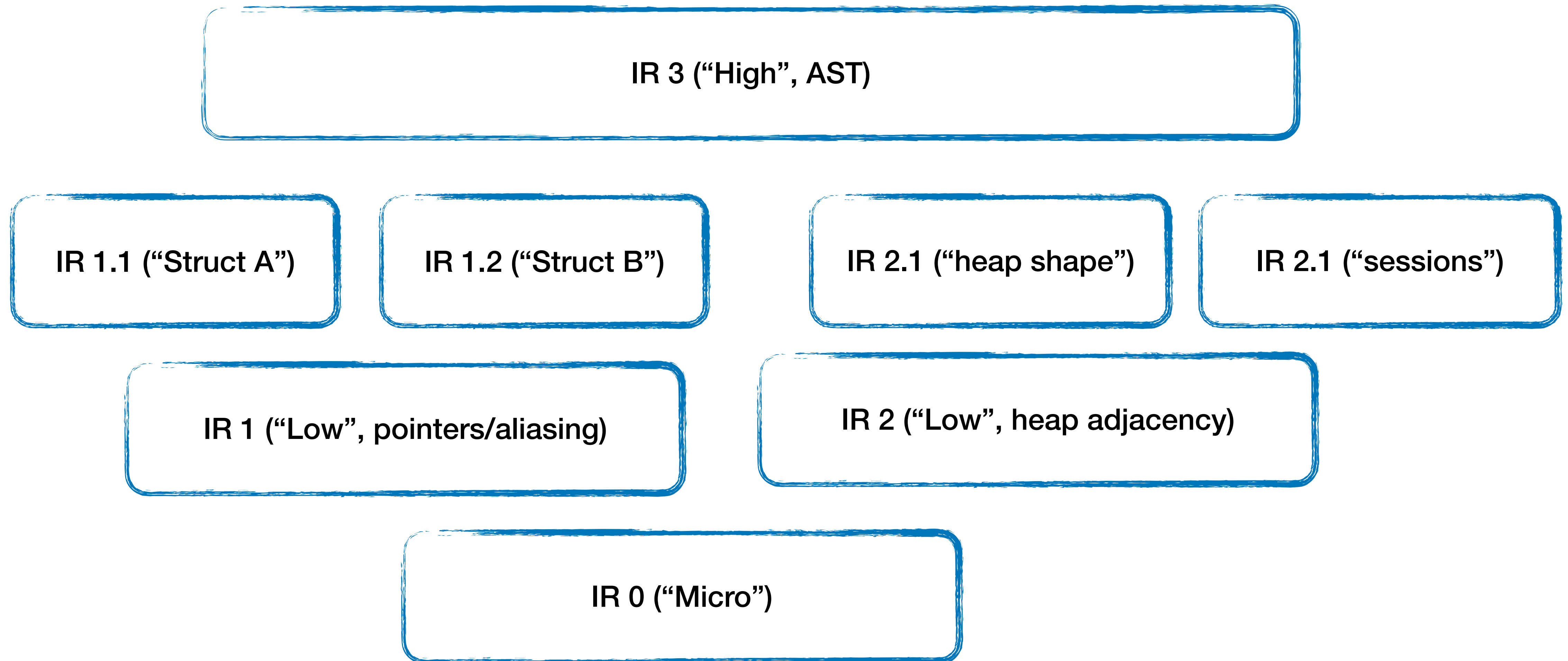


Figure 13: The y-axis is the number of covered branches normalized to one. It shows the branch coverage averaged over twenty runs with a 95% confidence interval.

| | | | | |
|-------|---------------------|----|----|--------------------------------------|
| A2MP | linux/bluetooth [7] | 16 | 2 | amp manager proto. |
| BNEP | linux/bluetooth [7] | 15 | 3 | BT network encapsulation proto. |
| CMTDP | linux/bluetooth [7] | 20 | 1 | c-api message transport proto. |
| HIDP | linux/bluetooth [7] | 17 | 4 | human interface device proto. |
| UDP | lwip [6] | 37 | 33 | user datagram proto. |
| ICMP | lwip [6] | 22 | 12 | internet control message proto. |
| DHCP | lwip [6] | 25 | 43 | dynamic host configuration proto. |
| ICMP6 | lwip [6] | 30 | 54 | internet control message proto. v6 |
| DHCP6 | lwip [6] | 35 | 51 | dynamic host configuration proto. v6 |
| BGP | frouting [3] | 13 | 2 | border gateway proto. |
| LDP | frouting [3] | 20 | 5 | label distribution proto. |
| BFD | frouting [3] | 10 | 17 | bidirectional forwarding detection |
| VRRP | frouting [3] | 8 | 12 | virtual router redundancy proto. |
| EIGRP | frouting [3] | 14 | 21 | interior gateway routing proto. |
| NHRP | frouting [3] | 11 | 11 | next hop resolution proto. |
| OSPF2 | frouting [3] | 9 | 14 | open shortest path first v2 |
| OSPF3 | frouting [3] | 7 | 16 | open shortest path first v3 |
| RIP1 | frouting [3] | 11 | 13 | routing information proto. v1 |
| RIP2 | frouting [3] | 11 | 15 | routing information proto. v2 |
| RIPng | frouting [3] | 7 | 41 | routing information proto. for ip6 |





A tree of IRs? A lattice of IRs?

“A sufficiently lifted IR is indistinguishable from a DSL”



Reverse Engineering ~ IR tower/tree lifting?

Halvar's "RE 2006: New Challenges Need Changing Tools" talk

- #1 and #2: Automated data structure recovery; building UML inheritance diagrams from binaries. 
- Coupling the above with a debugger to allow run-time object inspection and editing.
- #3: Automated modularization of binaries (decomposing binaries to recover library structure / groupings). 
- #4: De-templating of heavily templated C++ code.
- #7: "Normal forms" for sequences of code (a Groebner-base equivalent?)
- #8: A visualization for callgraphs that shows each node as a Poset to make sure the order of outgoing edges is visualized, too.
- 9#: Recovery of the internal state machine of a target. 
- 10#: Semantics-based FLIRT-style library identification. 

Interestingly, challenge #5 - automated input data creation - is the one where most progress has happened since the talk. To my great amusement, this talk suggests the use of SAT solvers to do it. At that time, I was obviously unaware at the time of the research on SMT that is happening and will lead to Vijay Ganesh's great 2007 thesis (and the release of STP).

Bottom-up verification

Lifting from a sound foundation



libLISA: Instruction Discovery and Analysis on x86-64

JOS CRAAIJO, Open Universiteit, Netherlands

FREEK VERBEEK, Open Universiteit, Netherlands and Virginia Tech, USA

BINOY RAVINDRAN, Virginia Tech, USA

Even though heavily researched, a full formal model of the x86-64 instruction set is still not available. We present **libLISA**, a tool for automated discovery and analysis of the ISA of a CPU. This produces the most extensive formal x86-64 model to date, with over 118 000 different instruction groups. The process requires as little human specification as possible: specifically, we do not rely on a human-written (dis)assembler to dictate which instructions are executable on a given CPU, or what their in- and outputs are. The generated model is CPU-specific: behavior that is “undefined” is synthesized for the current machine. Producing models for five different x86-64 machines, we mutually compare them, discover undocumented instructions, and generate instruction sequences that are CPU-specific. Experimental evaluation shows that we enumerate virtually all instructions within scope, that the instructions’ semantics are correct w.r.t. existing work, and that we improve existing work by exposing bugs in their handwritten models.

Recompilable/patchable/verifiable IRs

Recompilable disassembly with proofs

Verifiably Correct Lifting of Position-Independent x86-64 Binaries to Symbolized Assembly

Freek Verbeek
freek@vt.edu

Nico Naus
nico.naus@ou.nl

Binoy Ravindran
binoy@vt.edu

We present an approach to lift position-independent x86-64 binaries to symbolized NASM. Symbolization is a decompilation step that enables binary patching: functions can be modified, and instructions can be interspersed. Moreover, it is the first abstraction step in a larger decompilation chain. The produced NASM is recompilable, and we extensively test the recompiled binaries to see if they exhibit the same behavior as the original ones. In addition to testing, the produced NASM is accompanied with a certificate, constructed in such a way that if all theorems in the certificate hold, symbolization has occurred correctly. The original and recompiled binary are lifted again with a third-party decompiler (Ghidra).



- (1) The first lifting tool for lifting PIE x86-64 ELF binaries to symbolized NASM;
- (2) An approach to formal validation of recompiled binaries;
- (3) A demonstration of use-cases for binary patching enabled by symbolized NASM lifting;
- (4) Experimental results comparing original and recompiled binaries.

Is this the year of the first-class* IRs?

(*) Machine-readably defined as mathematical objects friendly to efficient algorithms

Right representation => Math => Algorithm => Tool => Pwnage

Hyperplanes

An experimental system for Interconnected IRs

Slides and examples to be posted be at <https://hyperplanes.io/talks/h2hc/>

Memorizer

Object-granular instrumentation of the Linux 6.6 kernels

- Stable, available with build instructions, manuals, and usage examples
 - <https://github.com/ITI/memorizer/>
 - Docs: <https://github.com/ITI/memorizer/tree/linux-6.6.y-memorizer-dev/Documentation>
 - Builds for Qemu, x86-64, InitRAM

Thank you

