Defending against Memory Corruption Vulnerability Exploitation

Michalis Polychronakis

Associate Professor, Stony Brook University

REACT Workshop – 20 May 2021

The Problem

Software vulnerability exploitation

- Among the leading causes of system compromise and malware infection

We have to live with C/C++

- Performance, compatibility, developer familiarity, vast existing code base, …
- Many memory-safe programming languages exist, but full transition requires an immense rewriting effort
- Unlikely to happen any time soon for systems code, core server and client software, resource-constrained IoT devices, … *(but we have started!)*

Memory corruption bugs in network-facing software can turn into *remotely exploitable vulnerabilities*

Dashboard: Zero-Days in Desktop W

https://www.radsix.com/dashboard1/

Dashboard: Zero-Days in Web Browsers

Chrome	Firefox	IE	Hardened IE*	Safari
11	5	5	1	7

The web browser with the most zero-day exploits in recent history is **Chrome**

Date	Browser	CVE Reference	CVSS	Type	Vendor Advisory
12 Mar 2021	Chrome	CVE-2021-21193	8.8	Use-after-free	Link
02 Mar 2021	Chrome	CVE-2021-21166	8.8	Object lifecycle	Link
05 Feb 2021	Chrome	CVE-2021-21148	8.8	Heap corruption	Link
04 Feb 2021	IE	CVE-2021-26411	8.8	Heap corruption	Link
11 Nov 2020	Chrome	CVE-2020-16017	8.8	Security bypass	Link
11 Nov 2020	Chrome	CVE-2020-16013	8.8	Heap corruption	Link
02 Nov 2020	Chrome	CVE-2020-16009	8.8	Heap corruption	Link
20 Oct 2020	Chrome	CVE-2020-15999	8.8	Heap corruption	Link
11 Aug 2020	IE	CVE-2020-1380	7.5	Use-after-free	Link
14 Jul 2020	Chrome	CVE-2020-6519	8.2	Security bypass	Link
03 Apr 2020	Firefox	CVE-2020-6820	8.8	Use-after-free	Link
03 Apr 2020	Firefox	CVE-2020-6819	8.8	Use-after-free	Link

0day "In the Wild" - Google She

https://docs.google.com/spreadsheets/d/1Y5_qtnJ3YcAH9I8x5vbMggSxIn6srz4_gzUS47Ze5UY/edit#gid=112329262

CVE	Vendor	Product	Type	Description	Date Patched	Reported By
CVE-2021-21193	Google	Chrome	Memory Corruption	Use-after-free in Blink	2021-03-12	???
CVE-2021-26411	Microsoft	Internet Explorer	Memory Corruption	Use-after-free in MSHTML	2021-03-09	yangkang(@dnpushme) & huangyi(
CVE-2021-21166	Google	Chrome	Memory Corruption	Object lifecycle issue in audio	2021-03-02	Alison Huffman, Microsoft Browser
CVE-2021-27065	Microsoft	Exchange Server	Logic/Design Flaw	Arbitrary file write	2021-03-02	Volexity, Orange Tsai from DEVCO
CVE-2021-26858	Microsoft	Exchange Server	Logic/Design Flaw	Arbitrary file write	2021-03-02	Microsoft Threat Intelligence Center
CVE-2021-26857	Microsoft	Exchange Server	Logic/Design Flaw	Insecure deserialization in the Unifed Messaging service	2021-03-02	Dubex and Microsoft Threat Intellige
CVE-2021-26855	Microsoft	Exchange Server	Logic/Design Flaw	Server-side request forgery (SSRF)	2021-03-02	Volexity, Orange Tsai from DEVCO
CVE-2021-1732	Microsoft	Windows	Memory Corruption	Unspecified win32k escalation of privilege	2021-02-09	JinQuan, MaDongZe, TuXiaoYi, and
CVE-2021-21017	Adobe	Reader	Memory Corruption	Heap-based buffer overflow	2021-02-09	???
CVE-2021-21148	Google	Chrome	Memory Corruption	Heap buffer overflow in V8	2021-02-04	Mattias Buelens
CVE-2021-1871	Apple	iOS	Logic/Design Flaw	Unspecified logic flaw in Webkit	2021-01-26	???
CVE-2021-1870	Apple	iOS	Logic/Design Flaw	Unspecified logic flaw in Webkit	2021-01-26	???
CVE-2021-1782	Apple	iOS	Memory Corruption	Unspecified kernel race condition	2021-01-26	???
CVE-2021-1647	Microsoft	Windows Defender	Memory Corruption	Unspecified remote code execution in Windows Defende	2021-01-12	???
CVE-2020-16017	Google	Chrome	Memory Corruption	Use-after-free in site isolation	2020-11-11	???
CVE-2020-16013	Google	Chrome	Memory Corruption	Unspecified memory corruption in v8	2020-11-11	???
CVE-2020-27932	Apple	iOS	Memory Corruption	Unspecified type confusion in kernel	2020-11-05	Google Project Zero
CVE-2020-27950	Apple	iOS	Information Leak	Unspecified memory initialization issue in kernel	2020-11-05	Google Project Zero
CVE-2020-27930	Apple	iOS	Memory Corruption	Unspecified memory corruption in font parsing	2020-11-05	Google Project Zero
CVE-2020-16010	Google	Chrome	Memory Corruption	Unspecified memory corruption in Chrome on Android sa	2020-11-02	Google Project Zero
CVE-2020-16009	Google	Chrome	Memory Corruption	Type confusion in TurboFan map deprecation	2020-11-02	Google Project Zero/Google TAG
CVE-2020-17087	Microsoft	Windows	Memory Corruption	Heap buffer overflow in cng.sys IOCTL 0x390400	2020-11-10	Google Project Zero

Introduction | All | 2021 | 2020 | 2019 | 2018 | 2017 | 2016 | 2015

Defending against Vulnerability Exploitation

Finding and killing bugs

- Sanitizers, fuzzing, symbolic execution, bug bounties, …
- *Who* will find the next 0-day?

~~Retrofit memory safety to C/C++~~ ➔ *rewrite critical components in Rust/Go*

- Eradicate the root cause of the problem: *memory errors*
- Performance and compatibility challenges
- No protection against transient execution attacks (!)

THIS TALK

Exploit mitigations

- Assuming a vulnerability exists, "raise the bar" for exploitation
- DEP, GS, SafeSEH, SEHOP, ASLR, CFI, sandboxing, …

Exploit Mitigations Do Raise the Bar…

Pwn2Own 2007

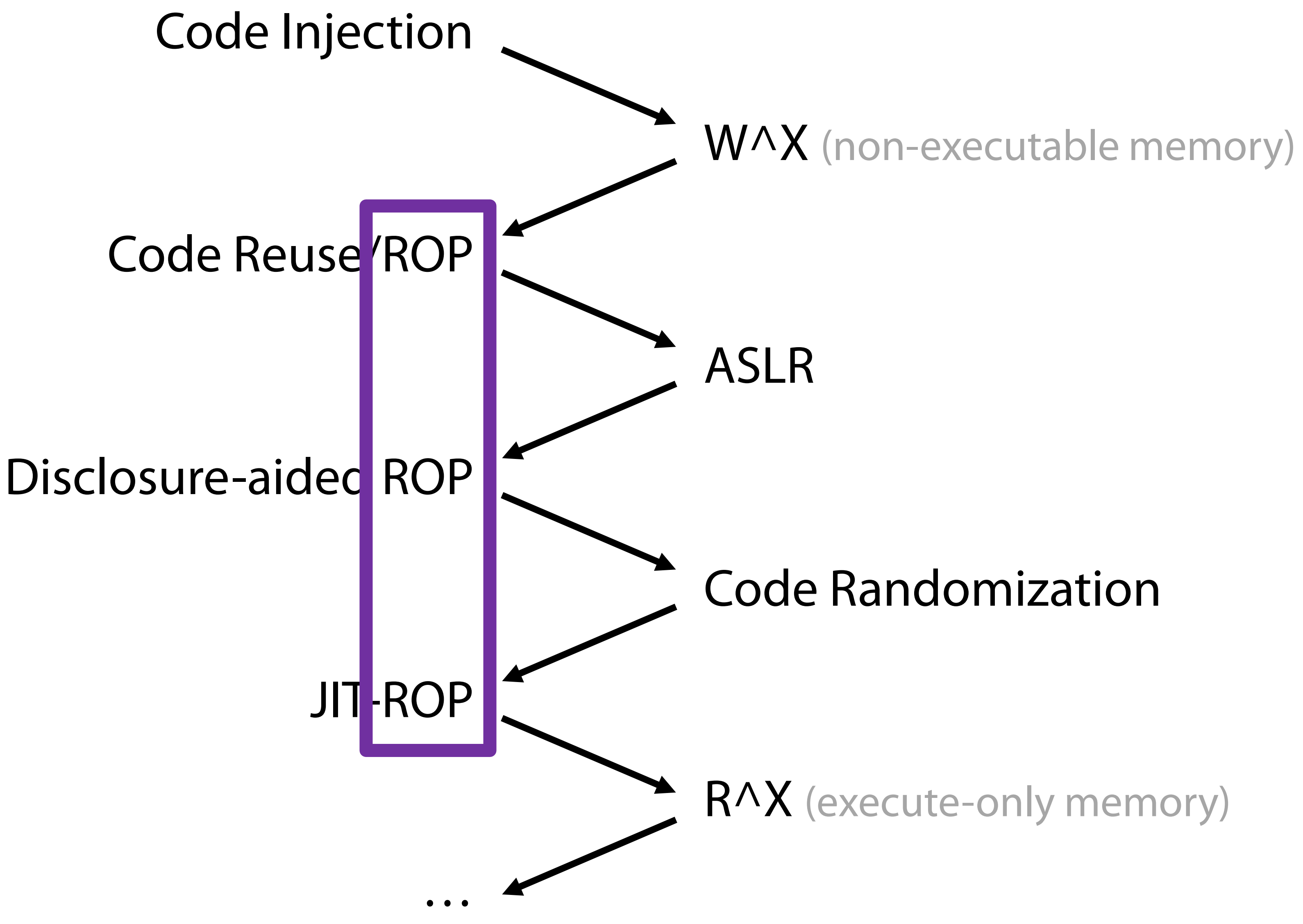
“A New York-based security researcher [Dino Dai Zovi] spent less than 12 hours to identify and exploit a zero-day vulnerability in Apple's Safari browser” [1]

Pwn2Own a decade later

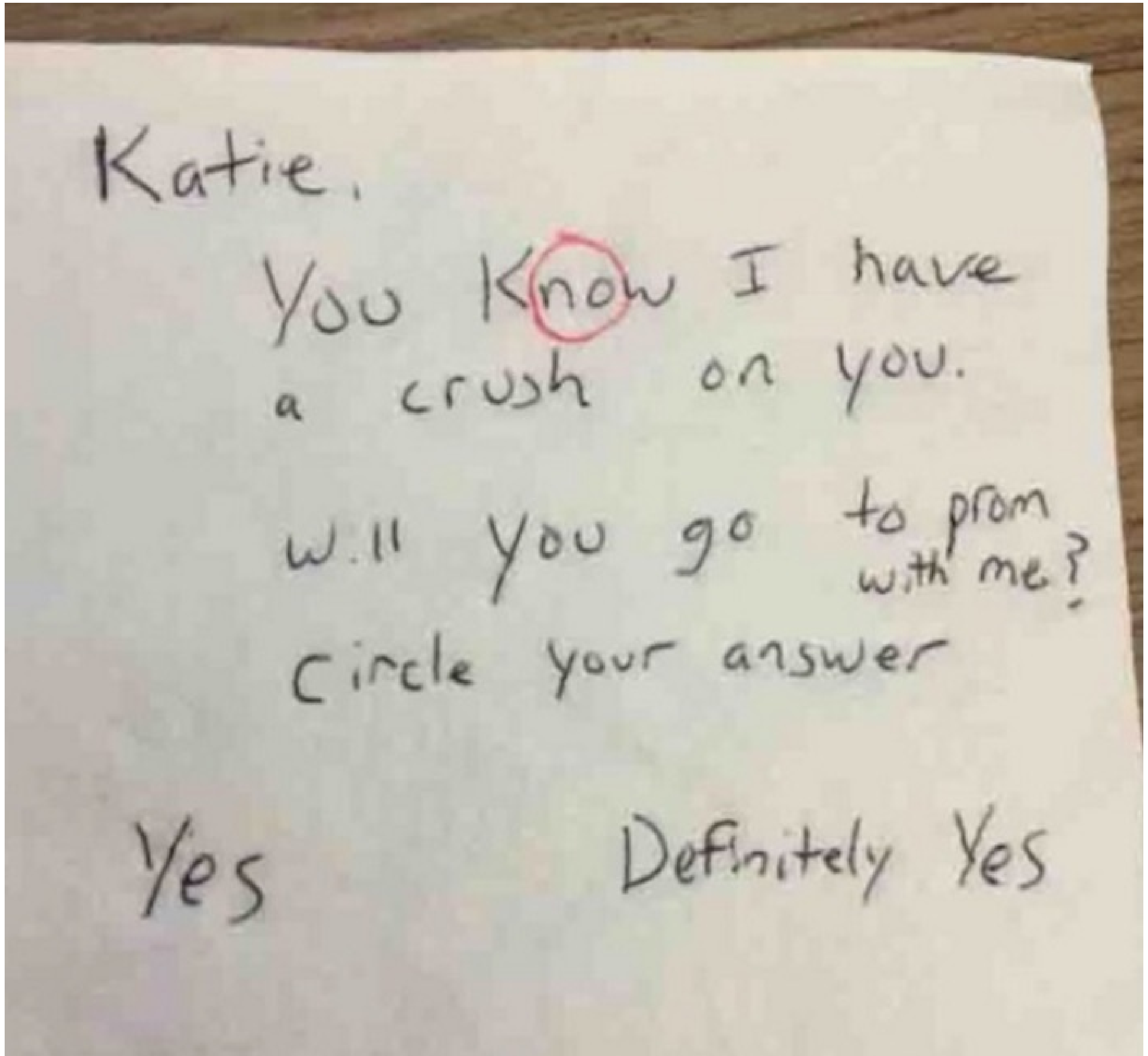
“This year saw several teams sponsored by their employers participating” [2]

[1] https://www.theregister.co.uk/2007/04/20/pwn-2-own_winner/
[2] https://blog.trendmicro.com/pwn2own-2017-event-ages/

…but Attackers Can Often Knock the Bar Off



Code Reuse (Return-oriented Programming)



Problem 1: ***Software monocultures and code bloat are facilitators of vulnerability exploitation***

Research Goal: ***Practical software specialization and shielding***

Unneeded code and logic removal

Reduce the attack surface

Code diversification

Undermine the assumptions of adversaries

Code Reuse → **Memory Disclosure**

Problem 2: ***Transient execution attacks bypass existing memory safety and isolation techniques***

Research Goal: ***Robust in-memory data protection***

Selective data isolation

Keep sensitive data always encrypted in memory

Code Specialization

Temporal System Call Specialization for Attack Surface Reduction – *USENIX Security 2020*

Confine: Automated System Call Policy Generation for Container Attack Surface Reduction – *RAID 2020*

Saffire: Context-sensitive Function Specialization and Hardening against Code Reuse Attacks – *IEEE EuroS&P 2020*

Configuration-driven Software Debloating – *EuroSec 2019*

Shredder: Breaking Exploits through API Specialization – *ACSAC 2018*

Software Debloating and Specialization

Development using libraries, frameworks, and toolkits has many benefits

- Rapid program development
- Disk and memory savings
- Easy maintenance: bug fixes, security patches, …

But applications end up including code they don’t use and have access to features they don’t need

- Some libraries/modules/plugins are not needed by certain *(or default)* configurations
- Some library functions are not imported at all
- Some system calls are never used
- …

Software Debloating and Specialization

Code bloat ➔ *increased attack surface*

Unneeded code: may still contain exploitable vulnerabilities (e.g., Heartbleed)

Unneeded code: more ROP gadgets for writing code reuse exploits

Unused (dangerous) system calls: exploit code can still invoke them to perform harmful operations (e.g., `execve`, `mprotect`)

Unused system calls: entry points for exploiting kernel vulnerabilities that can lead to privilege escalation

Our goal: *reduce the attack surface by removing unneeded code*

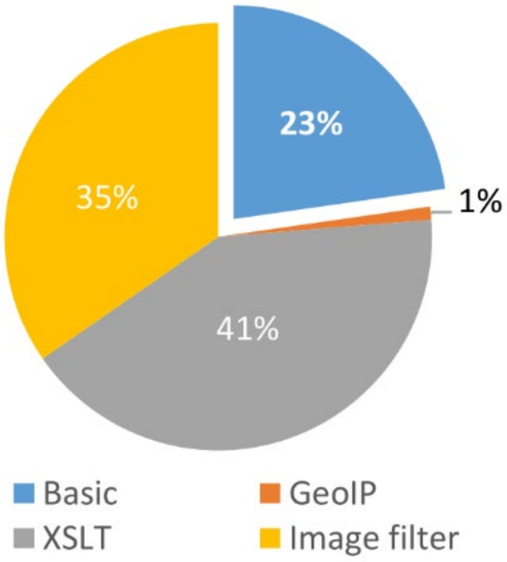
Main benefits:

Break exploit payloads (shellcode, ROP) or at least hinder their construction

Neutralize kernel vulnerabilities associated with certain system calls

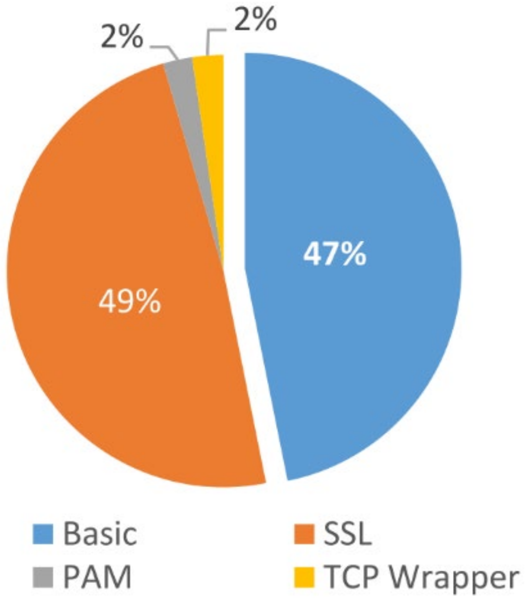
```
# /etc/nginx/nginx.conf
worker_processes 1;
error_log /var/log/nginx/error.log;

events { worker_connections 1024; }

http {
  include mime.types;
  index default.html default.htm;
  default_type application/octet-stream;

  access_log /usr/local/nginx/logs/nginx.pid;
  geoip_country /usr/local/nginx/conf/GeoIP.dat;  # libGeoIP.so
  charset UTF-8;
  keepalive_timeout 65;

  server {
    listen 443 ssl;                  # libssl.so
    gzip on;                         # libz.so
    ssl_certificate cert.pem;        # libssl.so
    ssl_certificate_key cert.key;    # libssl.so

    location / {
      root /var/www/hexlab;
      index default.php;
      image_filter resize 150 100;   # libgd.so
      rewrite ^(.*)$ /msie/$1 break; # libpcre.so
    }

    location /test {
      xml_entities /var/www/hexlab/entities.dtd;  # libxml2.so
      xslt_stylesheet /var/www/hexlab/one.xslt;   # libxslt.so
    }
  }
}
```

Configuration-driven Debloating [EuroSec '19]



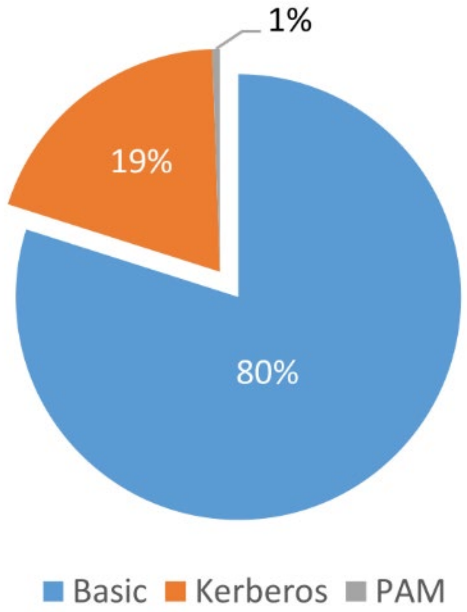
Nginx: **77%**
(25 out of 33 libraries)

*More than **3/4** of the code is removed (!)*



VSFTPD: **53%**
(7 out of 10 libraries)

*More than **half** of the code is removed (!)*



OpenSSH: **20%**
(7 out 22 libraries)

***1/5** of the code is removed*

Kernel Attack Surface Reduction

Most kernel CVEs that lead to local privilege escalation/container escape involve bugs in the implementation of specific system calls

- Exposing fewer system calls to containers reduces the kernel's attack surface

Docker prohibits access to 44 (rarely used) system calls by default

- Enforced by applying a Seccomp BPF filter during initialization

What about the rest of the system calls? Do all containers need them?

- Linux kernel v4.15 provides **333** system calls

Our goal: disable as many system calls as possible according to the actual needs of a given container

Confine: System Call Filtering for Containers [RAID '20]

Previous approaches: dynamic analysis and training

- Drawback: workload-specific, challenging to exercise all the code that may be needed

Static code analysis

- Inspect all execution paths of the containerized application and all its dependencies
- Identify the superset of system calls required for the operation of the container

Input: Docker container image

Output: ready-to-use Seccomp filter

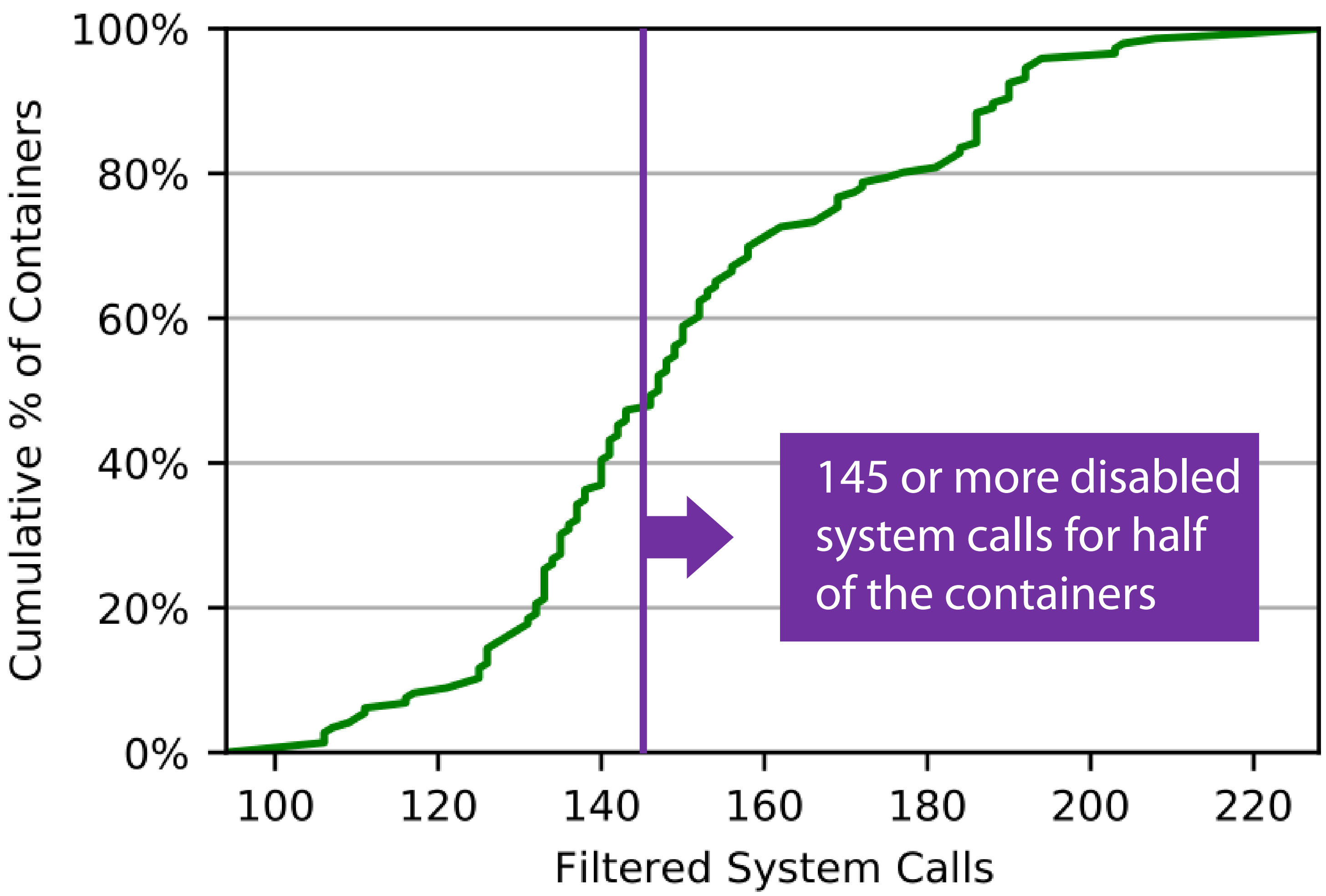
Open-source prototype: https://github.com/shamedgh/confine

Evaluation: Disabled System Calls

Data set: 150 Docker images downloaded from Docker Hub

Confine disables 145 or more system calls (out of 326) for about half of the containers

Worst case: 100 or more disabled system calls *(still at least twice than Docker's default filter)*



Evaluation: Neutralized Kernel CVEs

CVE to kernel function mapping

- Collected Linux kernel CVEs through web scraping
- Mapped CVEs to source code file and line based on git commit messages
- Assigned CVEs to functions

Created Linux kernel call graph based on KIRIN [1]

Result: **28 CVEs removed**

- 7 removed from more than 123 containers
- 16 removed from more than 100 containers

[1] PeX: A Permission Check Analysis Framework for Linux Kernel – USENIX Security '19

Can We Do Better?

Consider the behavior of processes across time [USENIX Security '20]

- Disable additional (dangerous) system calls that are needed only during the *initialization* phase, after entering the *serving* phase
- Example: Apache and Nginx invoke `execve` only during initialization

Specialize the remaining API calls [ACSAC '18, EuroS&P '20]

- Create a custom function per call site, tailored to its arguments
- Static argument binding: eliminate arguments with static values and concretize them within the function body
- Dynamic argument binding: apply a narrow-scope form of data flow integrity to restrict the acceptable values of arguments that cannot be statically derived

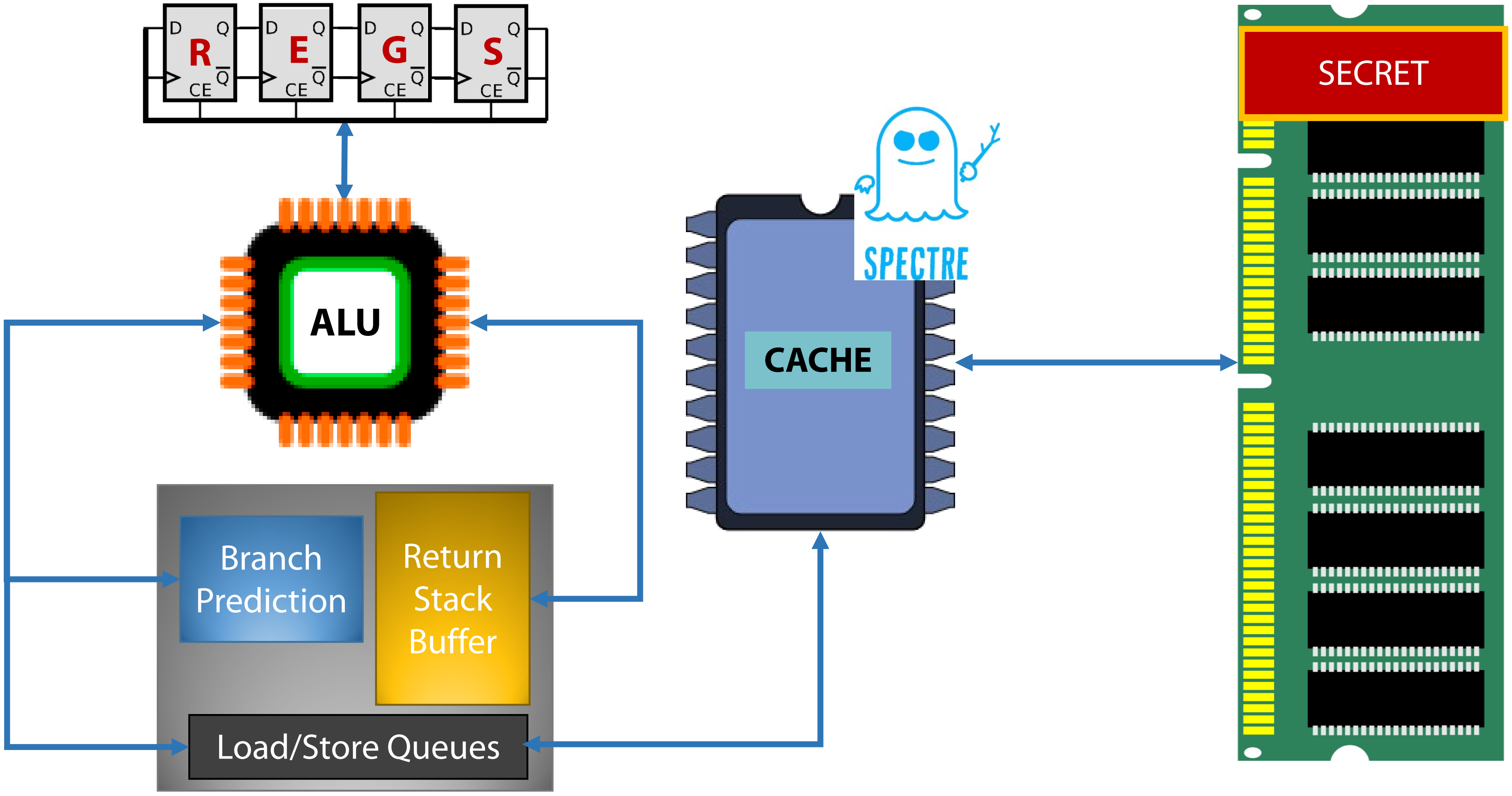
Selective Data Protection

DynPTA: Combining Static and Dynamic Analysis for Practical Selective Data Protection – IEEE S&P 2021

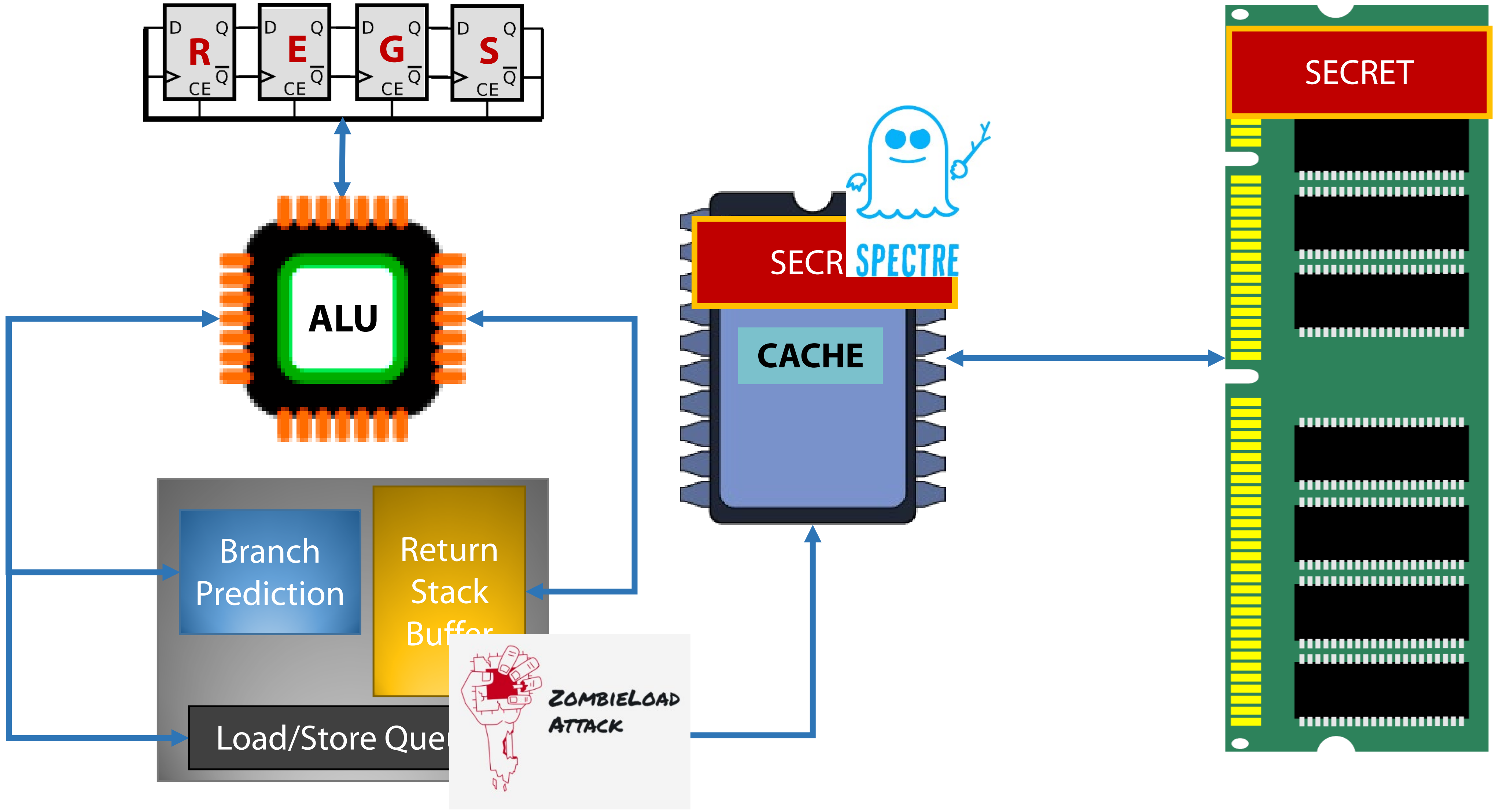
xMP: Selective Memory Protection for Kernel and User Space – IEEE S&P 2020

Mitigating Data Leakage by Protecting Memory-resident Sensitive Data – ACSAC 2019

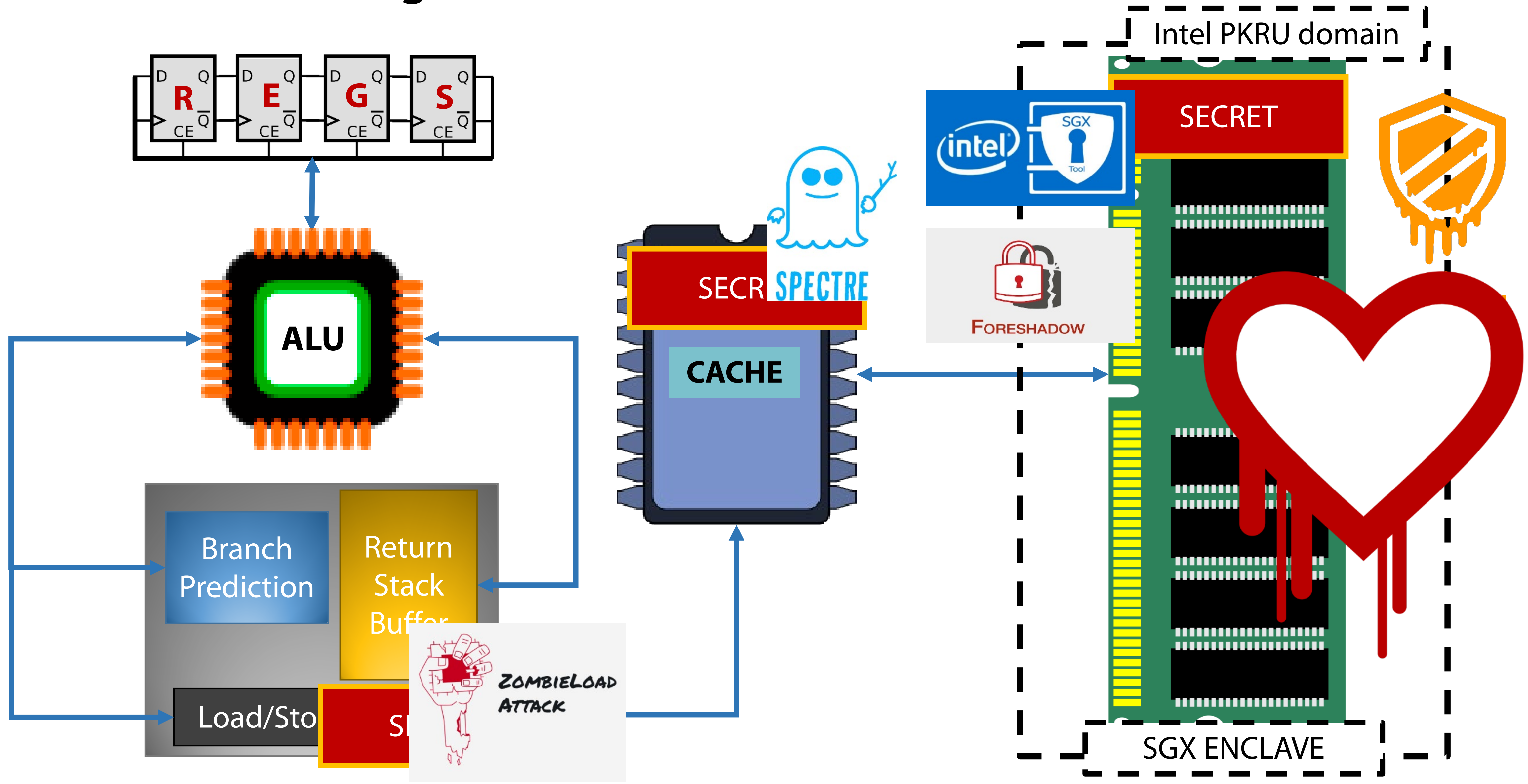
Process Data Leakage



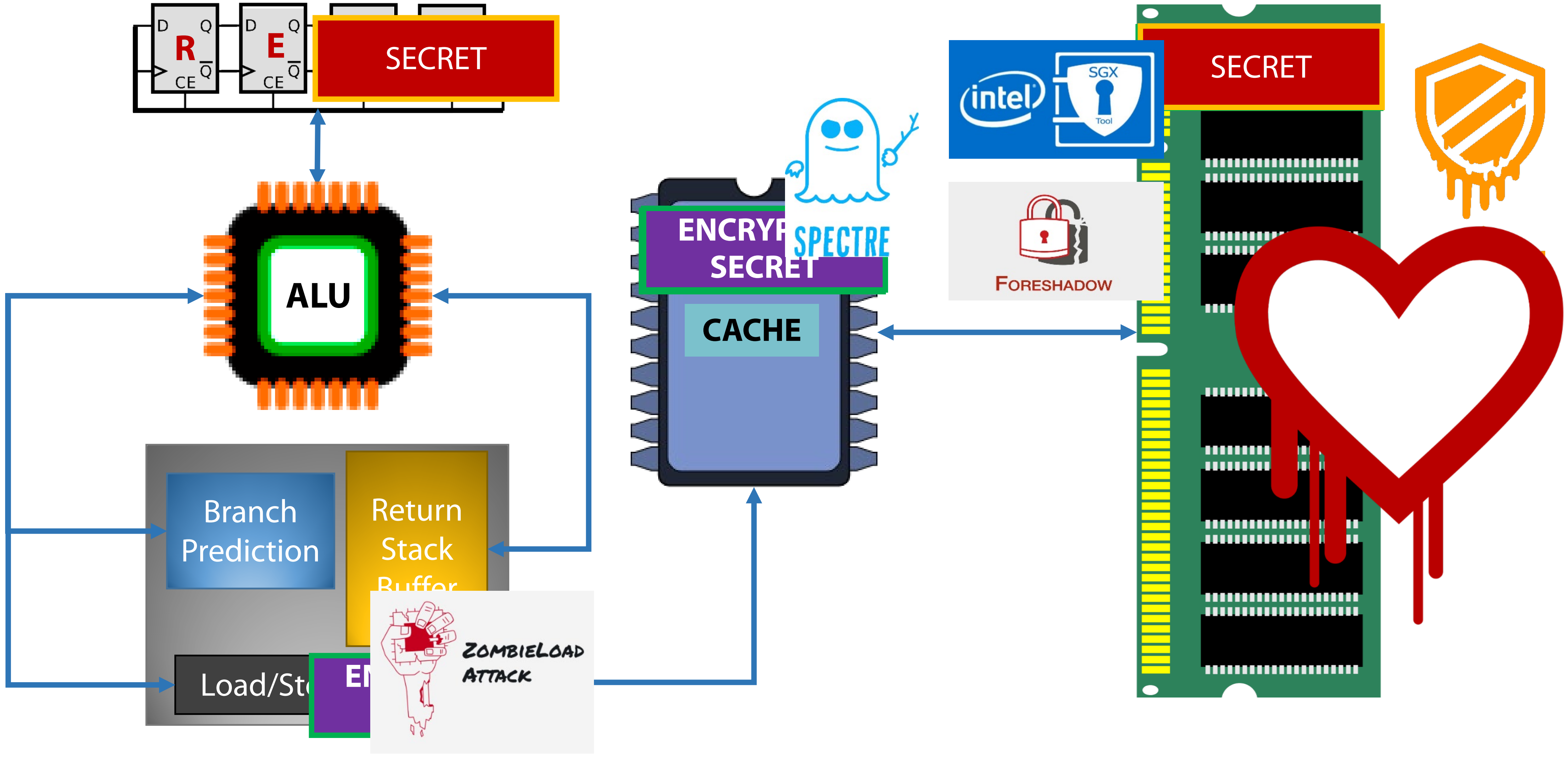
Process Data Leakage



Process Data Leakage



In-memory Data Encryption



Keeping In-Memory Data Encrypted [ACSAC '19]

Memory accesses must be instrumented at various program points

Example: Protect all accesses to PRIVATE_KEY

```
ptr = PRIVATE_KEY;

if (a > b) {
  d = 10 + c;
  *ptr = d;
}
```

```
ptr = PRIVATE_KEY;

if (a > b) {
  d = 10 + c;
  *ptr = d;
}
```

```
ptr = PRIVATE_KEY;

if (a > b) {
  d = 10 + c;
  encrypt(ptr, d);
}
```

Challenge: static (points-to) analysis is imprecise and leads to unnecessary memory encryption operations

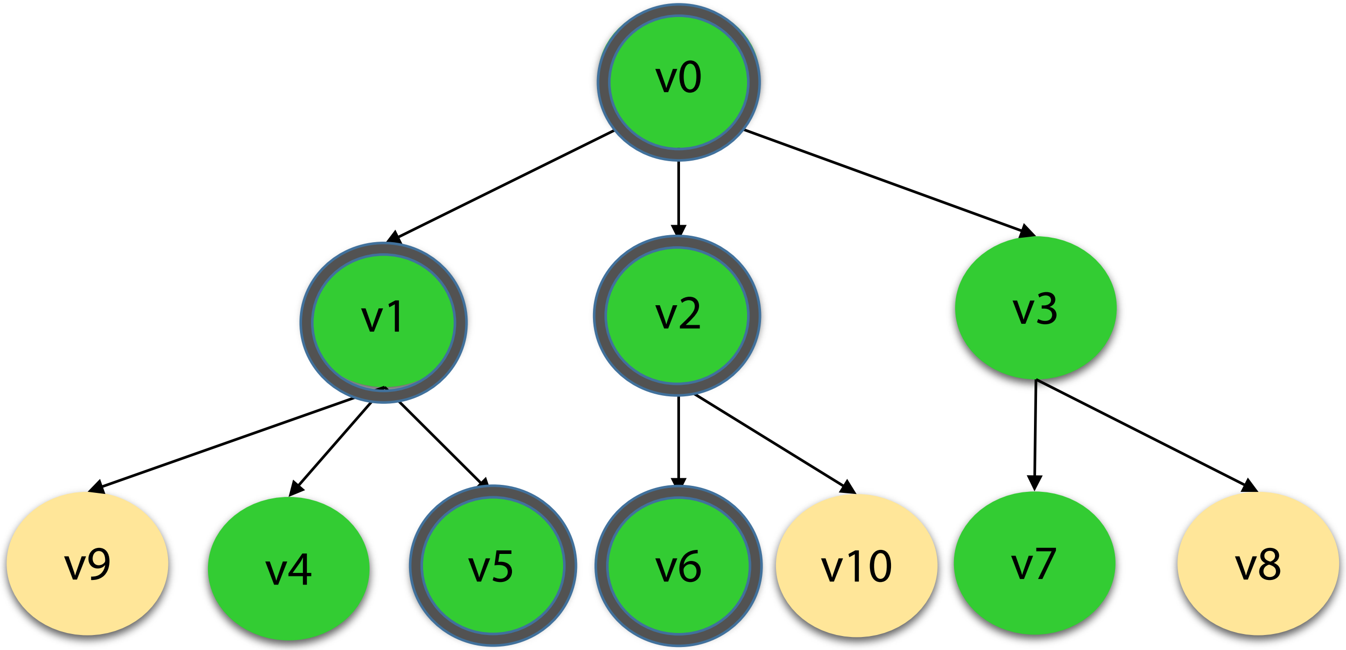
*Need a **sound** and **scalable** way to automatically instrument software*

DynPTA: Combining Static and Dynamic Analysis [IEEE S&P '21]

Goal: Identify all memory operations that need to be transformed

Static analysis:
Sound but imprecise

Dynamic analysis:
Precise but unsound



Best of both worlds: static analysis to ensure all code is covered, dynamic analysis to elide expensive instrumentation

Summary

Reduce the attack surface through software specialization

Prevent data leakage through selective in-memory data encryption

Open-source prototypes

- https://github.com/shamedgh/confine
- https://github.com/shamedgh/temporal-specialization
- https://github.com/taptipalit/dynpta

Stony Brook University

HEXLAB

DARPA YFA D18AP00045: Reducing Software Attack Surface through Compiler-Rewriter Cooperation

ONR N00014-17-1-2891: Multi-layer Software Transformation for Attack Surface Reduction and Shielding

NSF CNS-1749895: CAREER: Principled and Practical Software Shielding against Advanced Exploits