Why is memory safety still a concern?

Mohamed (Tarek Ibn Ziad) Hassan

https://www.cs.columbia.edu/~mtarek/

@M_TarekIbnZiad

Ph.D. Candidacy Exam

April 9th, 2020.

Why is memory safety still a concern?

Mohamed (Tarek Ibn Ziad) Hassan

https://www.cs.columbia.edu/~mtarek/

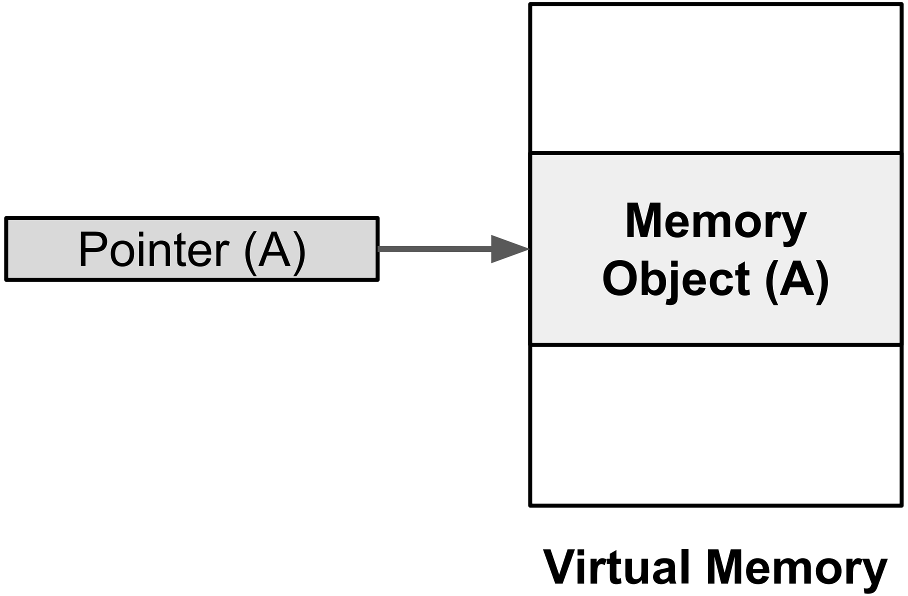
mtarek@cs.columbia.edu

Ph.D. Candidacy Exam

April 9th, 2020.

MEMORY SAFETY DEFINITION

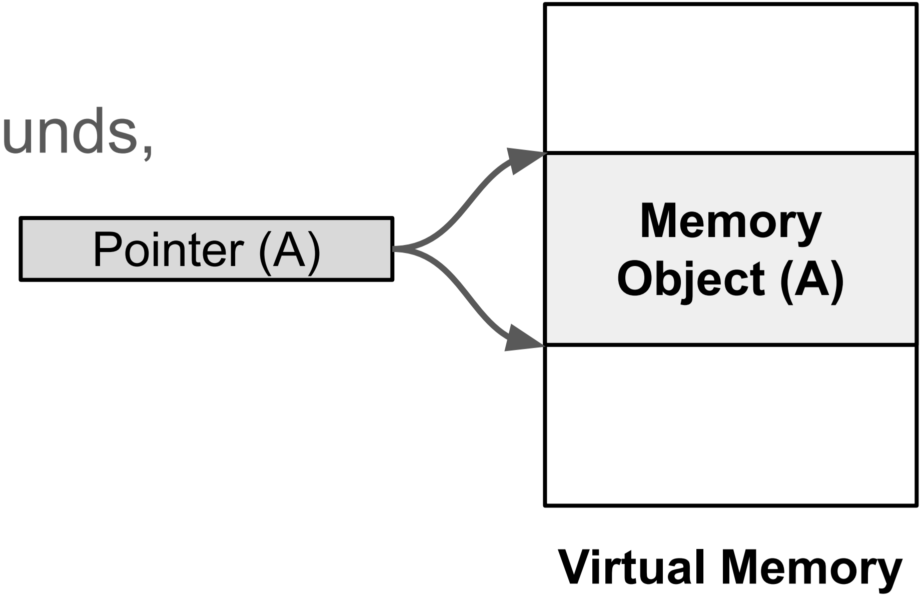
A program property that guarantees **memory objects** can only be accessed:



MEMORY SAFETY DEFINITION

A program property that guarantees **memory objects** can only be accessed:

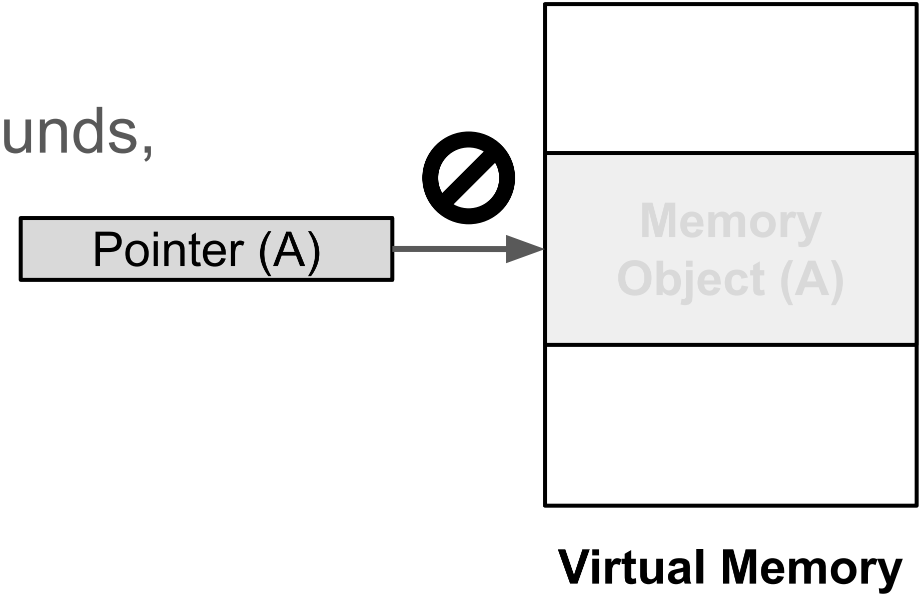
- Between their intended bounds,



MEMORY SAFETY DEFINITION

A program property that guarantees **memory objects** can only be accessed:

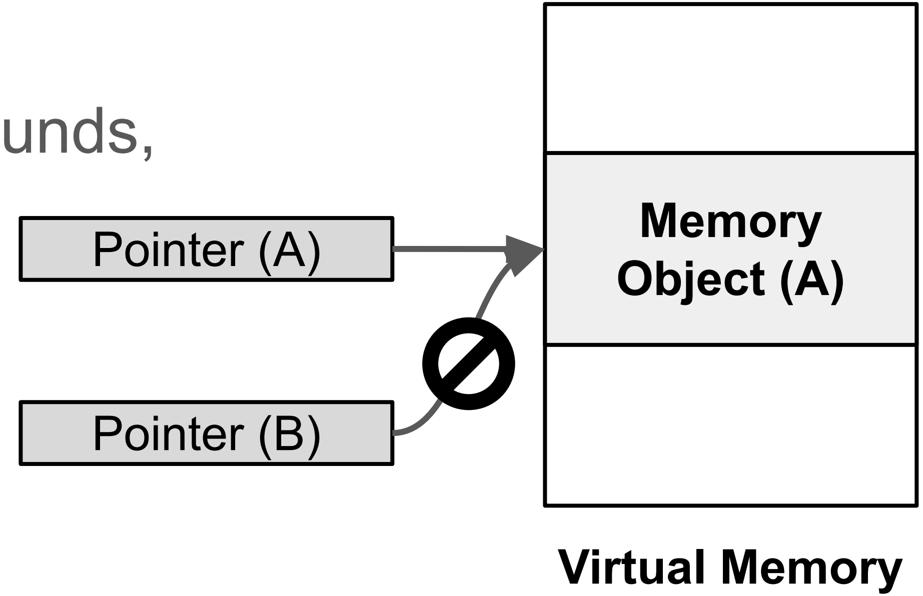
- Between their intended bounds,
- During their lifetime,



MEMORY SAFETY DEFINITION

A program property that guarantees **memory objects** can only be accessed:

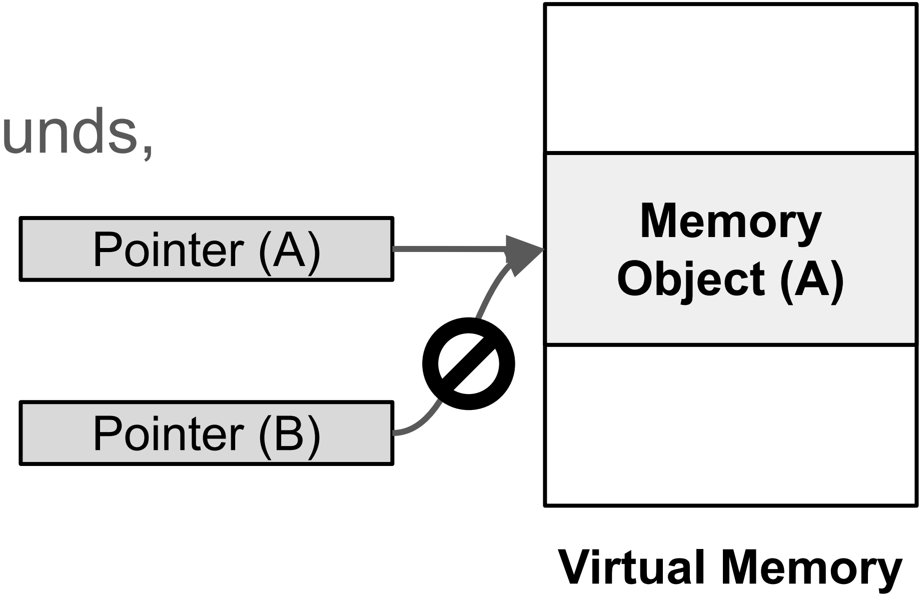
- Between their intended bounds,
- During their lifetime, and
- Given their original (or compatible) type.



MEMORY SAFETY ~~DEFINITION~~ VIOLATIONS

A program property that guarantees **memory objects** can only be accessed:

- Between their intended bounds,
- During their lifetime, and
- Given their original (or compatible) type.



MEMORY SAFETY ~~DEFINITION~~ VIOLATIONS

A program property that guarantees **memory objects** can only be accessed:

- ~~Between their intended bounds~~, Buffer overflow
- During their lifetime, and
- Given their original (or compatible) type.

MEMORY SAFETY ~~DEFINITION~~ VIOLATIONS

A program property that guarantees **memory objects** can only be accessed:

- Between their intended bounds,
- ~~During their lifetime, and~~ Use-after-free
- Given their original (or compatible) type.

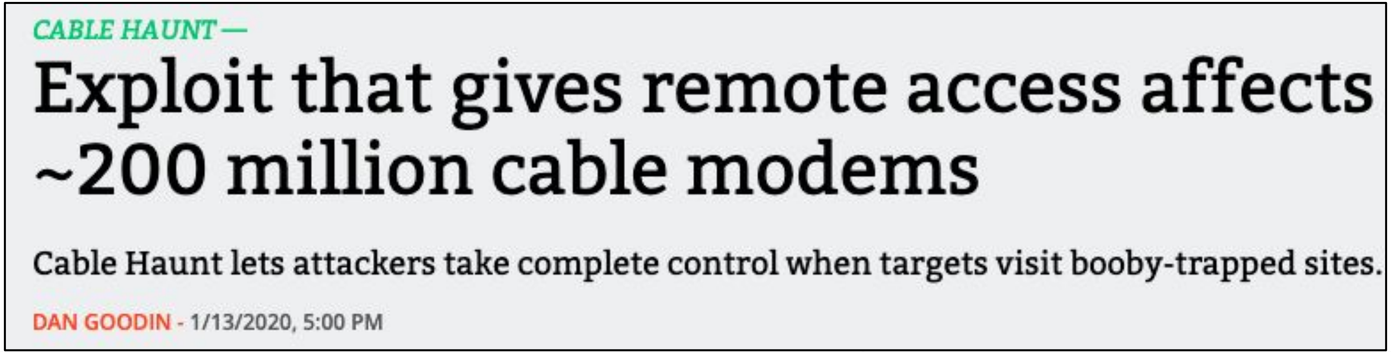
MEMORY SAFETY ~~DEFINITION~~ VIOLATIONS

A program property that guarantees **memory objects** can only be accessed:

- Between their intended bounds,
- During their lifetime, and
- ~~Given their original (or compatible) type.~~

Type confusion

MEMORY SAFETY IS A SERIOUS PROBLEM!

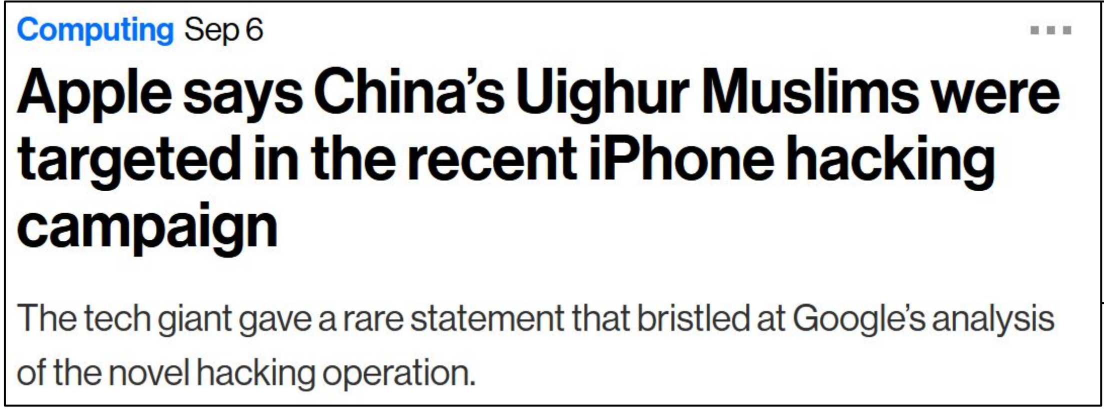


CABLE HAUNT —

Exploit that gives remote access affects ~200 million cable modems

Cable Haunt lets attackers take complete control when targets visit booby-trapped sites.

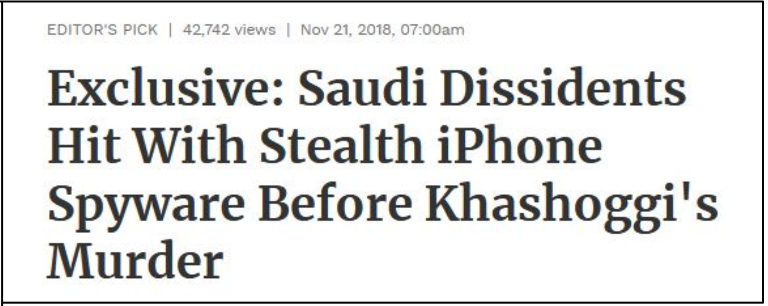
DAN GOODIN - 1/13/2020, 5:00 PM



Computing Sep 6

Apple says China’s Uighur Muslims were targeted in the recent iPhone hacking campaign

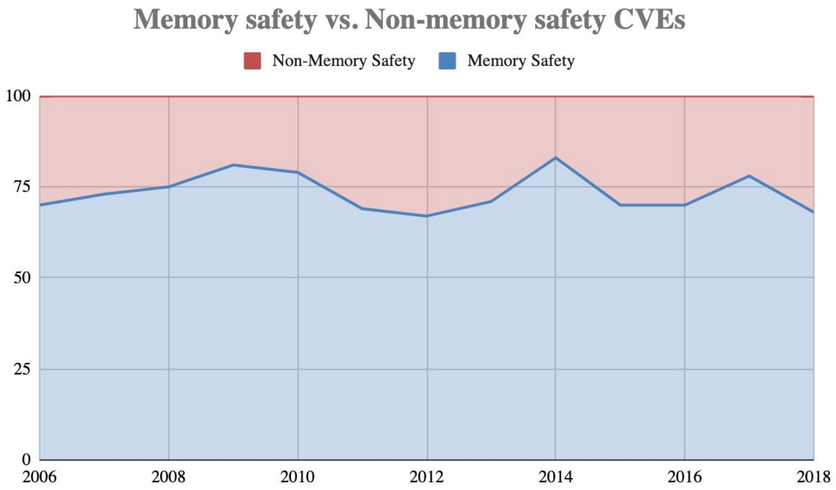
The tech giant gave a rare statement that bristled at Google’s analysis of the novel hacking operation.



EDITOR'S PICK | 42,742 views | Nov 21, 2018, 07:00am

Exclusive: Saudi Dissidents Hit With Stealth iPhone Spyware Before Khashoggi's Murder

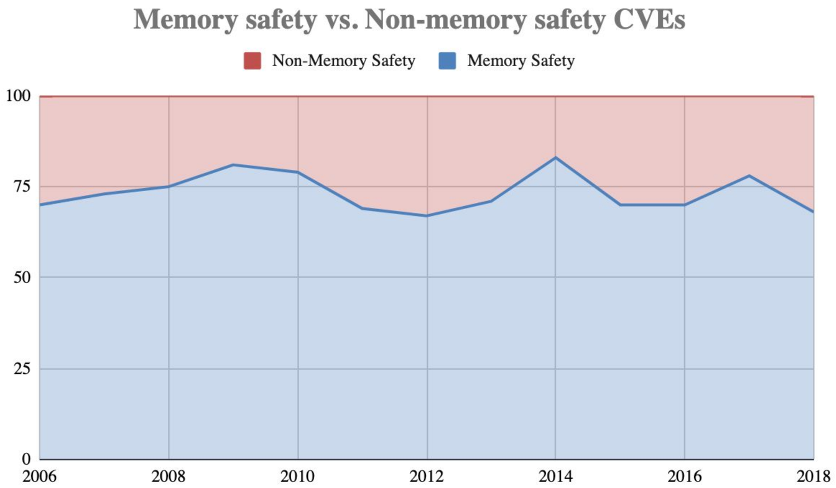
PREVALENCE OF MEMORY SAFETY VULNS



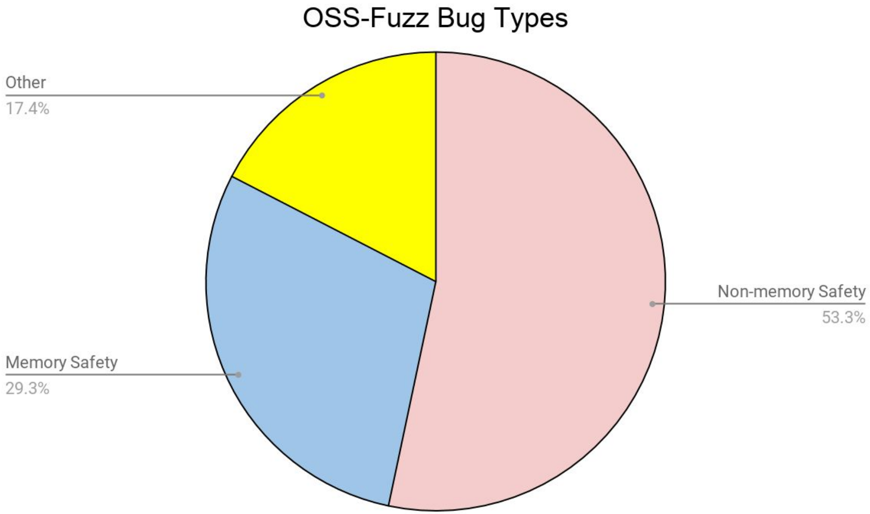
Microsoft Product CVEs

Source: Matt Miller, Microsoft Security Response Center (MSRC) **[BlueHat 2019]**

PREVALENCE OF MEMORY SAFETY VULNS



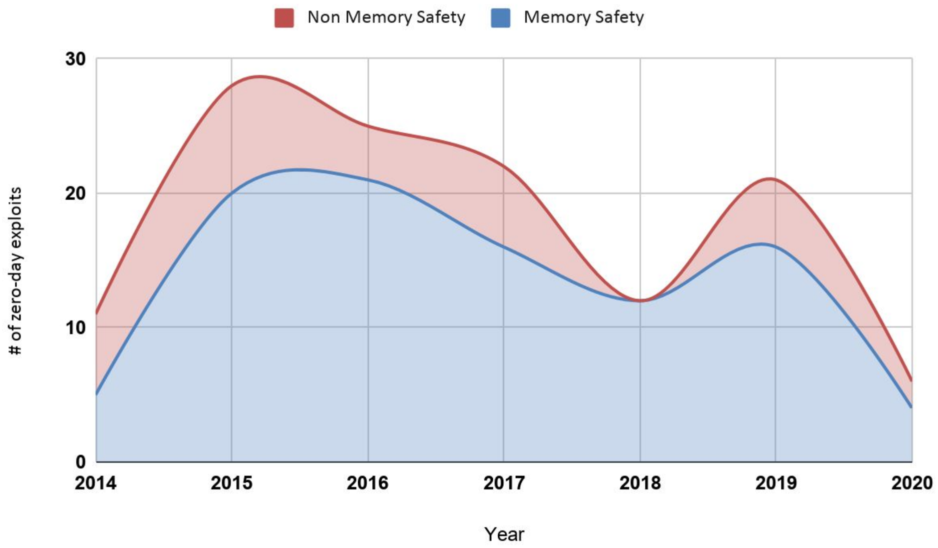
Microsoft Product CVEs



Google OSS-Fuzz bugs from 2016-2018.

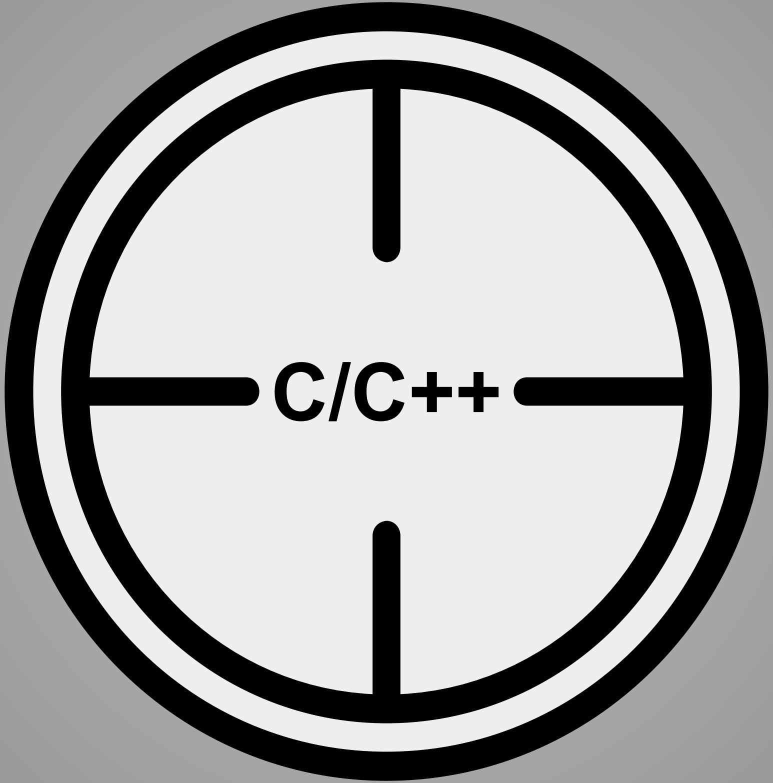
Source: https://security.googleblog.com/2018/11/a-new-chapter-for-oss-fuzz.html

ATTACKERS PREFER MEMORY SAFETY VULNS



Zero-day “in the wild” exploits
from 2014-2020

Source: Google Project Zero, 0day "In the Wild" spreadsheet. Last updated: April 3rd, 2020



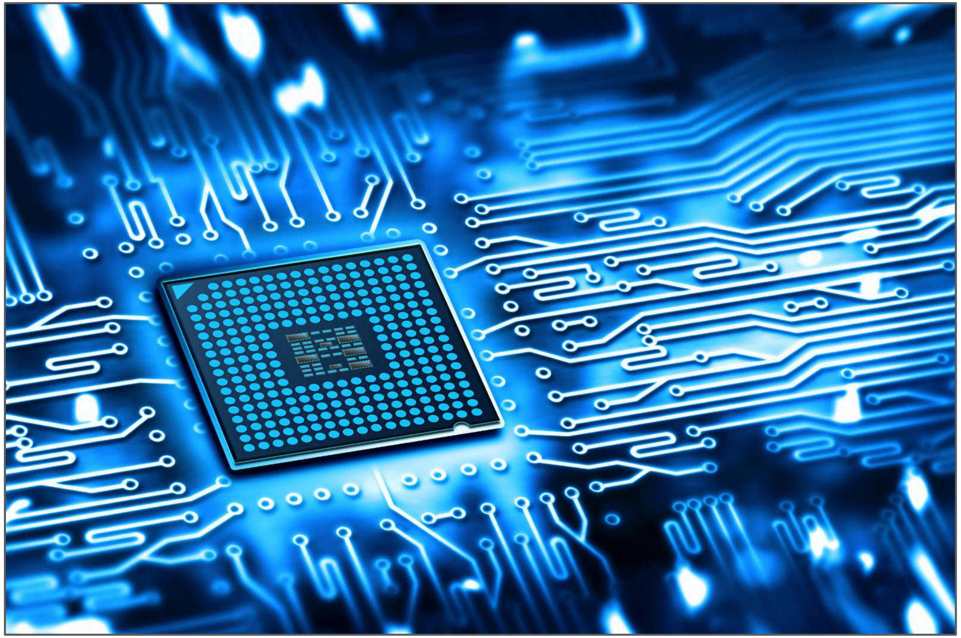
C/C++

C/C++ IS HERE TO STAY

- Performance.

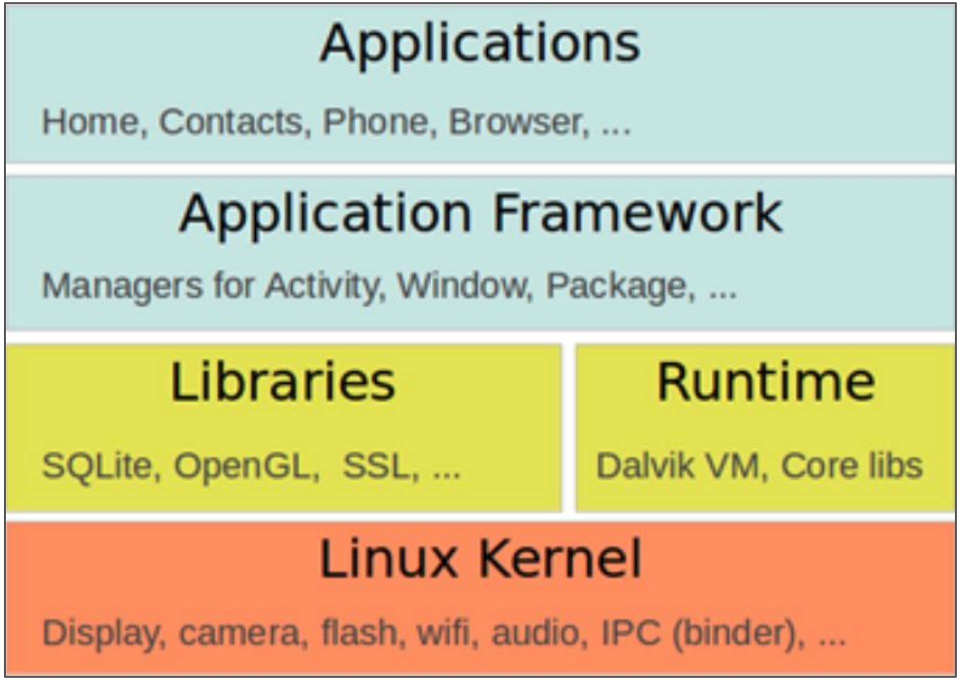
C/C++ IS HERE TO STAY

- Performance.
- Communication.



C/C++ IS HERE TO STAY

- Performance.
- Communication.
- Completeness.



Applications

Home, Contacts, Phone, Browser, ...

Application Framework

Managers for Activity, Window, Package, ...

Libraries

SQLite, OpenGL, SSL, ...

Runtime

Dalvik VM, Core libs

Linux Kernel

Display, camera, flash, wifi, audio, IPC (binder), ...

C/C++ IS HERE TO STAY

- Performance.
- Communication.
- Completeness.
- Maturity.

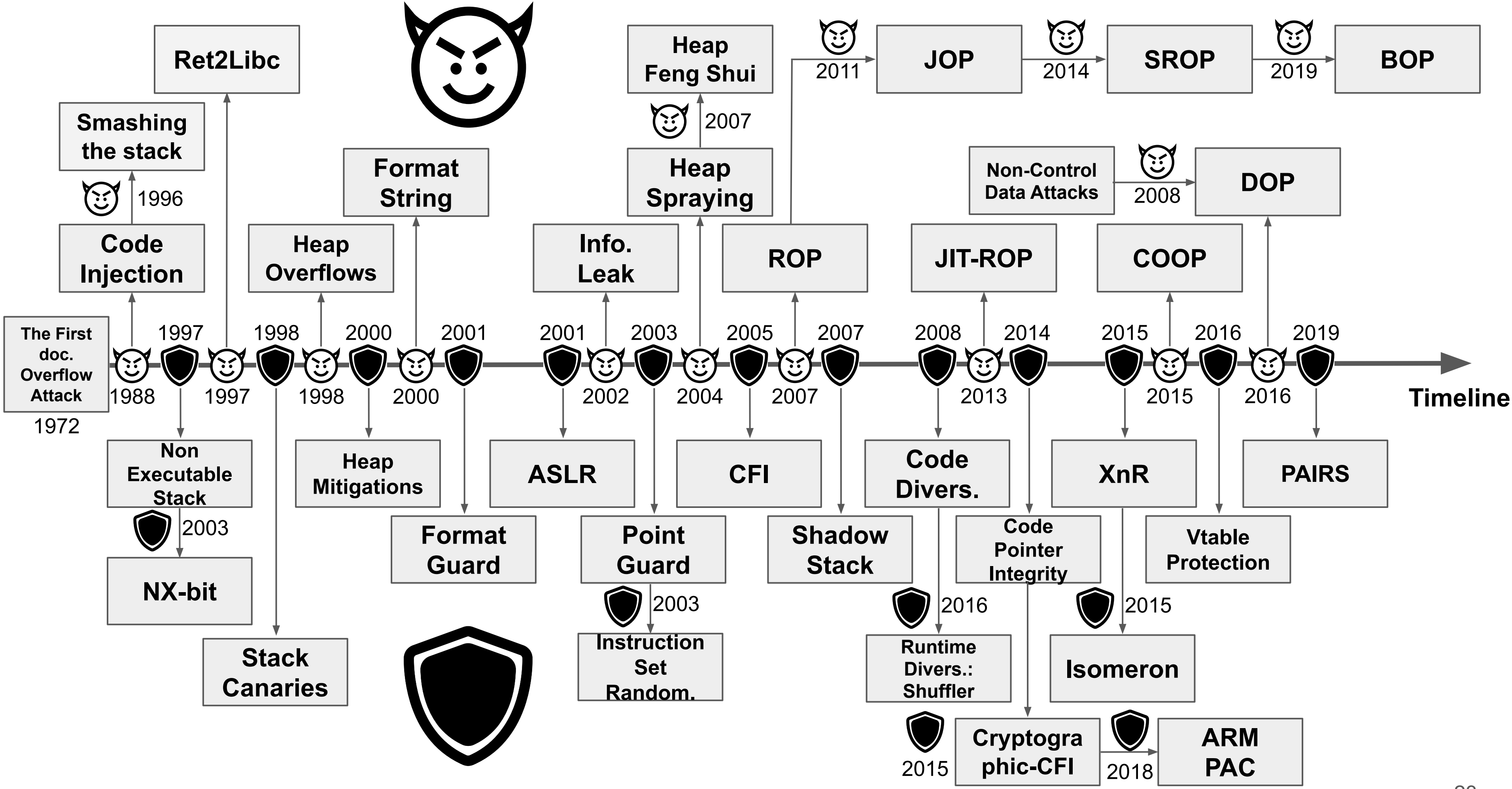
C/C++ IS HERE TO STAY

- Performance.
- Communication.
- Completeness.
- Maturity.
- Legacy code.

REST OF THE TALK

- ⚔ Memory Corruption Attacks & Defenses
- 🏰 Memory Safety Techniques
- ⏳ Future Work Map

Memory Corruption Attacks & Defenses



The First doc. Overflow Attack
1972

1988
Code Injection
1996
Smashing the stack

1997
Non Executable Stack
2003
NX-bit

1997
Ret2Libc

1998
Stack Canaries

1998
Heap Overflows

2000
Heap Mitigations

2000
Format String

2001
Format Guard

2001
ASLR

2002
Info. Leak

2003
Point Guard
2003
Instruction Set Random.

2004
Heap Spraying
2007
Heap Feng Shui

2005
CFI

2007
ROP
2011
JOP
2014
SROP
2019
BOP

2007
Shadow Stack

2008
Code Divers.
2016
Runtime Divers.: Shuffler

2013
JIT-ROP

2014
Code Pointer Integrity
2015
Cryptographic-CFI
2018
ARM PAC

2015
XnR
2015
Isomeron

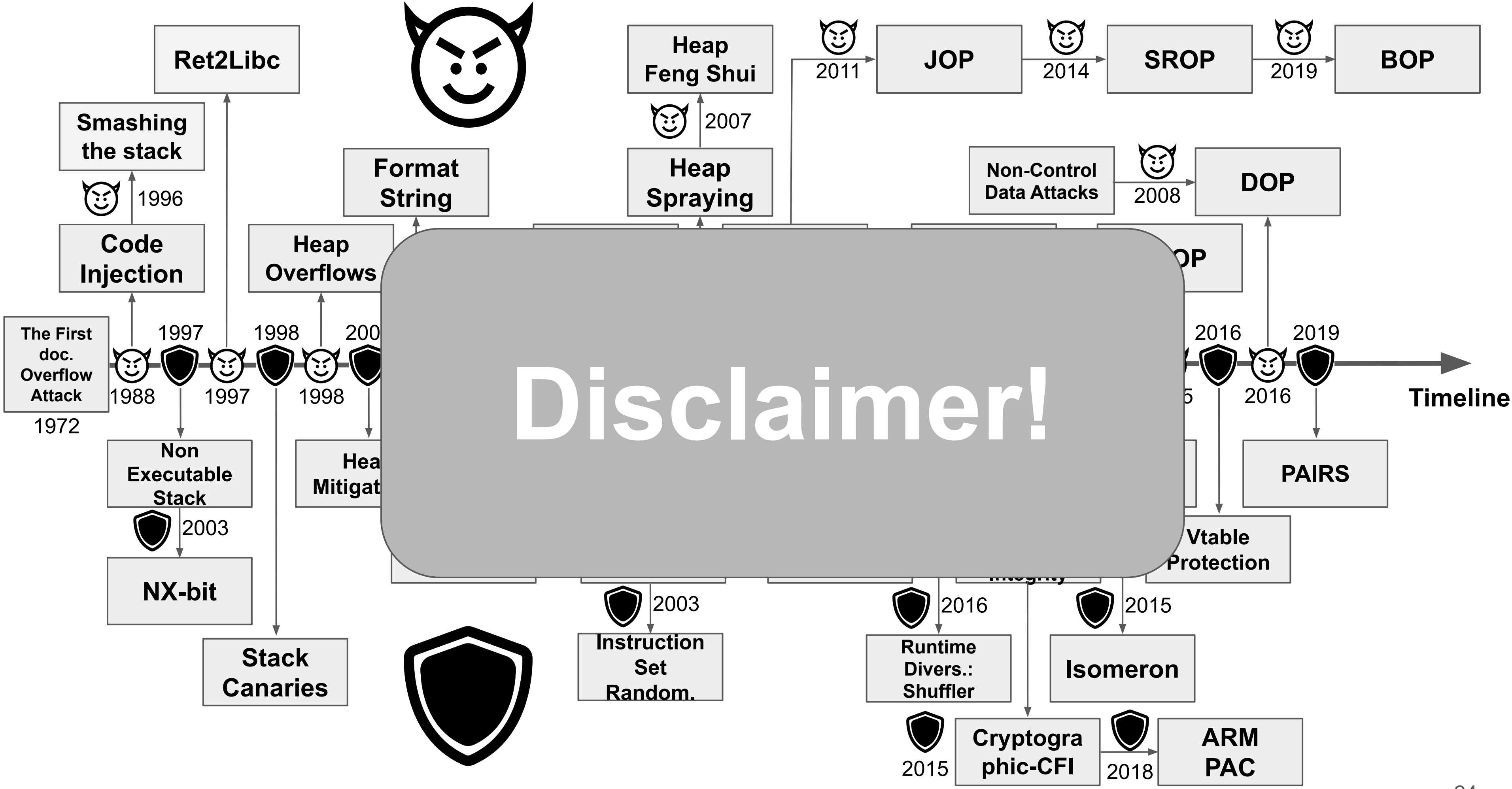
2015
COOP

2016
Vtable Protection

2016
Non-Control Data Attacks
2008
DOP

2019
PAIRS

Timeline



Disclaimer!

The First doc. Overflow Attack

1972

1988

Code Injection

1996

Smashing the stack

1997

Ret2Libc

1997

Non Executable Stack

2003

NX-bit

1998

Stack Canaries

1998

Heap Overflows

200

Hea Mitigat

Format String

Heap Spraying

2007

Heap Feng Shui

2003

Instruction Set Random.

2011

JOP

2014

SROP

2019

BOP

Non-Control Data Attacks

2008

DOP

OP

5

2016

2016

2019

PAIRS

Vtable Protection

2016

Runtime Divers.: Shuffler

2015

Isomeron

2015

Cryptographic-CFI

2018

ARM PAC

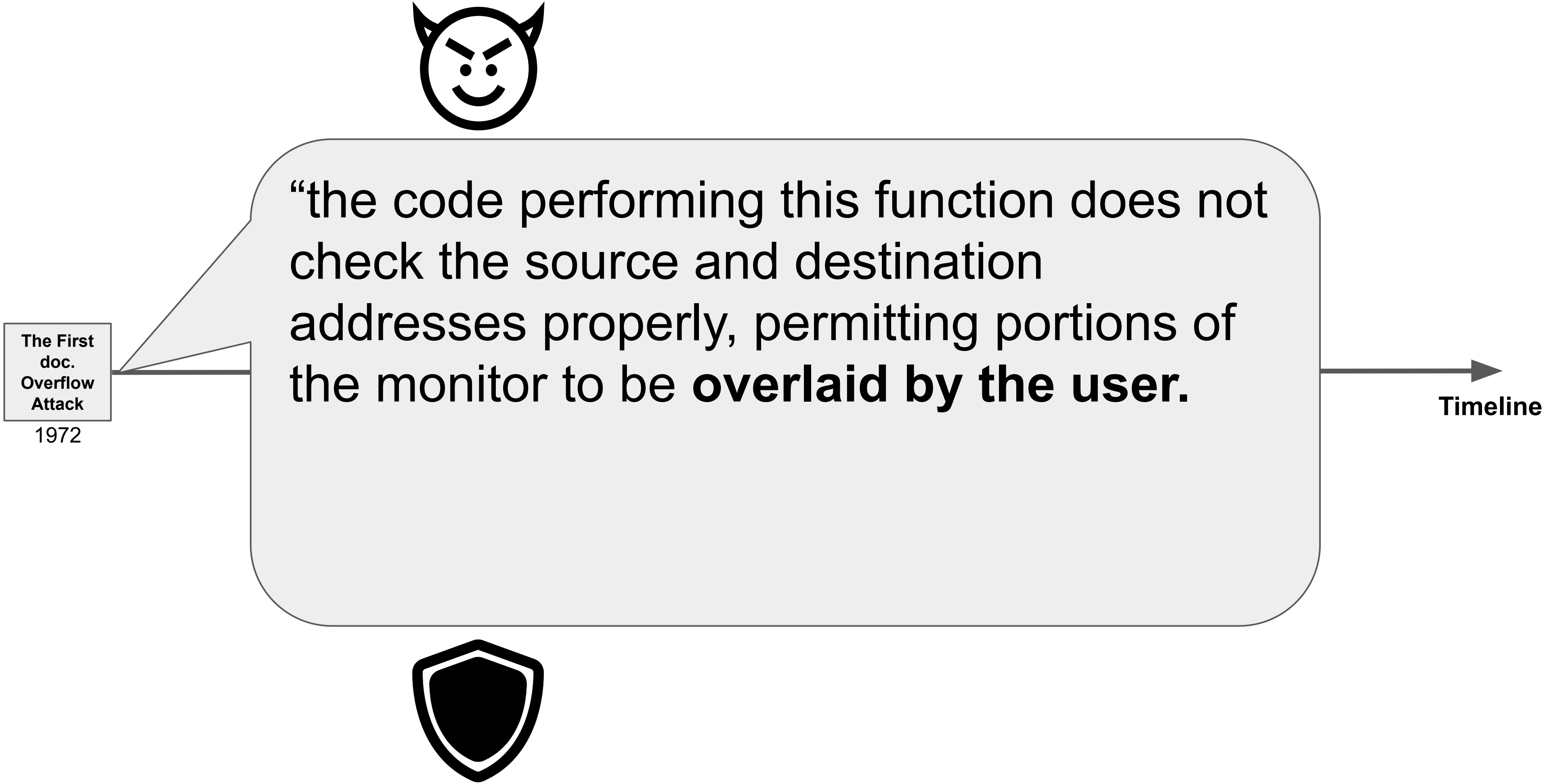
Timeline

The First doc. Overflow Attack

1972

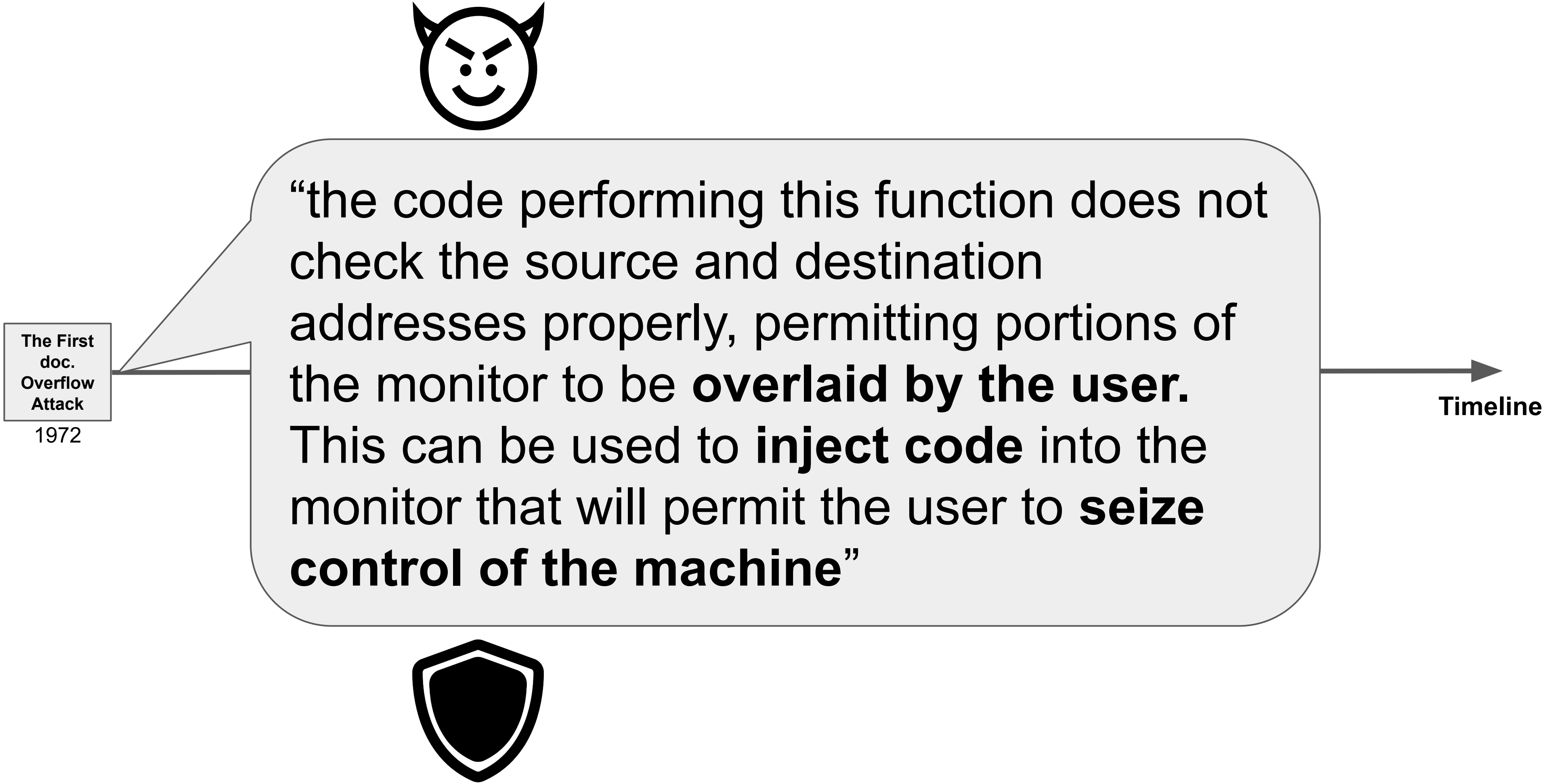
Timeline

Source: James P. Anderson, Computer Security Technology Planning Study, October 1972
http://seclab.cs.ucdavis.edu/projects/history/papers/ande72.pdf



“the code performing this function does not check the source and destination addresses properly, permitting portions of the monitor to be **overlaid by the user.**

Source: James P. Anderson, Computer Security Technology Planning Study, October 1972
http://seclab.cs.ucdavis.edu/projects/history/papers/ande72.pdf



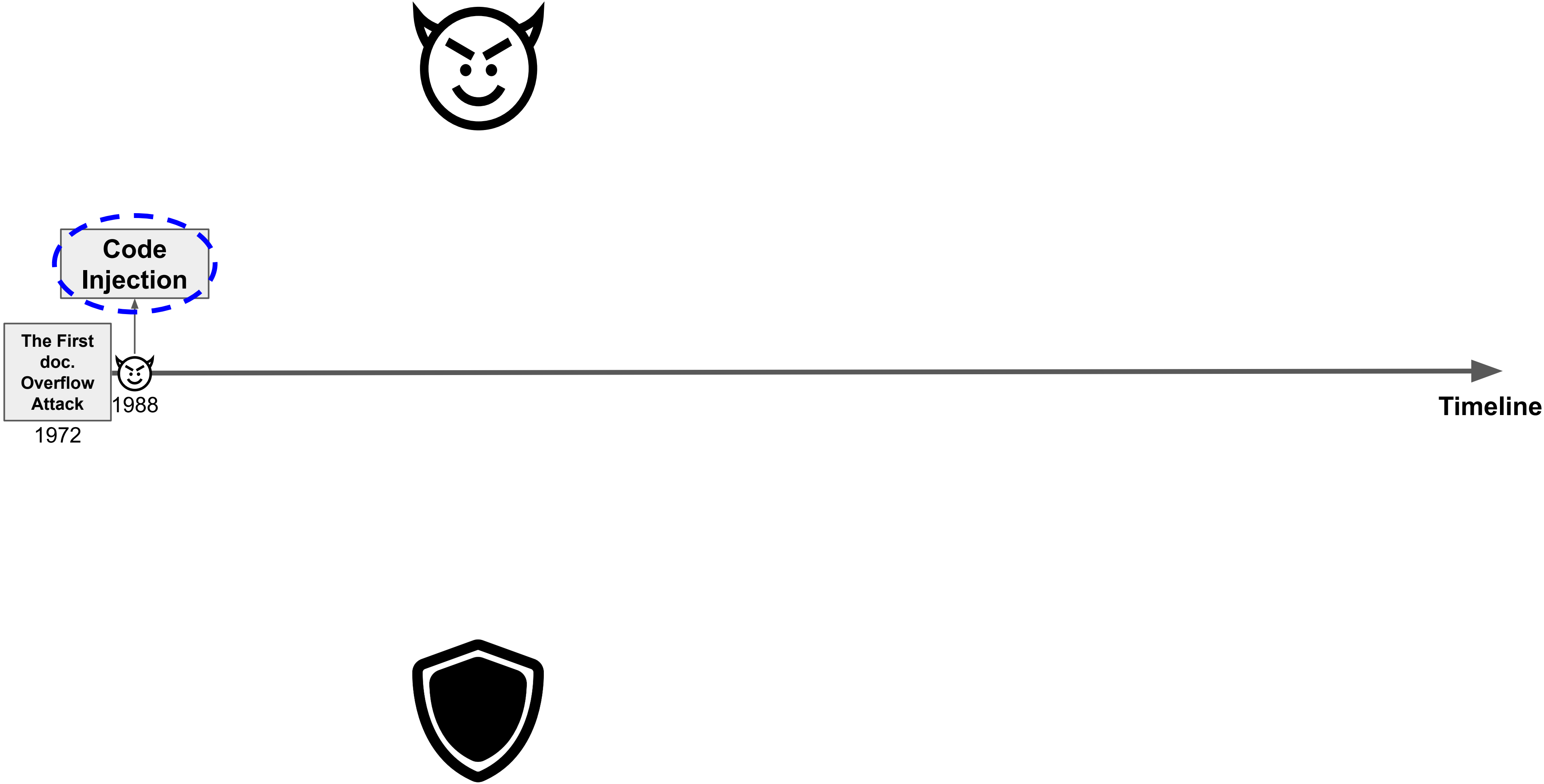
The First doc. Overflow Attack

1972

“the code performing this function does not check the source and destination addresses properly, permitting portions of the monitor to be **overlaid by the user.** This can be used to **inject code** into the monitor that will permit the user to **seize control of the machine**”

Timeline

Source: James P. Anderson, Computer Security Technology Planning Study, October 1972
http://seclab.cs.ucdavis.edu/projects/history/papers/ande72.pdf



Code Injection

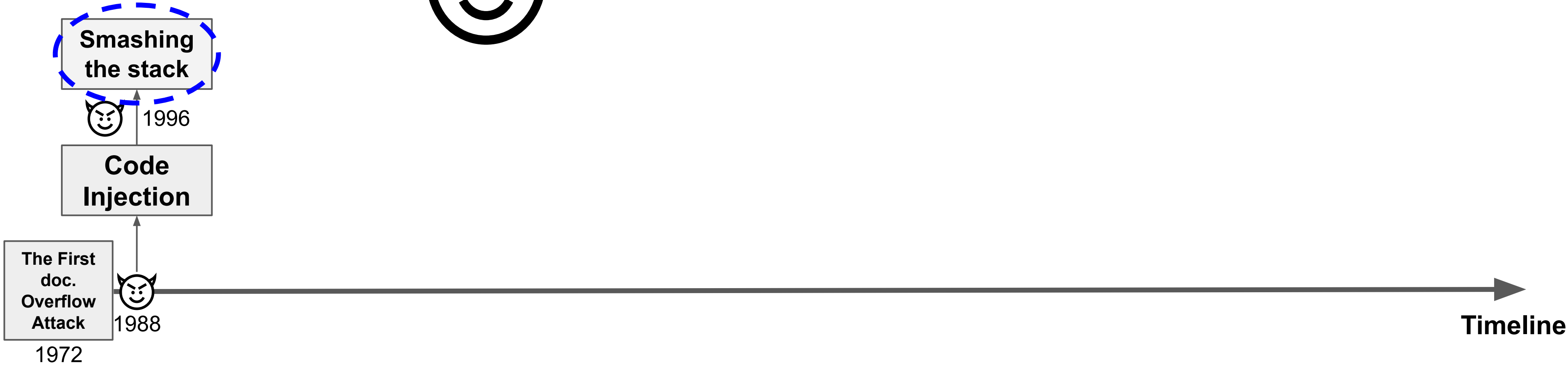
The First doc. Overflow Attack

1972

1988

Timeline

[4] Szekeres et. al, SoK: Eternal war in memory. **[S&P 2013]**



Smashing the stack

1996

Code Injection

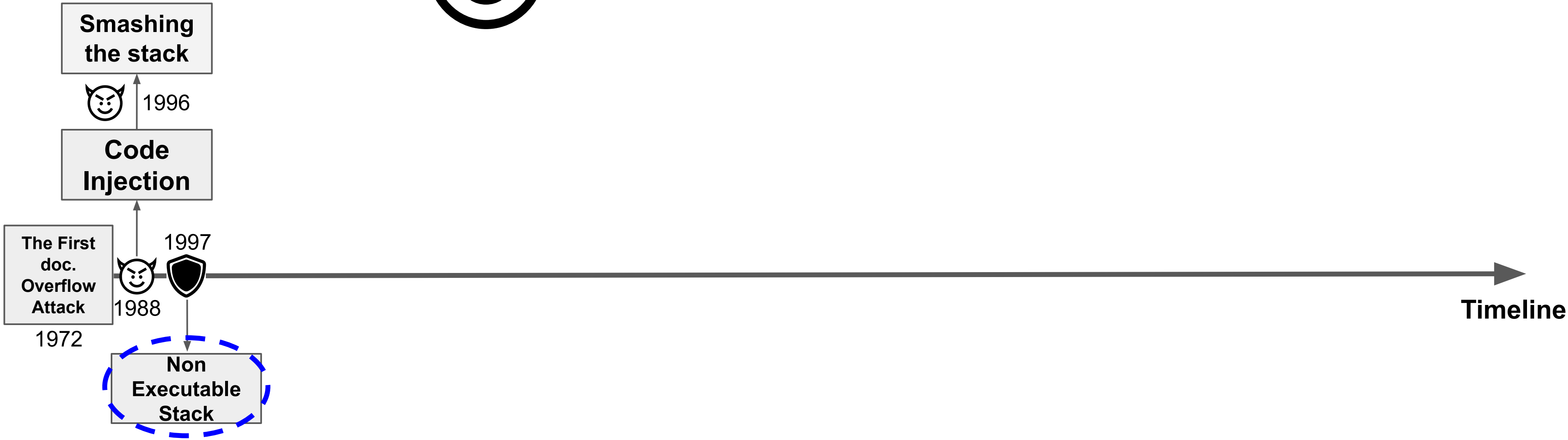
The First doc. Overflow Attack

1972

1988

Timeline

[3] Van der Veen et. al., Memory errors: The past, the present, and the future. **[RAID 2012]**



Smashing the stack

1996

Code Injection

The First doc. Overflow Attack

1972

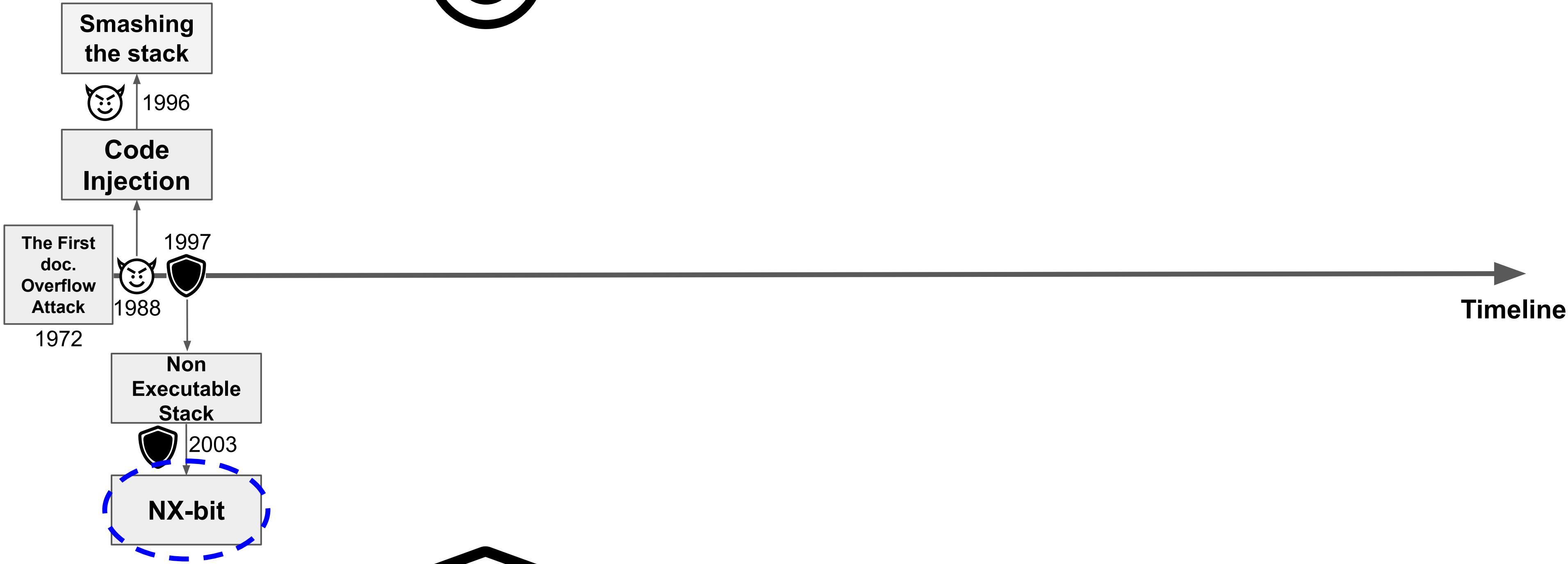
1988

1997

Non Executable Stack

Timeline

[3] Van der Veen et. al., Memory errors: The past, the present, and the future. **[RAID 2012]**



Smashing the stack

1996

Code Injection

The First doc. Overflow Attack

1972

1988

1997

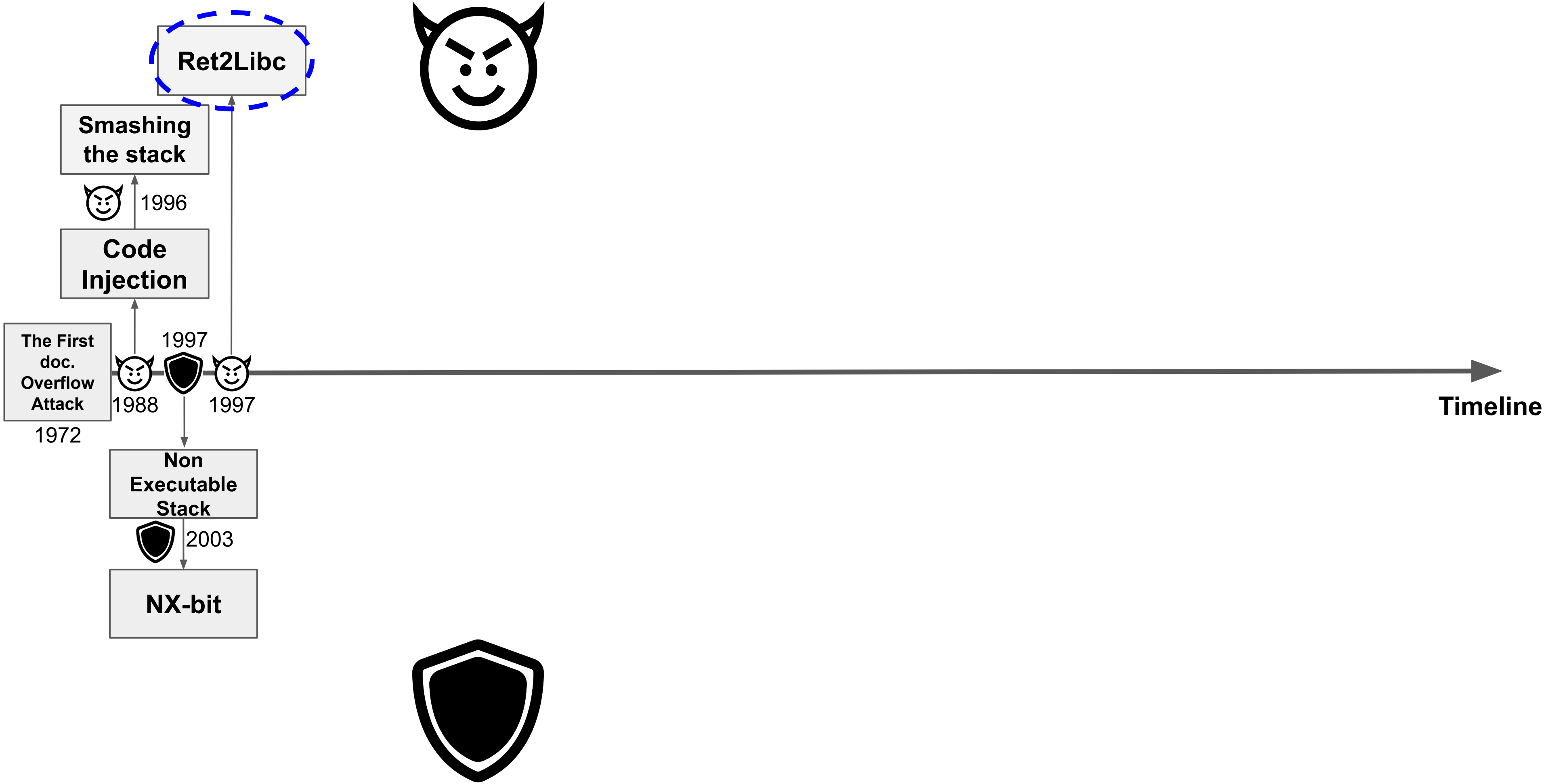
Timeline

Non Executable Stack

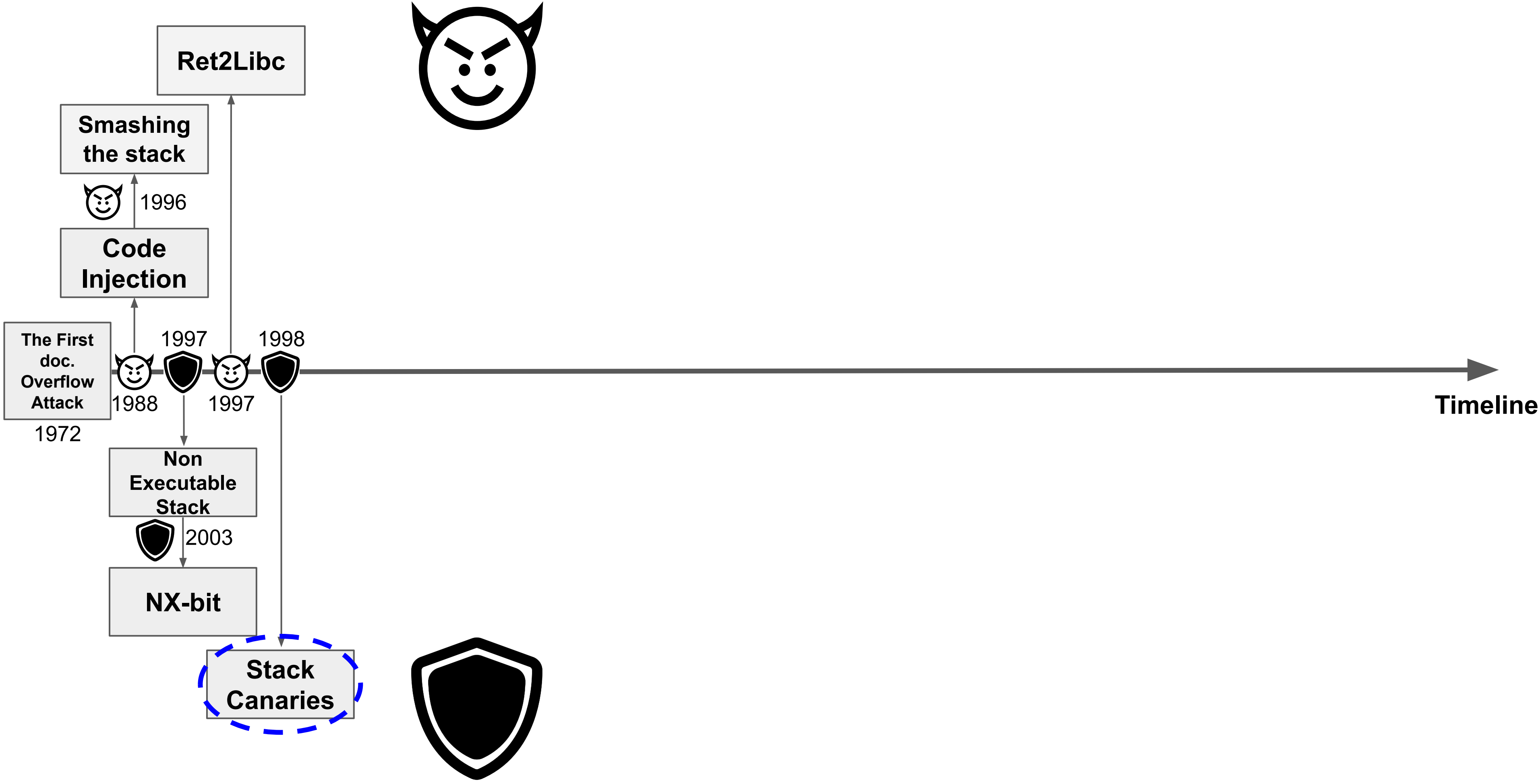
2003

NX-bit

[3] Van der Veen et. al., Memory errors: The past, the present, and the future. **[RAID 2012]**



[3] Van der Veen et. al., Memory errors: The past, the present, and the future. **[RAID 2012]**



Ret2Libc

Smashing the stack

1996

Code Injection

The First doc. Overflow Attack

1972

1988

1997

1997

1998

Timeline

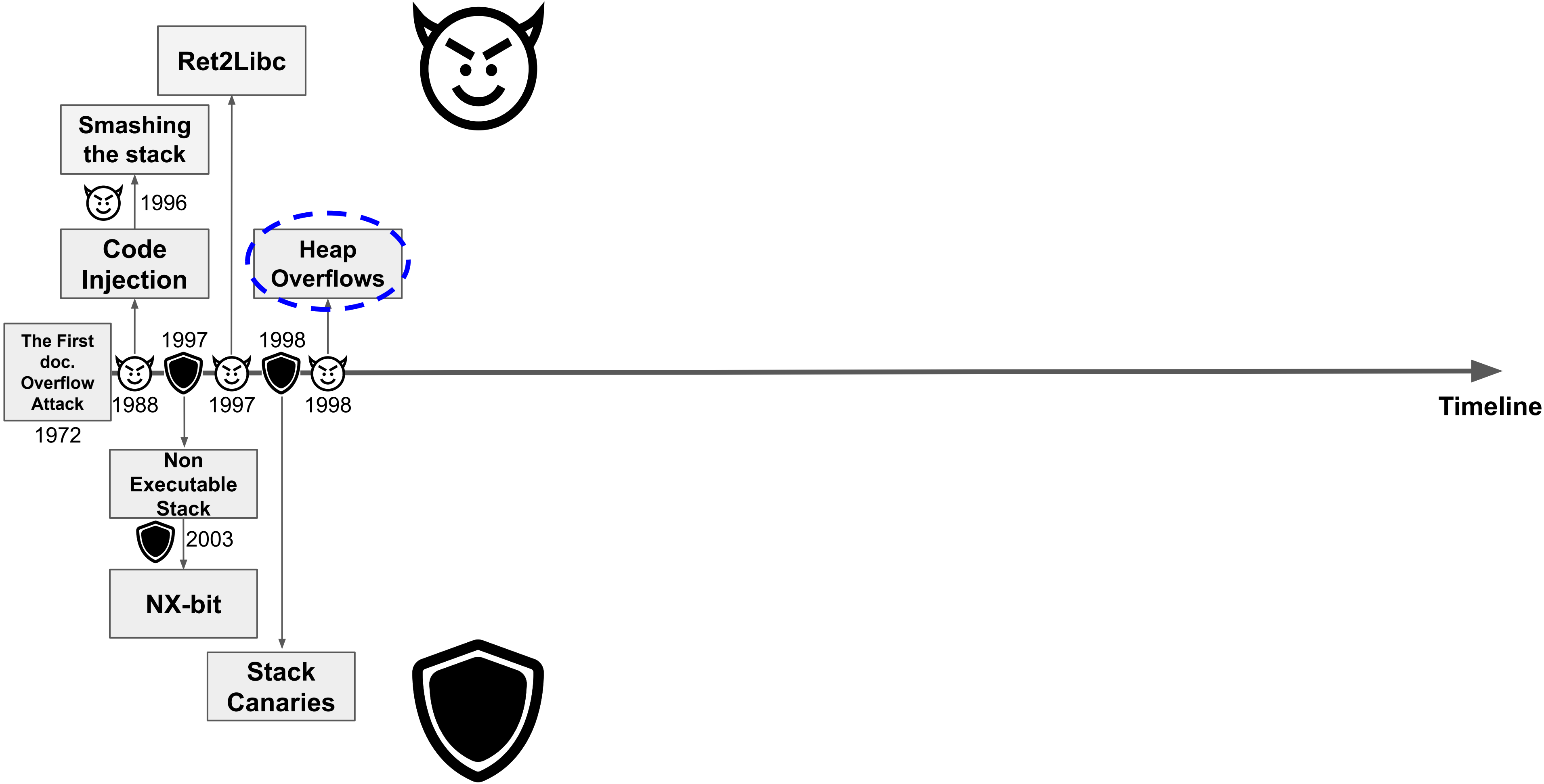
Non Executable Stack

2003

NX-bit

Stack Canaries

[3] Van der Veen et. al., Memory errors: The past, the present, and the future. **[RAID 2012]**



The First doc. Overflow Attack

1972

1988

Code Injection

1996

Smashing the stack

1997

Non Executable Stack

2003

NX-bit

1997

Ret2Libc

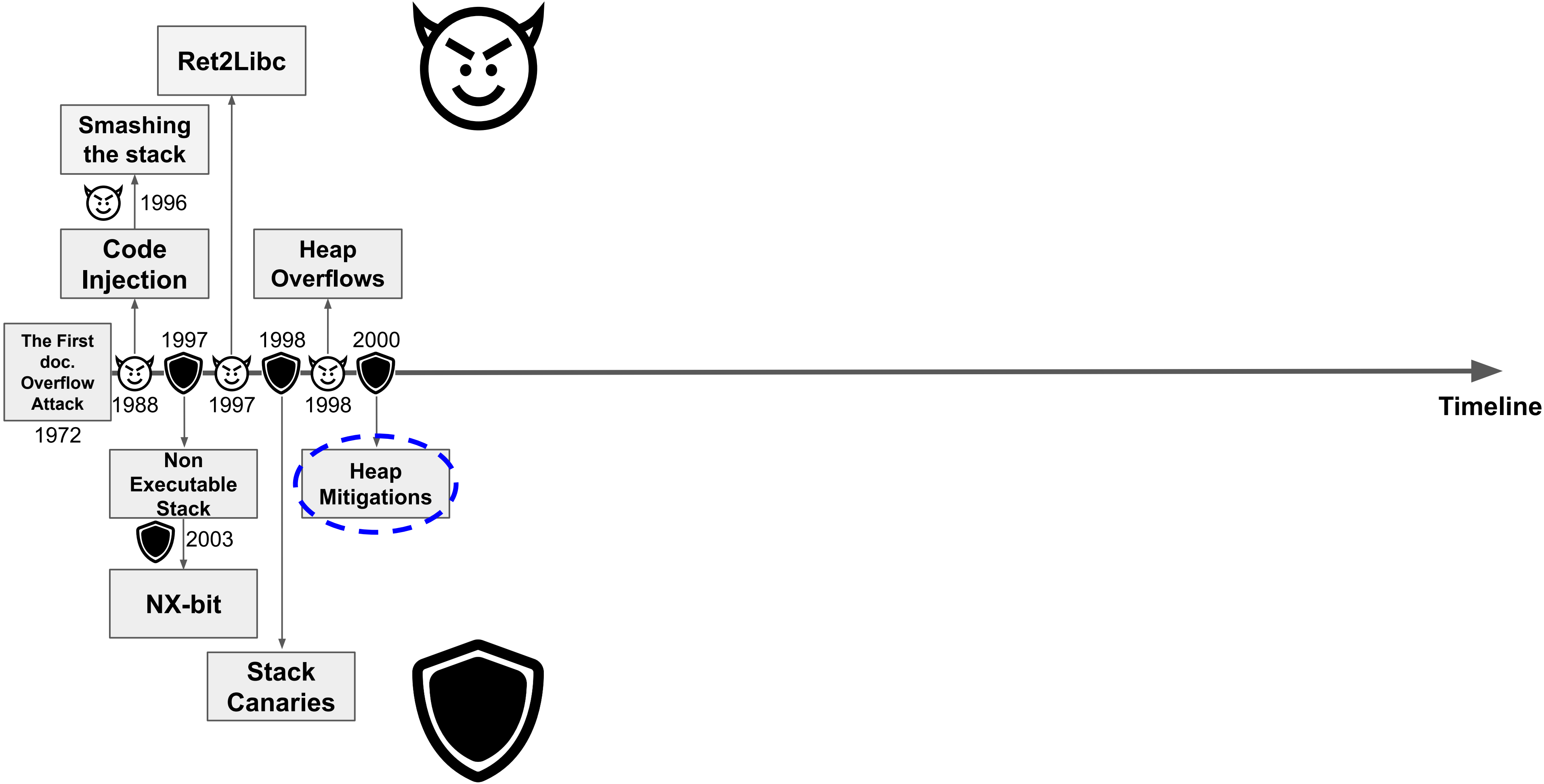
1998

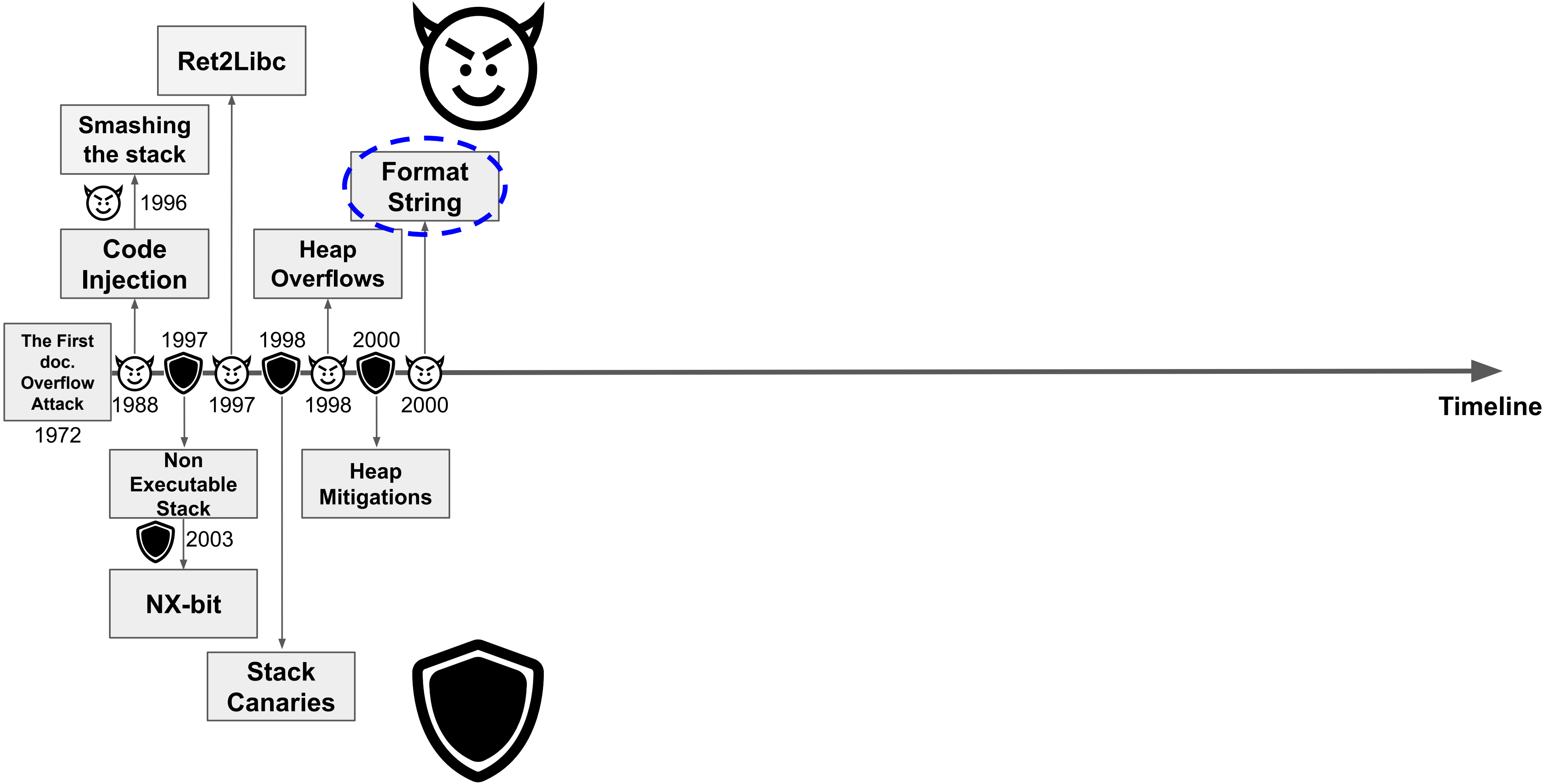
Stack Canaries

1998

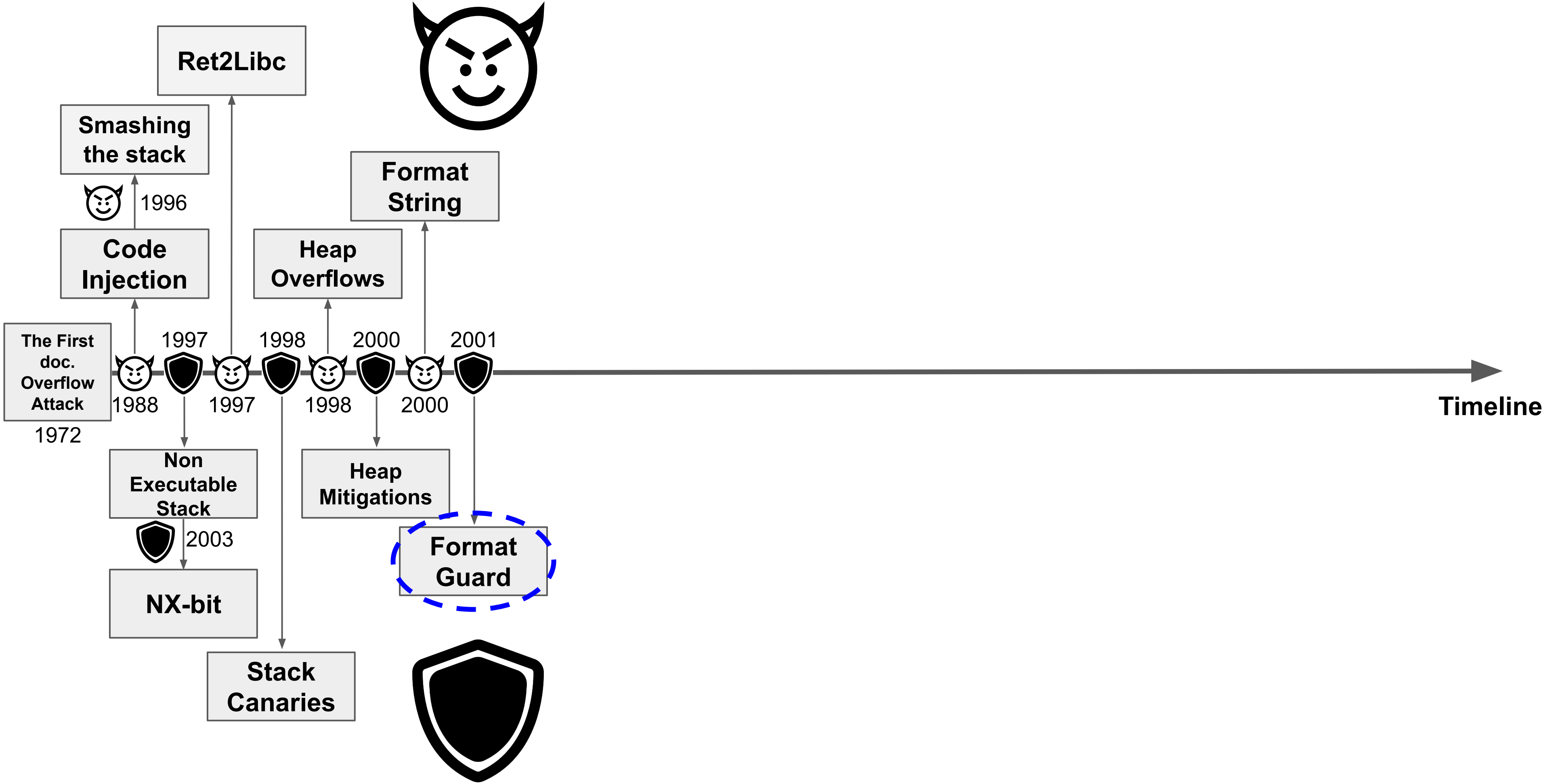
Heap Overflows

Timeline

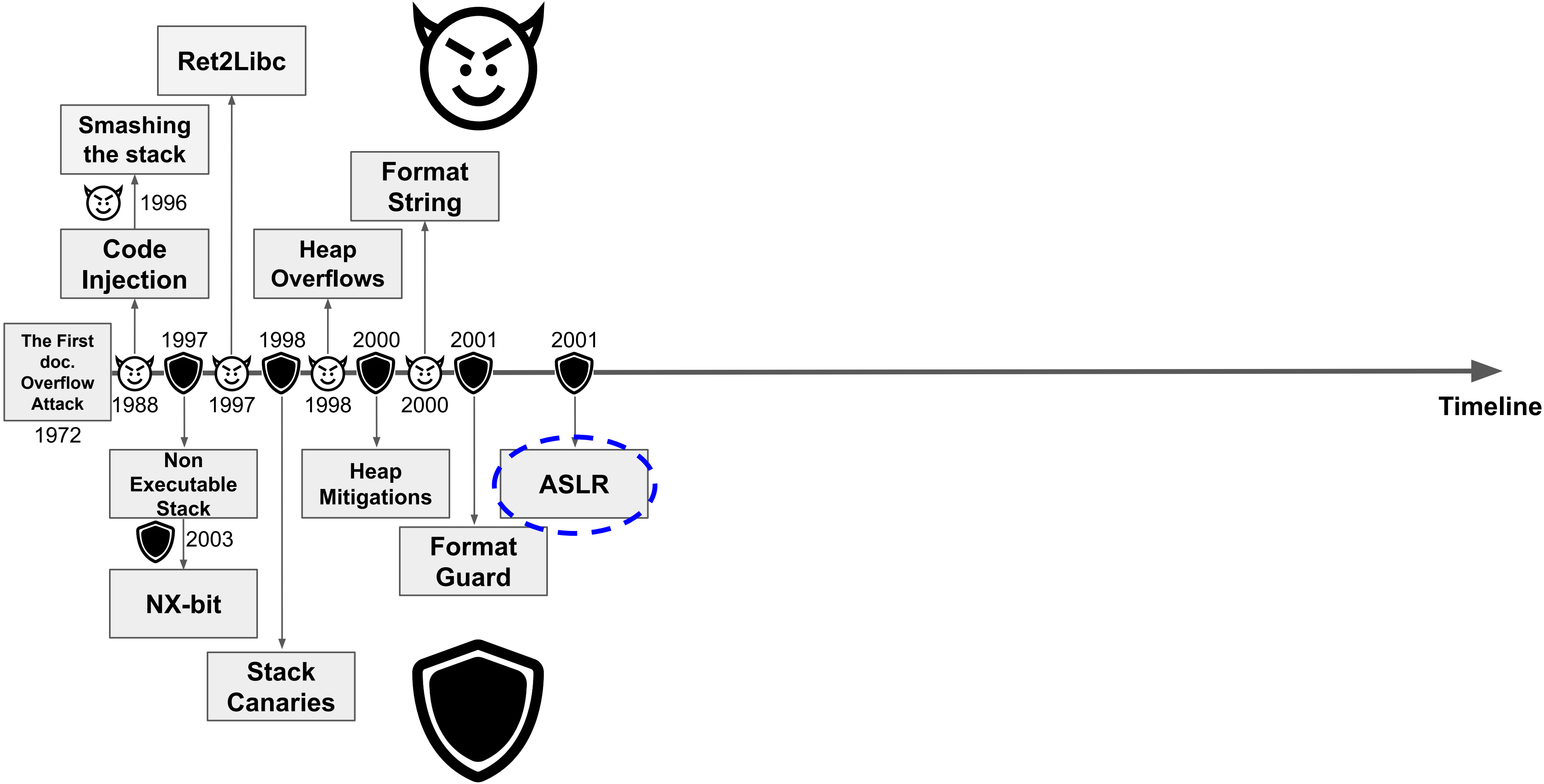

Ret2Libc
Smashing the stack
1996
Code Injection
Heap Overflows
The First doc. Overflow Attack
1972
1988
1997
1997
1998
1998
2000
Timeline
Non Executable Stack
2003
NX-bit
Stack Canaries
Heap Mitigations



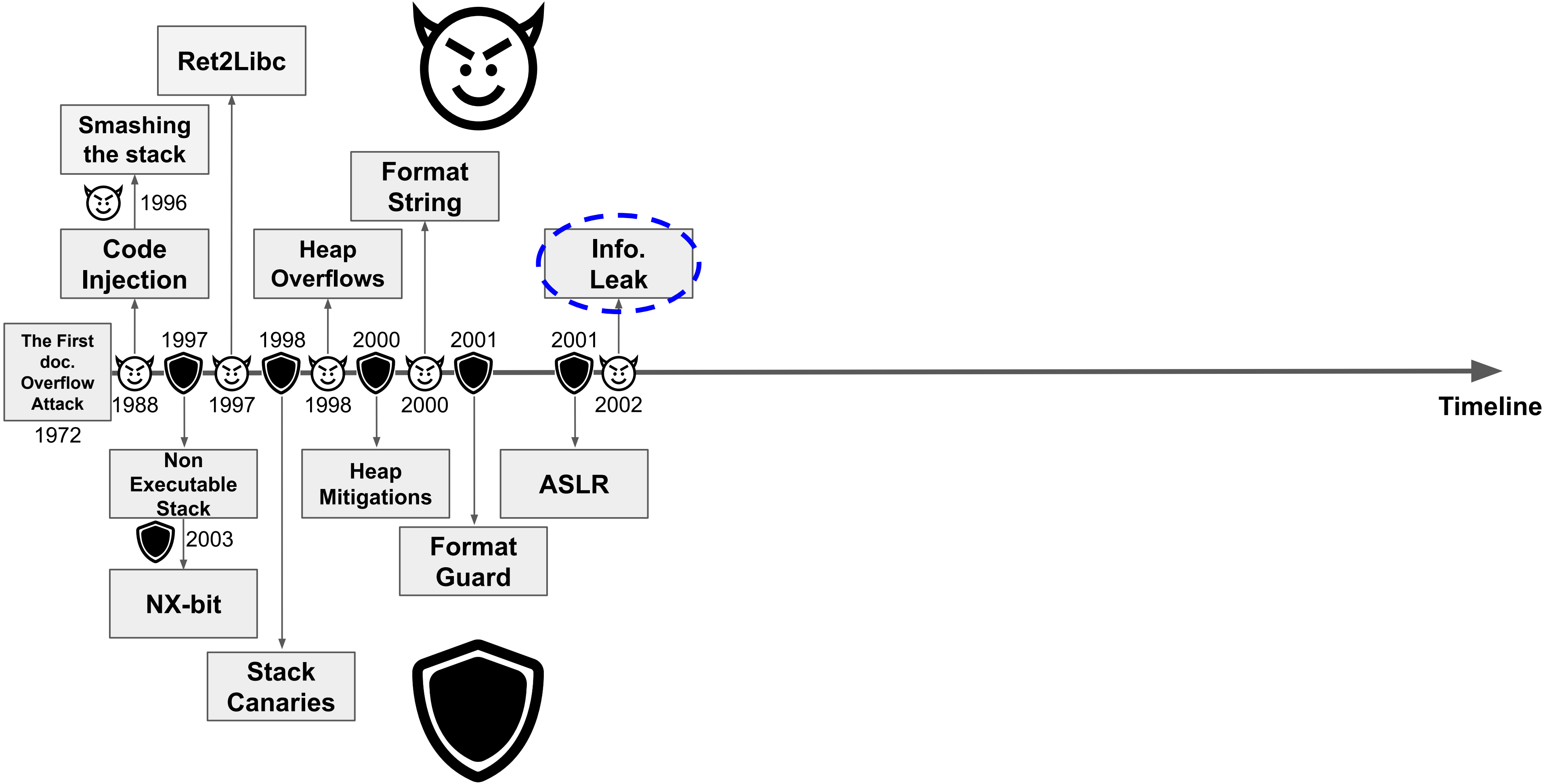
[5] Song et. al., SoK: Sanitizing for security. **[S&P 2019]**



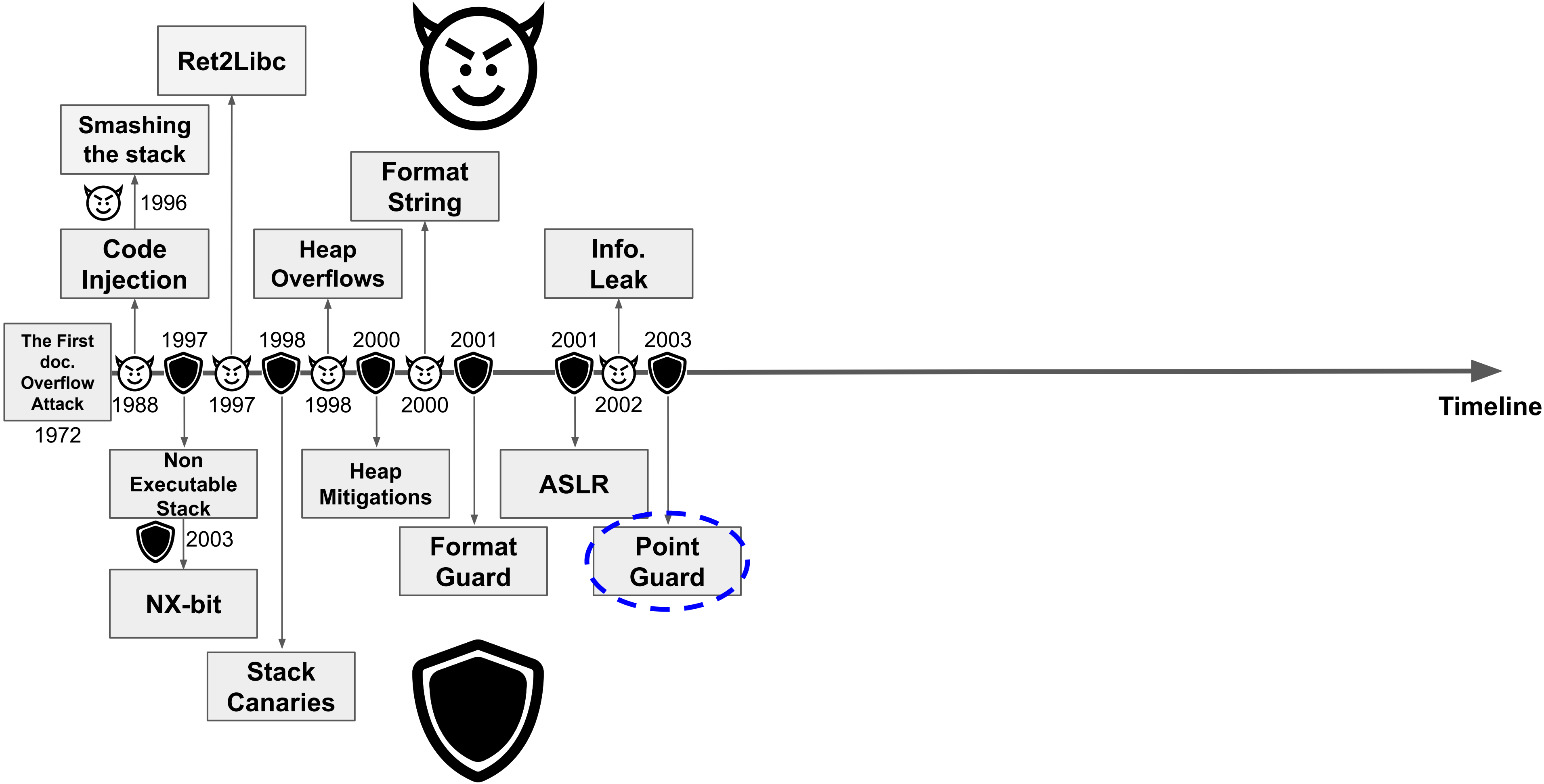
[5] Song et. al., SoK: Sanitizing for security. **[S&P 2019]**



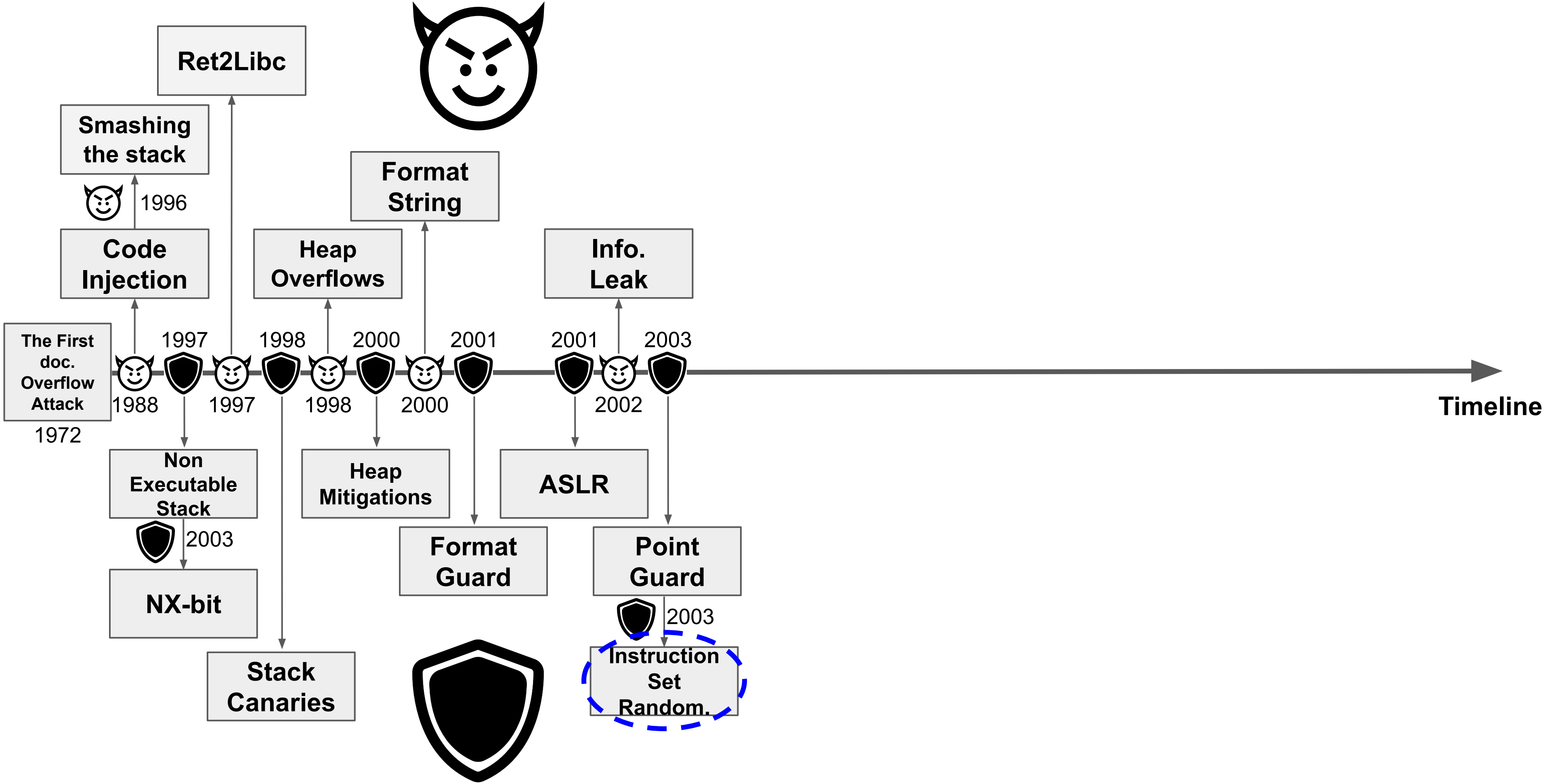
[11] PaX-Team, PaX address space layout randomization. **[2003]**



[4] Szekeres et. al, SoK: Eternal war in memory. **[S&P 2013]**



[4] Szekeres et. al, SoK: Eternal war in memory. **[S&P 2013]**



The First doc. Overflow Attack

1972

1988

Code Injection

1996

Smashing the stack

1997

Non Executable Stack

2003

NX-bit

1997

Ret2Libc

1998

Stack Canaries

1998

Heap Overflows

2000

Heap Mitigations

2000

Format String

2001

Format Guard

2001

ASLR

2002

Info. Leak

2003

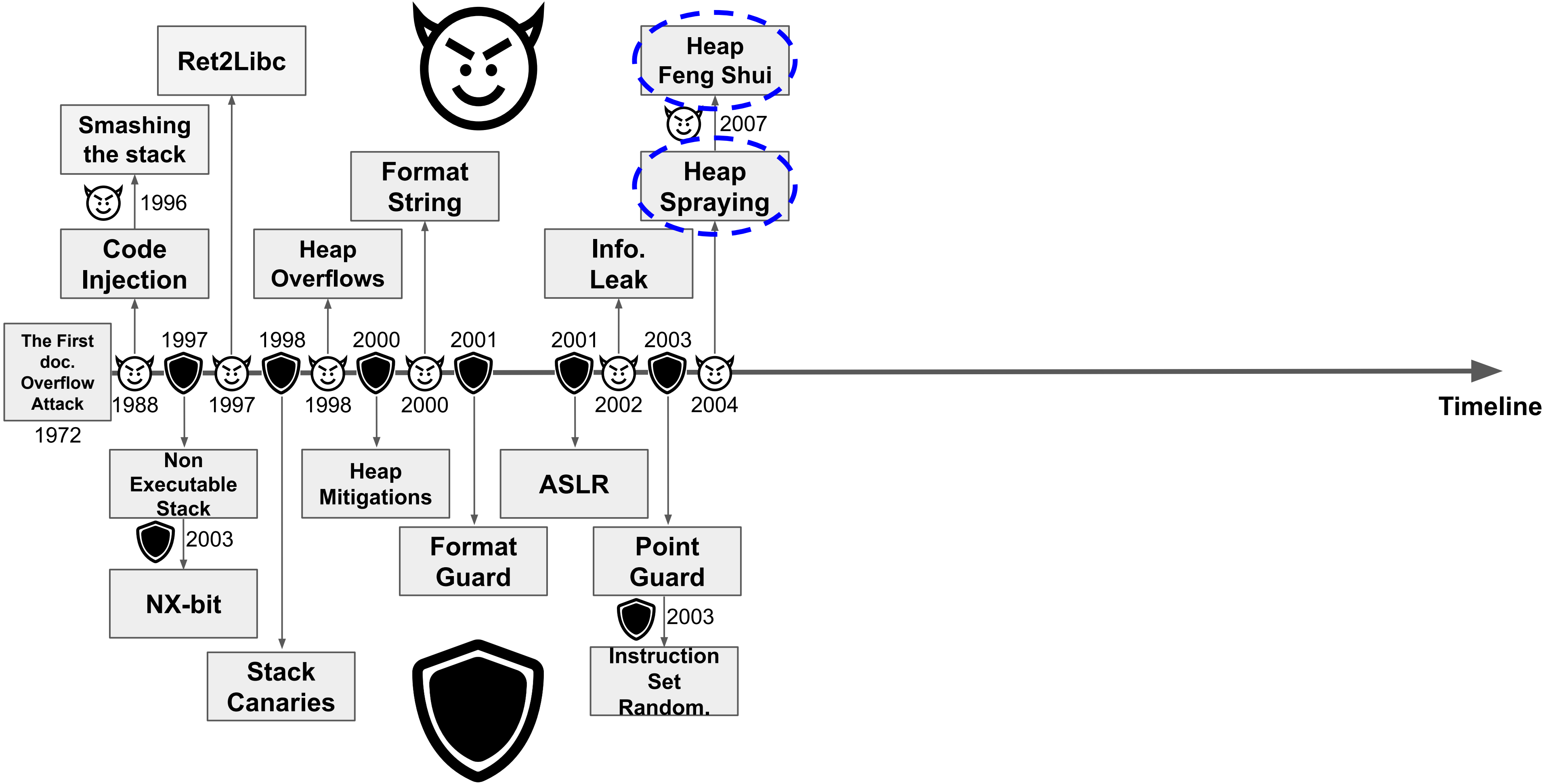
Point Guard

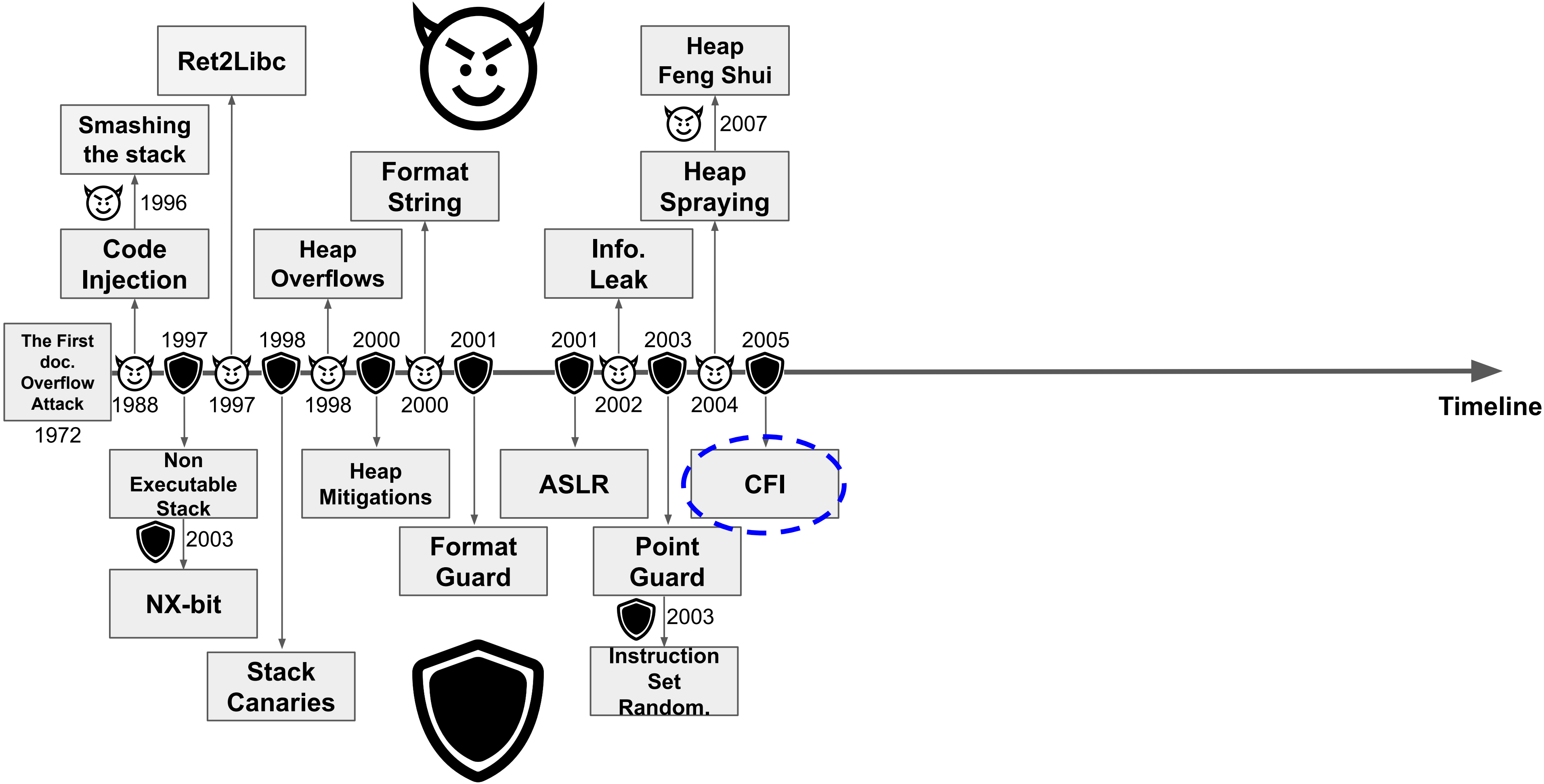
2003

Instruction Set Random.

Timeline

[4] Szekeres et. al, SoK: Eternal war in memory. **[S&P 2013]**


Ret2Libc
Heap
Feng Shui
2007
Smashing
the stack
1996
Format
String
Heap
Spraying
Code
Injection
Heap
Overflows
Info.
Leak
The First
doc.
Overflow
Attack
1972
1988
1997
1997
1998
1998
2000
2000
2001
2001
2002
2003
2004
Timeline
Non
Executable
Stack
2003
NX-bit
Heap
Mitigations
Format
Guard
ASLR
Point
Guard
2003
Instruction
Set
Random.
Stack
Canaries



The First doc. Overflow Attack

1972

1988

Code Injection

1996

Smashing the stack

1997

Non Executable Stack

2003

NX-bit

1997

Ret2Libc

1998

Stack Canaries

1998

Heap Overflows

2000

Heap Mitigations

2000

Format String

2001

Format Guard

2001

ASLR

2002

Info. Leak

2003

Point Guard

2003

Instruction Set Random.

2004

Heap Spraying

2007

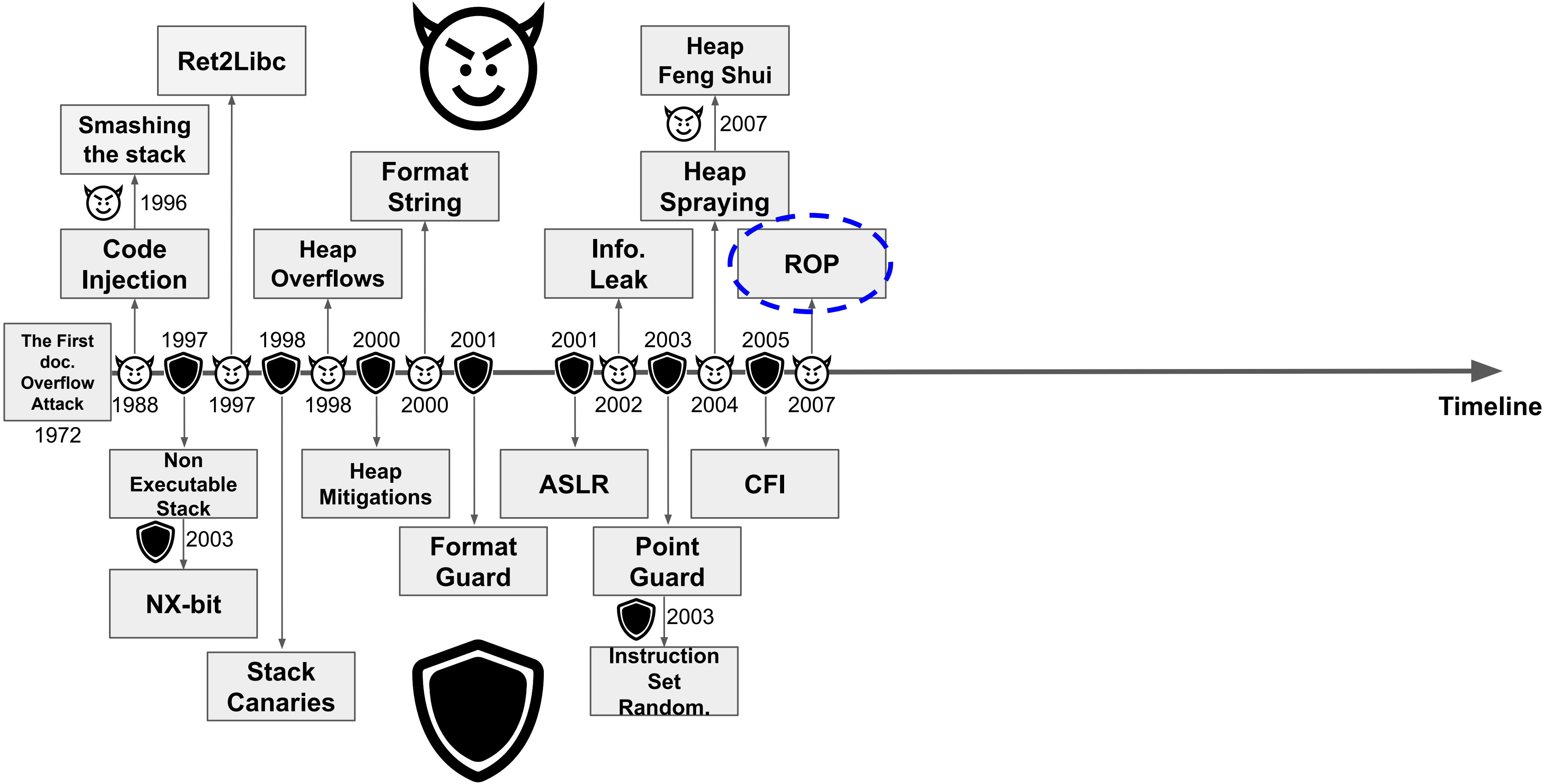
Heap Feng Shui

2005

CFI

Timeline

[12] Burow et. al, Control-flow integrity: Precision, security, and performance. **[ACM Surveys 2017]**



The First doc. Overflow Attack

1972

1988

Code Injection

1996

Smashing the stack

1997

Non Executable Stack

2003

NX-bit

1997

Ret2Libc

1998

Stack Canaries

1998

Heap Overflows

2000

Heap Mitigations

2000

Format String

2001

Format Guard

2001

ASLR

2002

Info. Leak

2003

Point Guard

2003

Instruction Set Random.

2004

Heap Spraying

2007

Heap Feng Shui

2005

CFI

2007

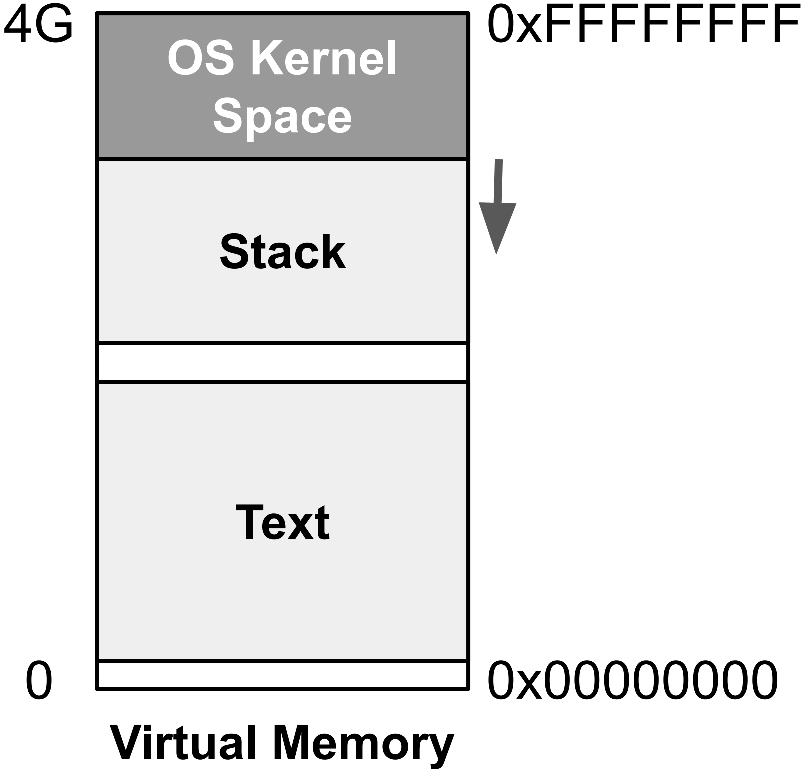
ROP

Timeline

[7] Hovav Shacham. The geometry of innocent flesh on the bone. **[CCS 2007]**

😈 RETURN ORIENTED PROGRAMMING (ROP)

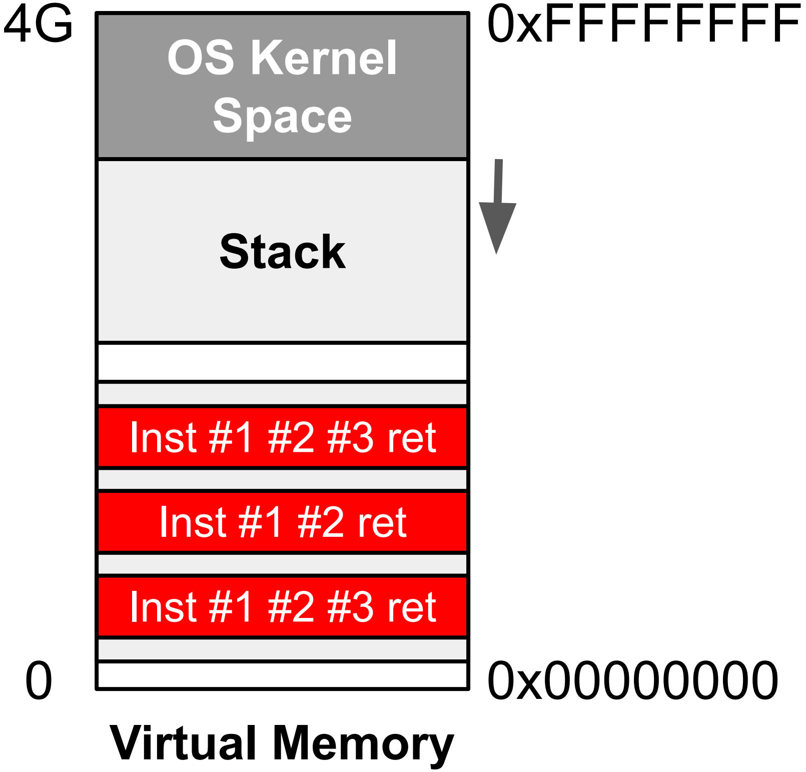
- Attack procedures:



[7] Hovav Shacham. The geometry of innocent flesh on the bone. **[CCS 2007]**

😈 RETURN ORIENTED PROGRAMMING (**ROP**)

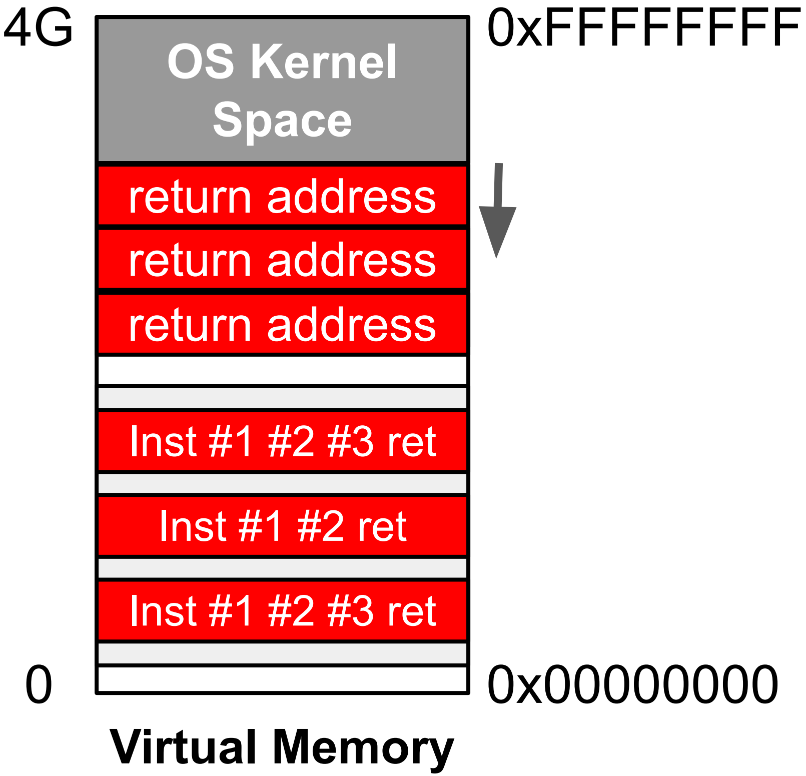
- Attack procedures:
    - Locate interesting gadgets.



[7] Hovav Shacham. The geometry of innocent flesh on the bone. **[CCS 2007]**

😈 RETURN ORIENTED PROGRAMMING (**ROP**)

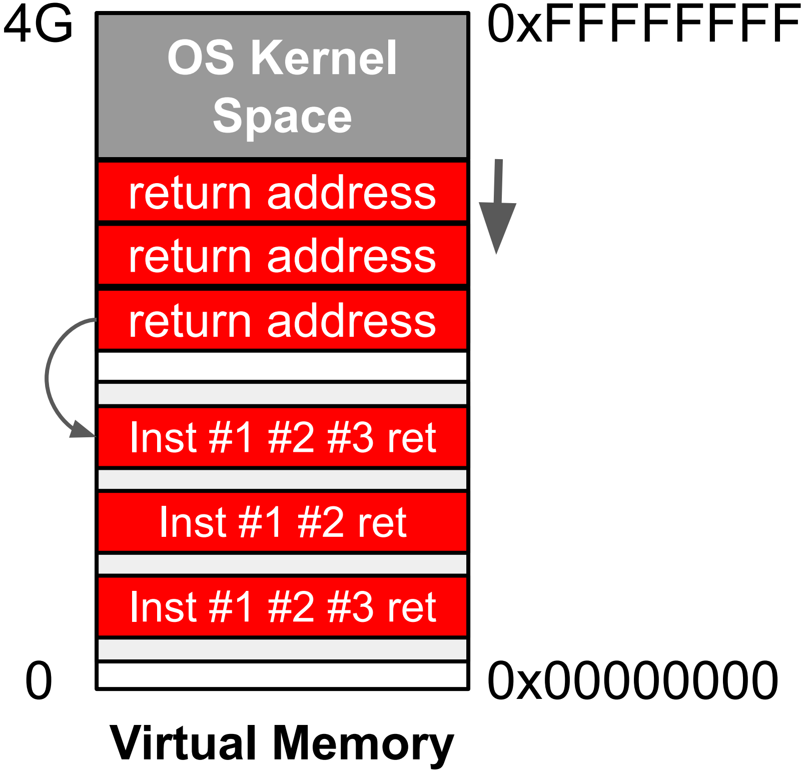
- Attack procedures:
    - Locate interesting gadgets.
    - Push sequence of gadget addresses to the stack.



[7] Hovav Shacham. The geometry of innocent flesh on the bone. **[CCS 2007]**

😈 RETURN ORIENTED PROGRAMMING (**ROP**)

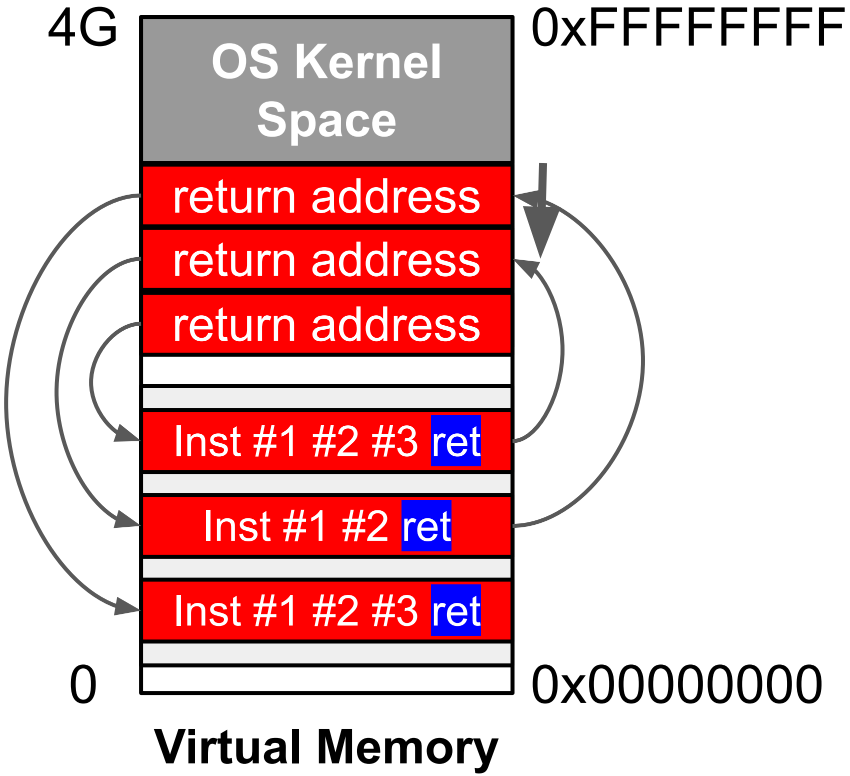
- Attack procedures:
  - Locate interesting gadgets.
  - Push sequence of gadget addresses to the stack.
  - Run!



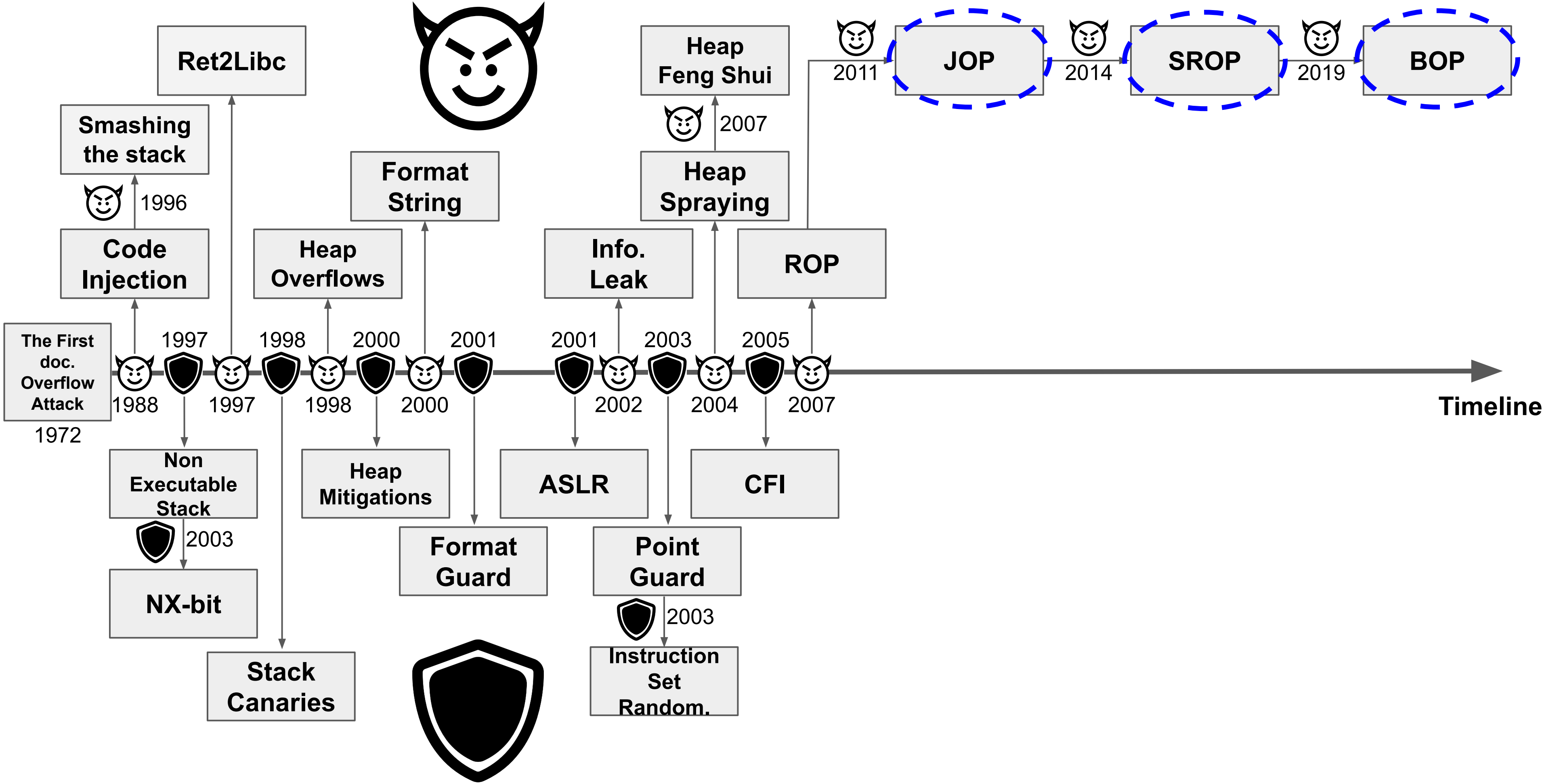
[7] Hovav Shacham. The geometry of innocent flesh on the bone. **[CCS 2007]**

😈 RETURN ORIENTED PROGRAMMING (**ROP**)

- Attack procedures:
  - Locate interesting gadgets.
  - Push sequence of gadget addresses to the stack.
  - Run!



[7] Hovav Shacham. The geometry of innocent flesh on the bone. **[CCS 2007]**



The First doc. Overflow Attack
1972

1988
Code Injection
1996
Smashing the stack

1997
Non Executable Stack
2003
NX-bit

1997
Ret2Libc

1998
Stack Canaries

1998
Heap Overflows

2000
Heap Mitigations

2000
Format String

2001
Format Guard

2001
ASLR

2002
Info. Leak

2003
Point Guard
2003
Instruction Set Random.

2004
Heap Spraying
2007
Heap Feng Shui

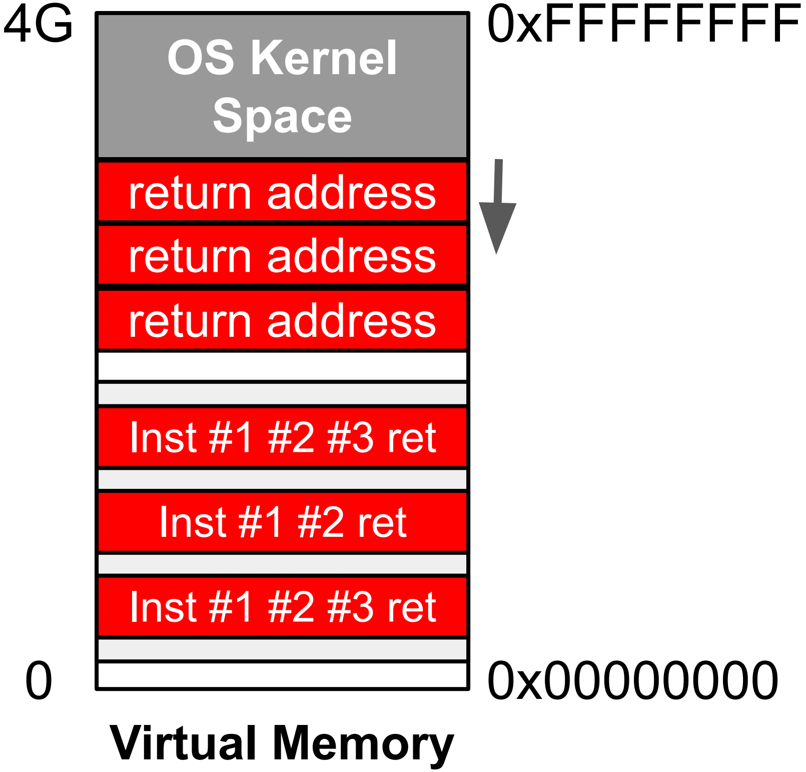
2005
CFI

2007
ROP
2011
JOP
2014
SROP
2019
BOP

Timeline

😈 RETURN ORIENTED PROGRAMMING (**ROP**)

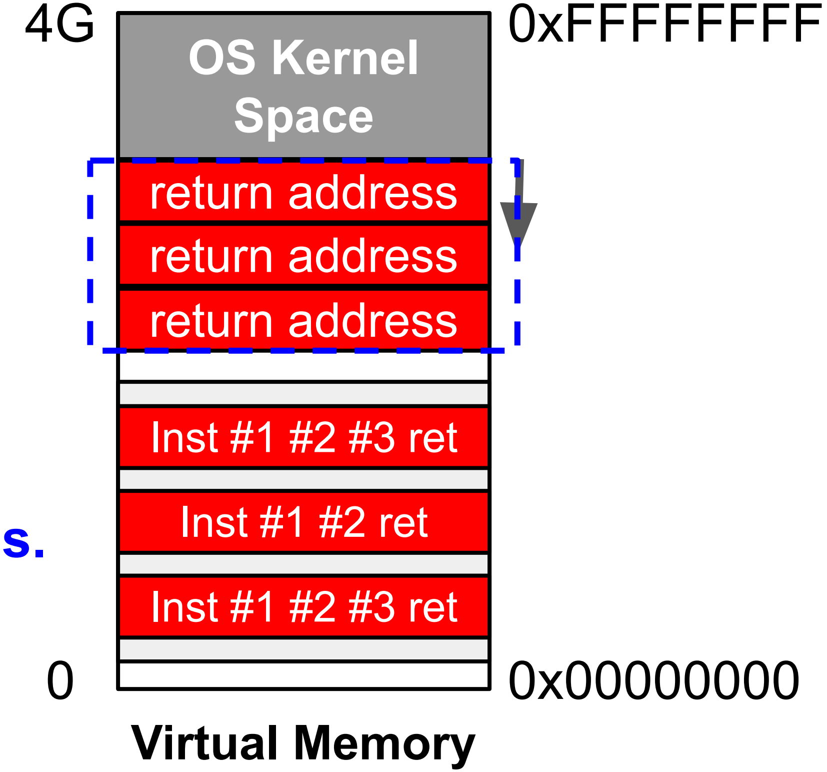
- Attack procedures:
  - Locate interesting gadgets.
  - Push sequence of gadget addresses to the stack.
  - Run!
- **Mitigations**:



[7] Hovav Shacham. The geometry of innocent flesh on the bone. **[CCS 2007]**

😈 RETURN ORIENTED PROGRAMMING (**ROP**)

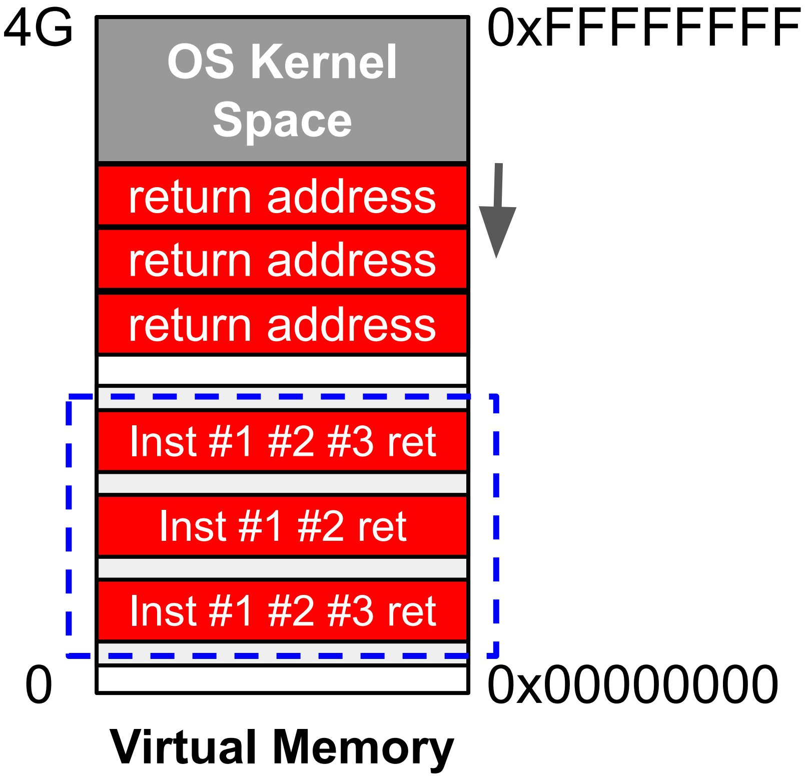
- Attack procedures:
  - Locate interesting gadgets.
  - Push sequence of gadget addresses to the stack.
  - Run!
- Mitigations:
  - **Protect the return addresses.**



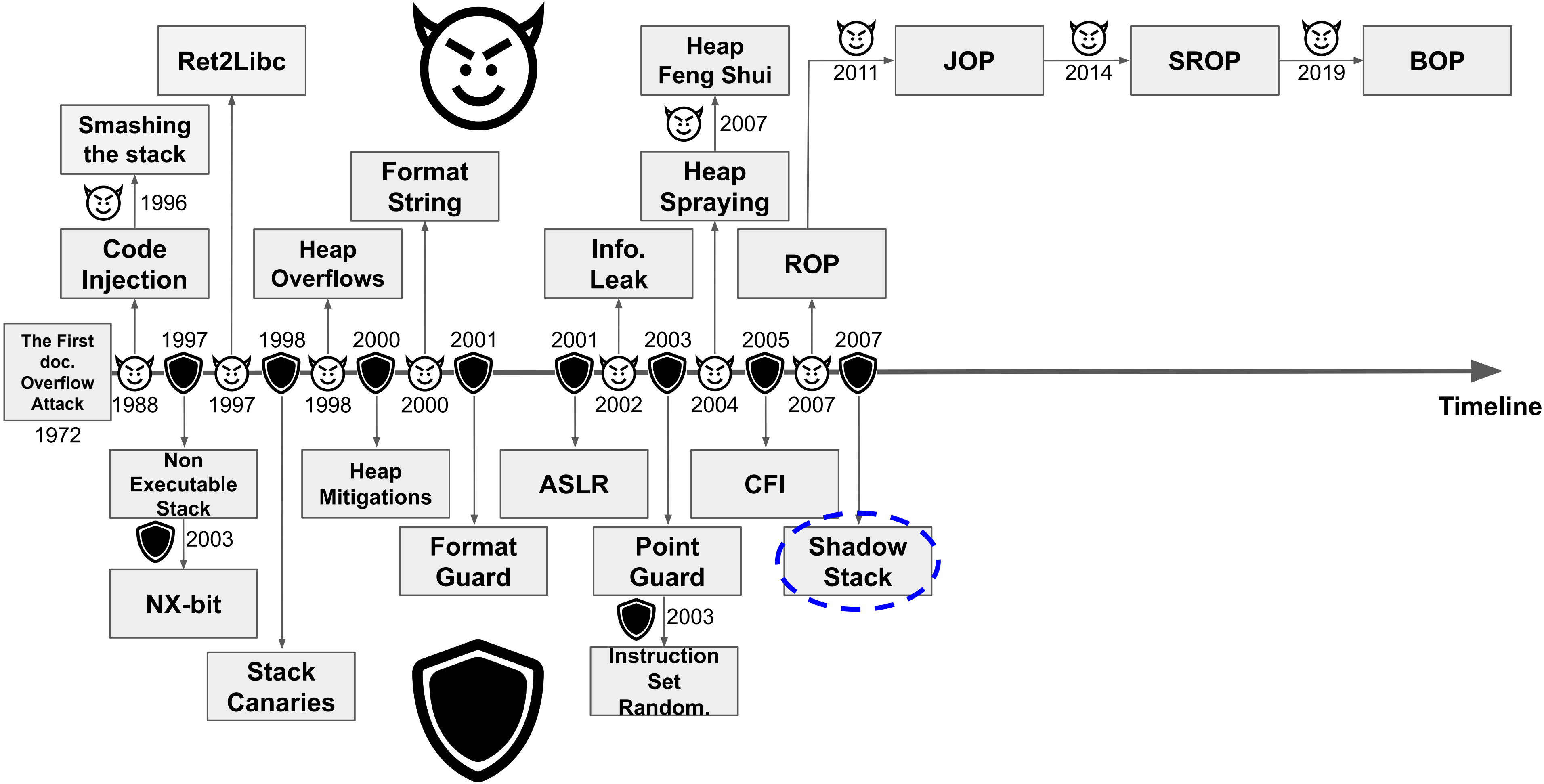
[7] Hovav Shacham. The geometry of innocent flesh on the bone. **[CCS 2007]**

😈 RETURN ORIENTED PROGRAMMING (**ROP**)

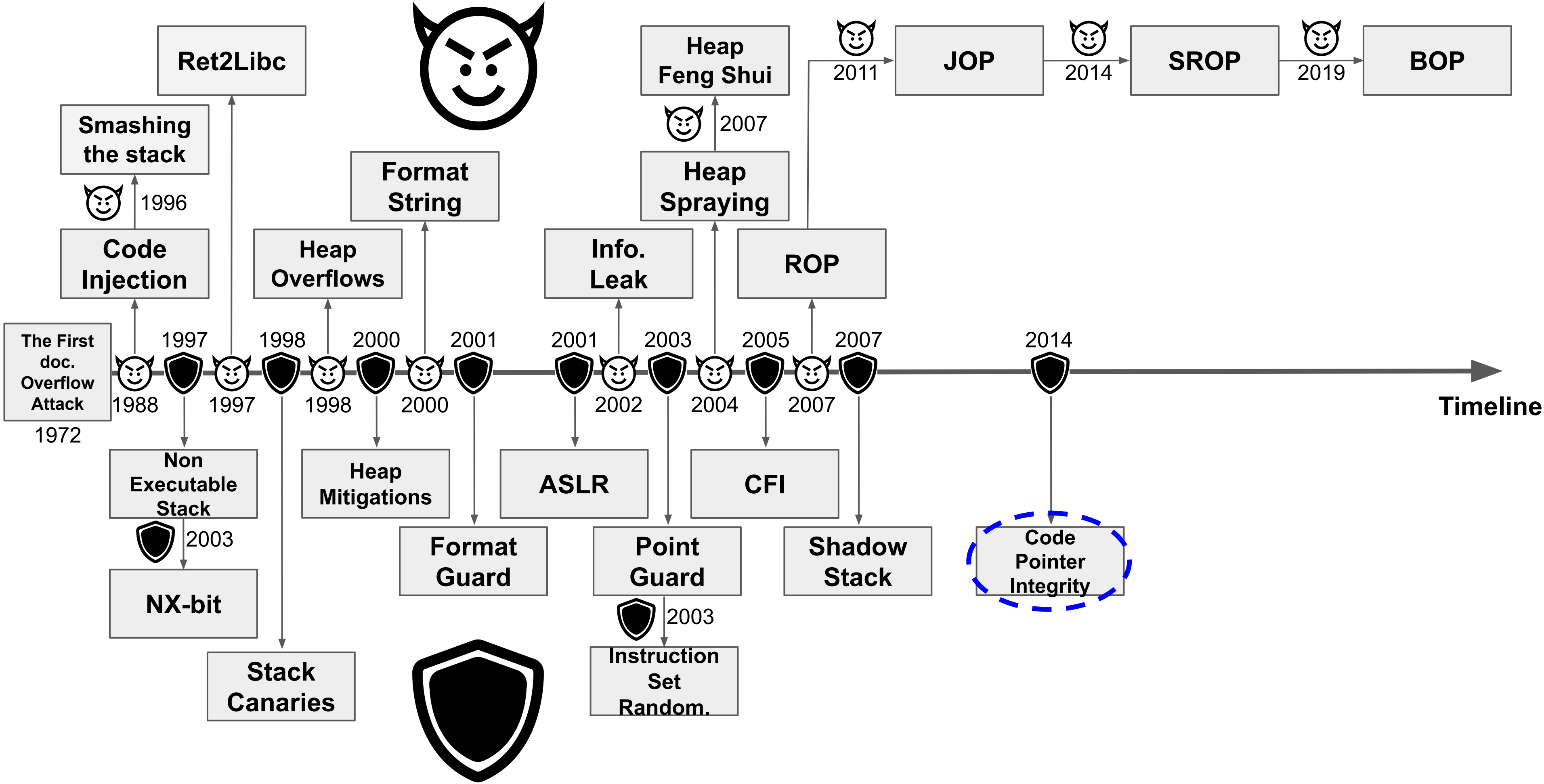
- Attack procedures:
  - ○ Locate interesting gadgets.
  - ○ Push sequence of gadget addresses to the stack.
  - ○ Run!
- Mitigations:
  - ○ Protect the return addresses.
  - ○ **Protect the gadgets.**

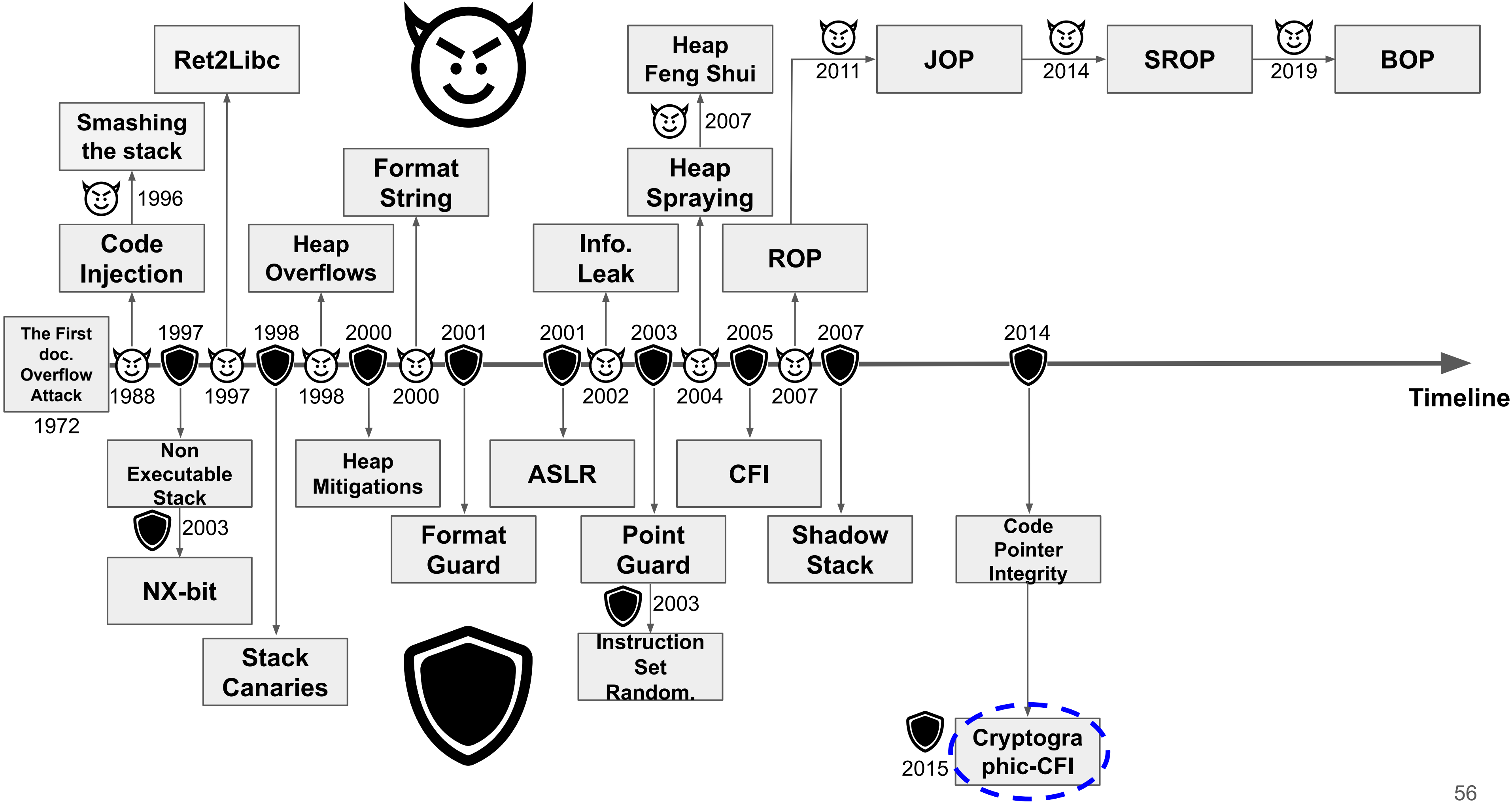


[7] Hovav Shacham. The geometry of innocent flesh on the bone. **[CCS 2007]**



[12] Burow et. al, Control-flow integrity: Precision, security, and performance. **[ACM Surveys 2017]**


The First doc. Overflow Attack
1972
1988
Code Injection
1996
Smashing the stack
1997
Non Executable Stack
2003
NX-bit
1997
Ret2Libc
1998
Stack Canaries
1998
Heap Overflows
2000
Heap Mitigations
2000
Format String
2001
Format Guard
2001
ASLR
2002
Info. Leak
2003
Point Guard
2003
Instruction Set Random.
2004
Heap Spraying
2007
Heap Feng Shui
2005
CFI
2007
ROP
2011
JOP
2014
SROP
2019
BOP
2007
Shadow Stack
2014
Code Pointer Integrity
Timeline



The First doc. Overflow Attack

1972

1988 Code Injection

1996 Smashing the stack

1997 Non Executable Stack

2003 NX-bit

1997 Ret2Libc

1998 Stack Canaries

1998 Heap Overflows

2000 Heap Mitigations

2000 Format String

2001 Format Guard

2001 ASLR

2002 Info. Leak

2003 Point Guard

2003 Instruction Set Random.

2004 Heap Spraying

2007 Heap Feng Shui

2005 CFI

2007 ROP

2011 JOP

2014 SROP

2019 BOP

2007 Shadow Stack

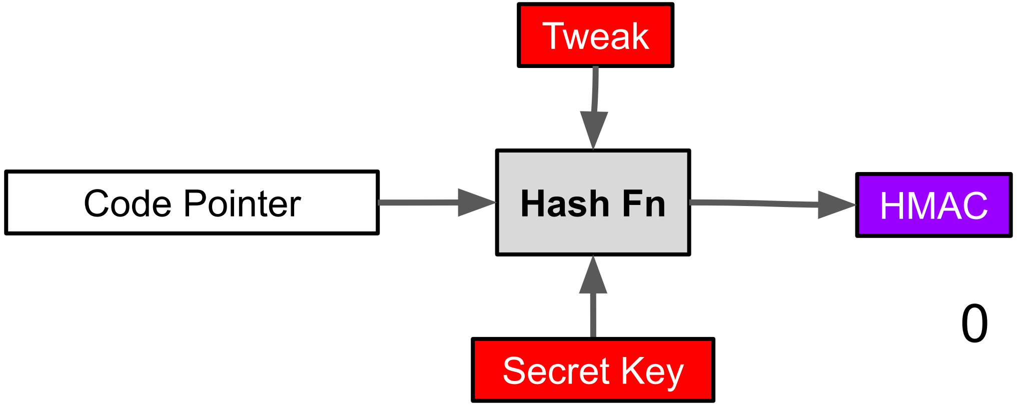
2014 Code Pointer Integrity

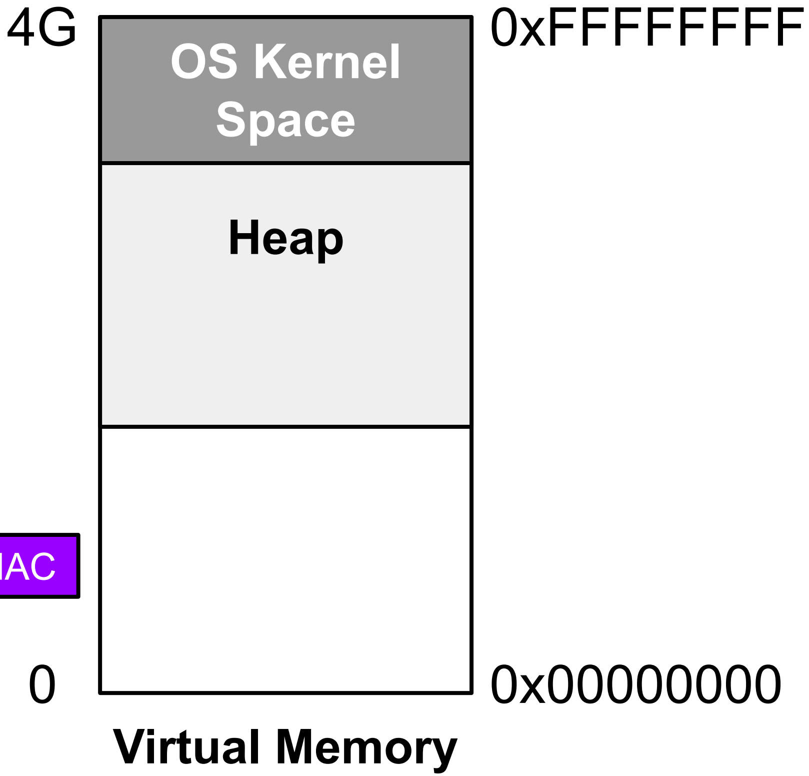
2015 Cryptographic-CFI

Timeline

CRYPTOGRAPHIC CONTROL FLOW INTEGRITY

- Create message authentication code (HMAC) for code pointers.
- Store HMAC in pointer itself.
- Verify HMAC upon pointer load.

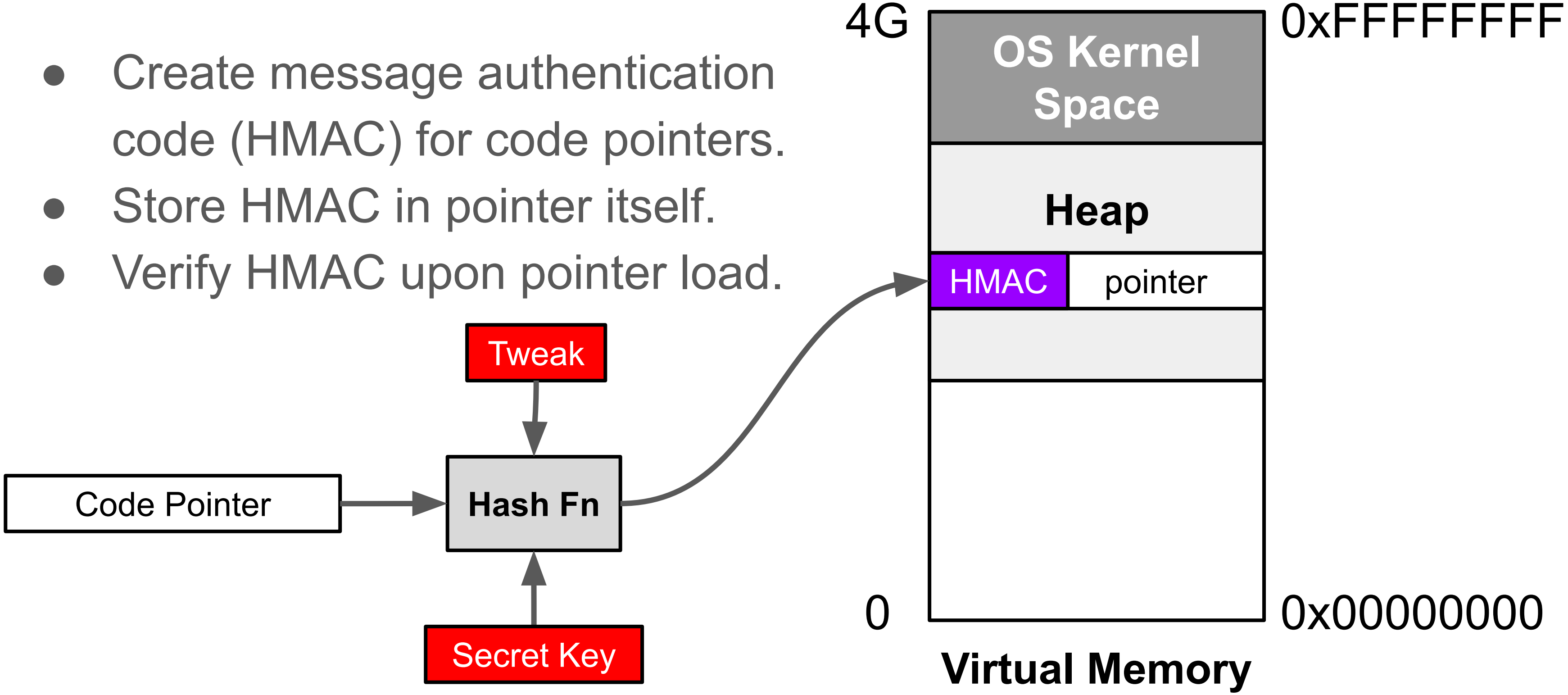




[16] Liljestrand et. al., PAC it up: Towards pointer integrity using ARM pointer authentication. **[USENIX 2019]**

CRYPTOGRAPHIC CONTROL FLOW INTEGRITY

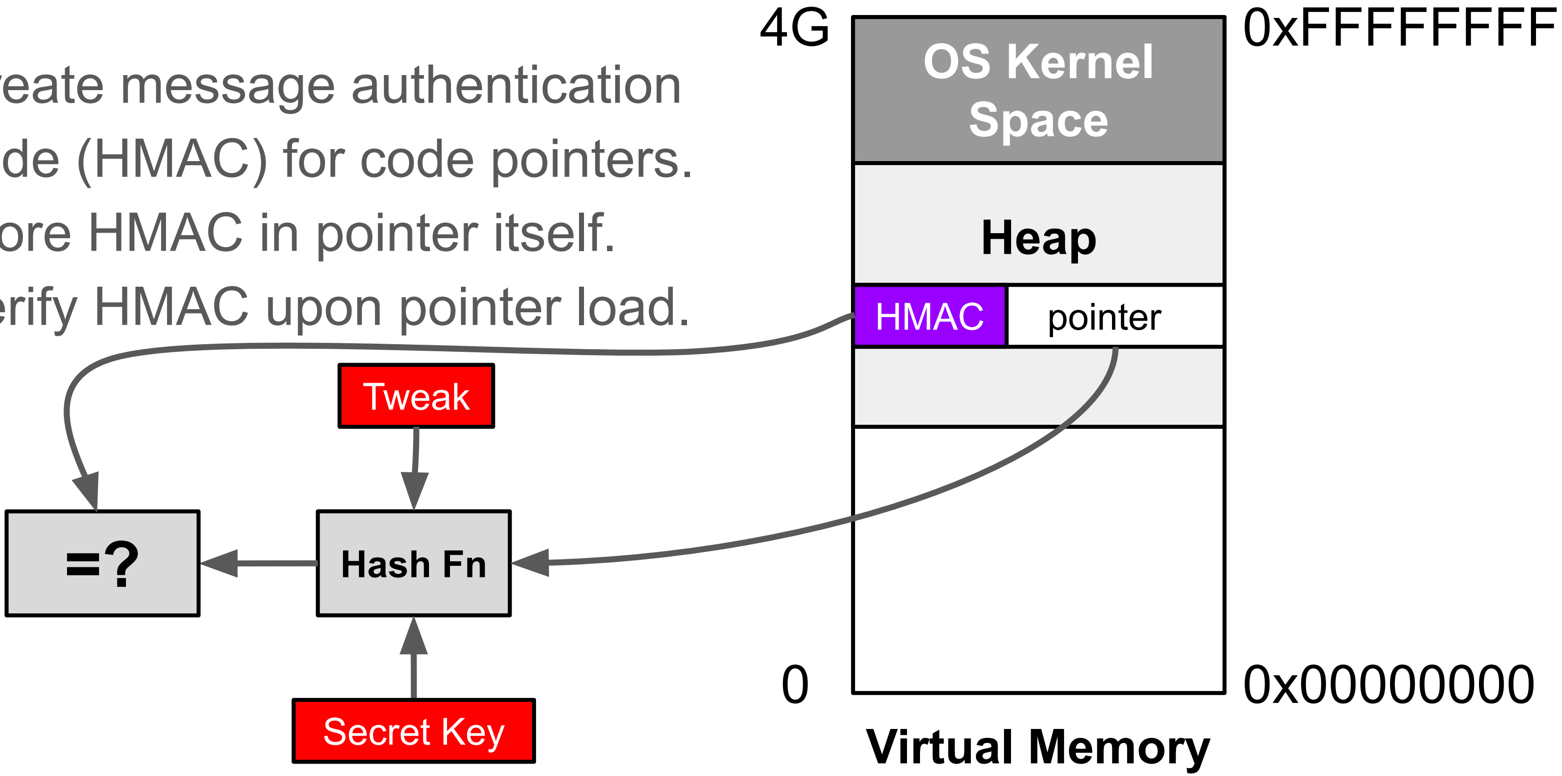
- Create message authentication code (HMAC) for code pointers.
- Store HMAC in pointer itself.
- Verify HMAC upon pointer load.



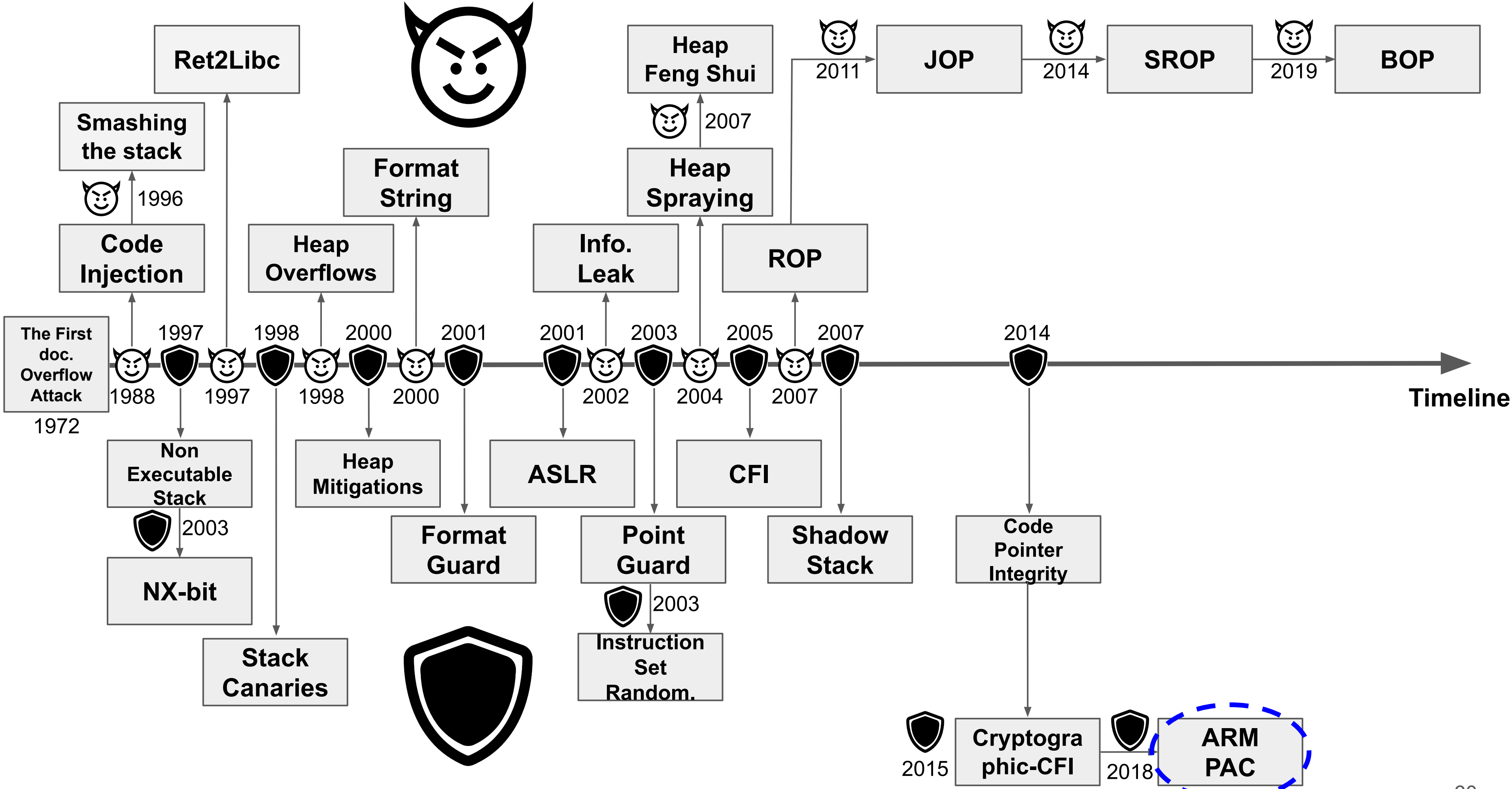
[16] Liljestrand et. al., PAC it up: Towards pointer integrity using ARM pointer authentication. **[USENIX 2019]**

CRYPTOGRAPHIC CONTROL FLOW INTEGRITY

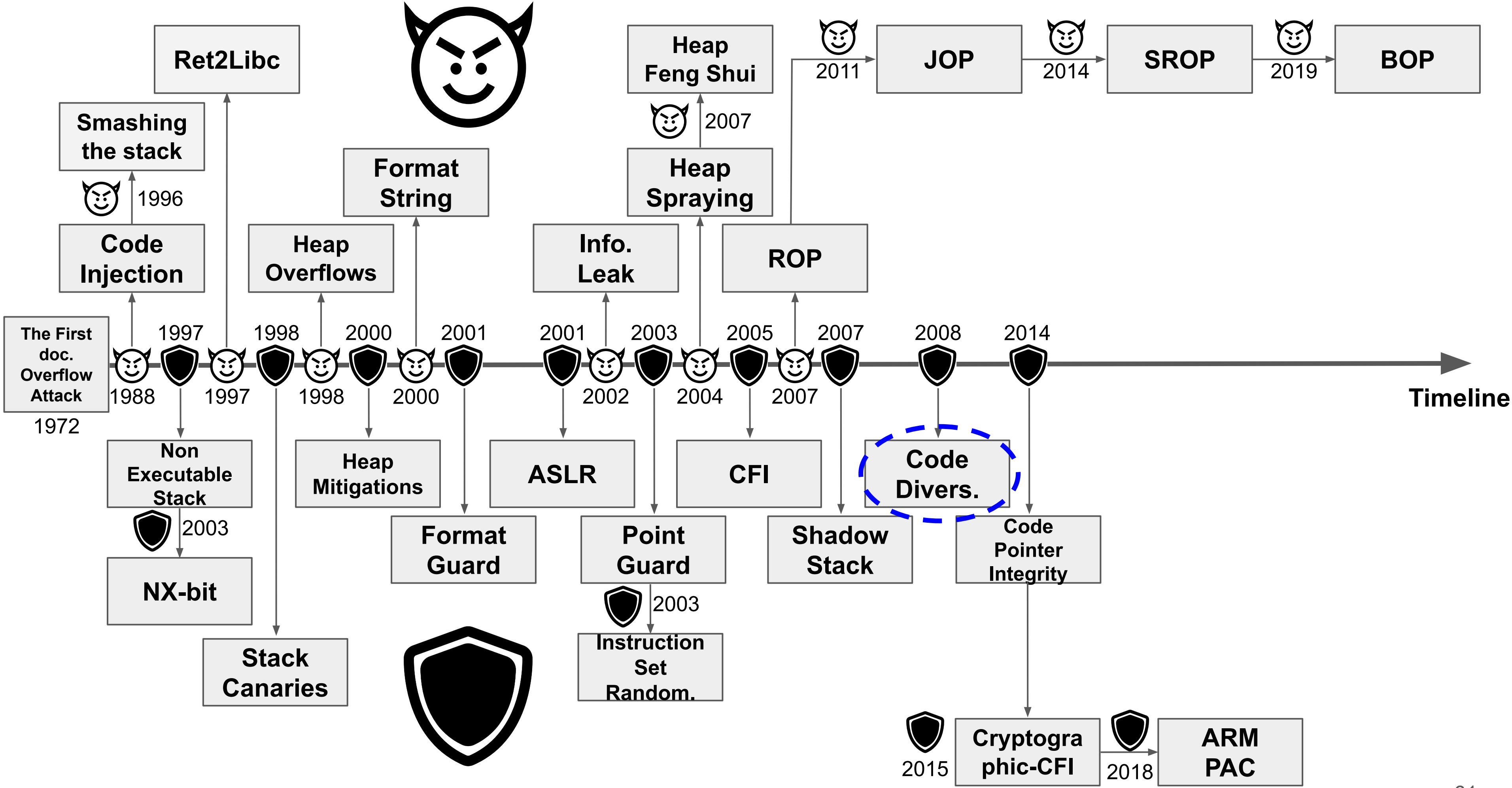
- Create message authentication code (HMAC) for code pointers.
- Store HMAC in pointer itself.
- Verify HMAC upon pointer load.



[16] Liljestrand et. al., PAC it up: Towards pointer integrity using ARM pointer authentication. **[USENIX 2019]**



[16] Liljestrand et. al., PAC it up: Towards pointer integrity using ARM pointer authentication. **[USENIX 2019]**



The First doc. Overflow Attack

1972

1988 Code Injection

1996 Smashing the stack

1997 Non Executable Stack

2003 NX-bit

1997 Ret2Libc

1998 Stack Canaries

1998 Heap Overflows

2000 Heap Mitigations

2000 Format String

2001 Format Guard

2001 ASLR

2002 Info. Leak

2003 Point Guard

2003 Instruction Set Random.

2004 Heap Spraying

2007 Heap Feng Shui

2005 CFI

2007 ROP

2011 JOP

2014 SROP

2019 BOP

2007 Shadow Stack

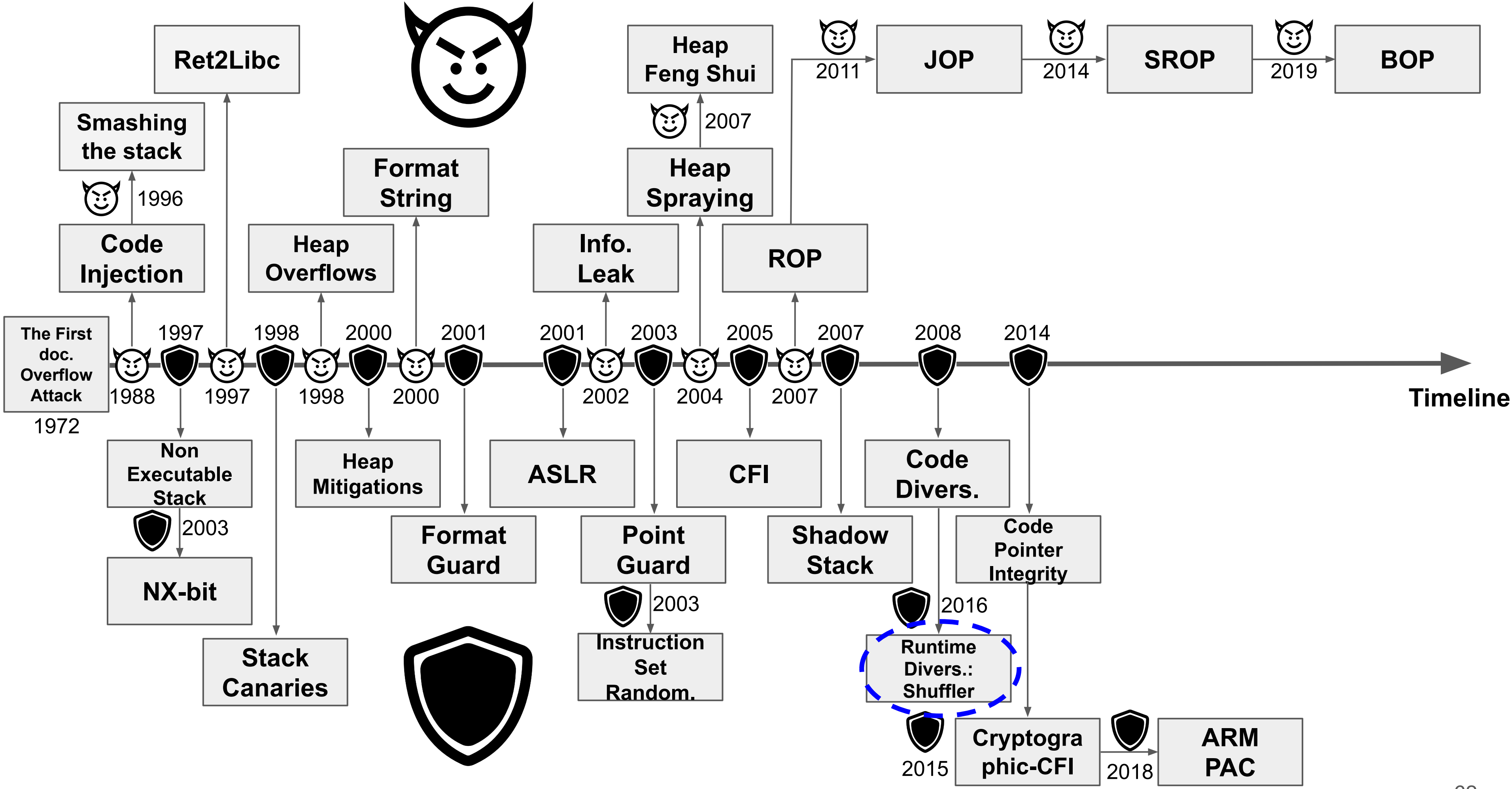
2008 Code Divers.

2014 Code Pointer Integrity

2015 Cryptographic-CFI

2018 ARM PAC

Timeline



The First doc. Overflow Attack

1972

1988 Code Injection

1996 Smashing the stack

1997 Non Executable Stack

2003 NX-bit

1997 Ret2Libc

1998 Stack Canaries

1998 Heap Overflows

2000 Heap Mitigations

2000 Format String

2001 Format Guard

2001 ASLR

2002 Info. Leak

2003 Point Guard

2003 Instruction Set Random.

2004 Heap Spraying

2007 Heap Feng Shui

2005 CFI

2007 ROP

2011 JOP

2014 SROP

2019 BOP

2007 Shadow Stack

2008 Code Divers.

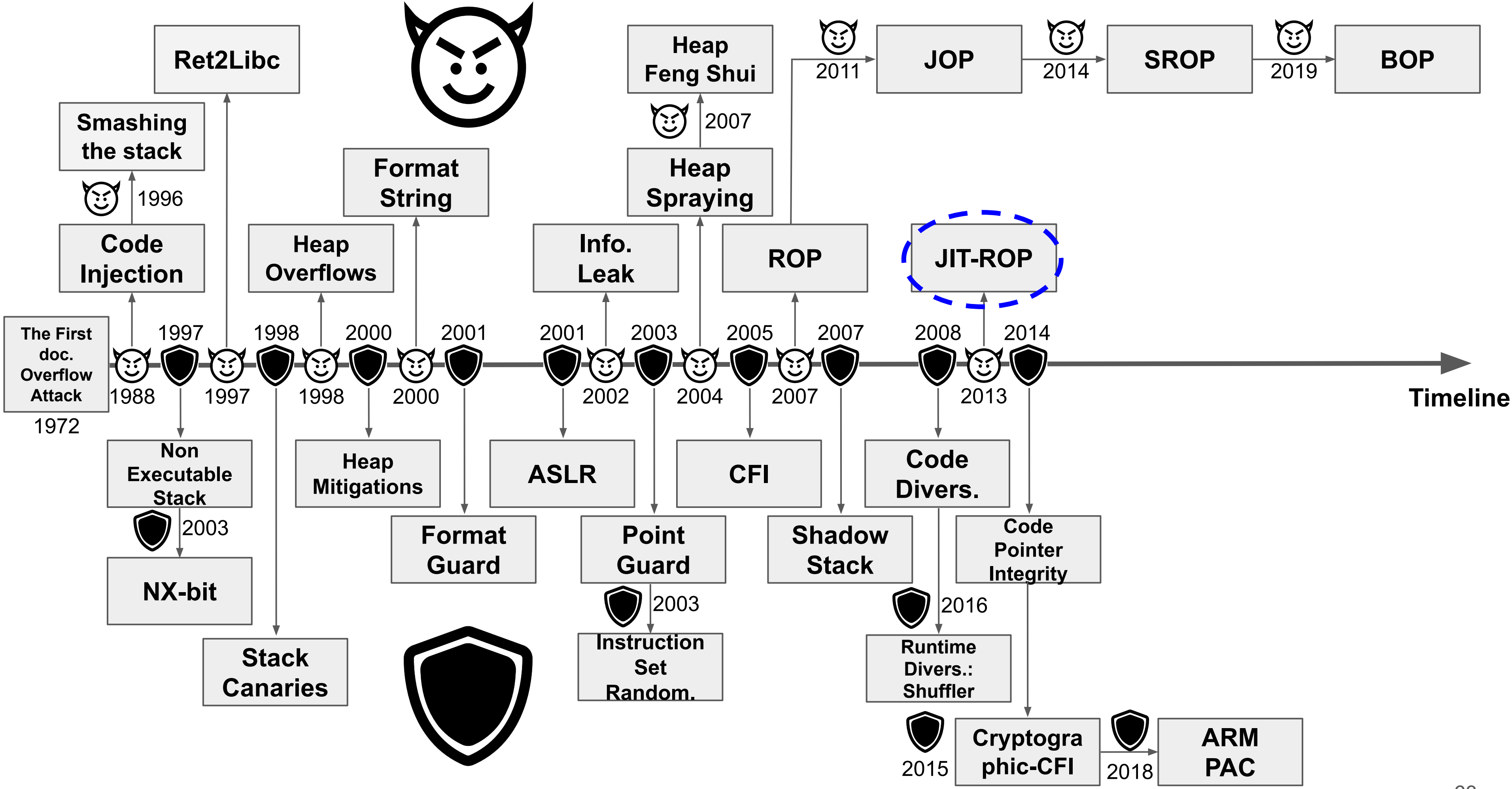
2016 Runtime Divers.: Shuffler

2014 Code Pointer Integrity

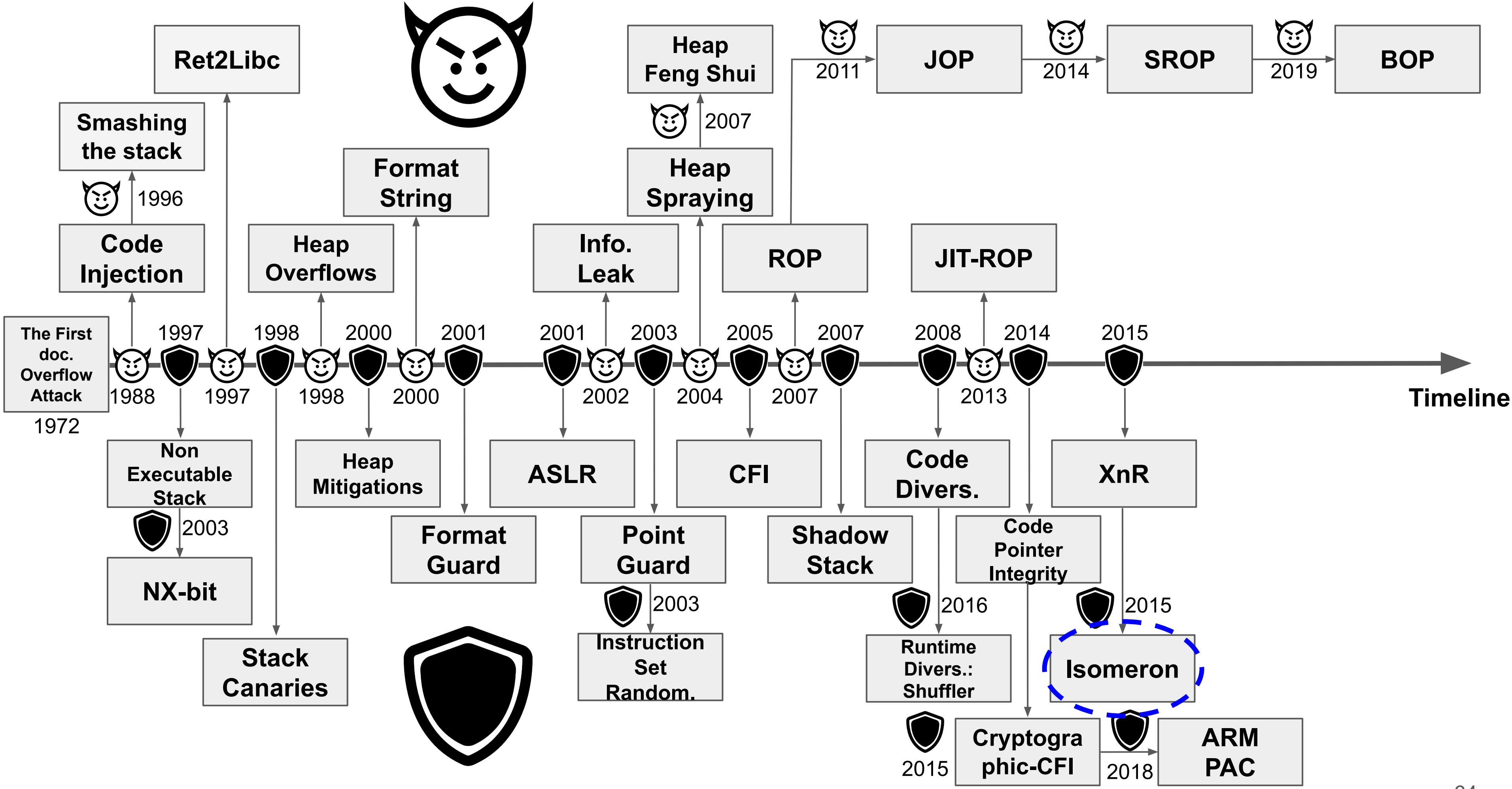
2015 Cryptographic-CFI

2018 ARM PAC

Timeline



[8] Snow et. al,Just-in-time code reuse: On the effectiveness of fine-grained ASLR. **[S&P 2013]**



The First doc. Overflow Attack

1972

1988

Code Injection

1996

Smashing the stack

1997

Non Executable Stack

2003

NX-bit

1997

Ret2Libc

1998

Stack Canaries

1998

Heap Overflows

2000

Heap Mitigations

2000

Format String

2001

Format Guard

2001

ASLR

2002

Info. Leak

2003

Point Guard

2003

Instruction Set Random.

2004

Heap Spraying

2007

Heap Feng Shui

2005

CFI

2007

ROP

2011

JOP

2014

SROP

2019

BOP

2007

Shadow Stack

2008

Code Divers.

2016

Runtime Divers.: Shuffler

2013

JIT-ROP

2014

Code Pointer Integrity

2015

Cryptographic-CFI

2015

XnR

2015

Isomeron

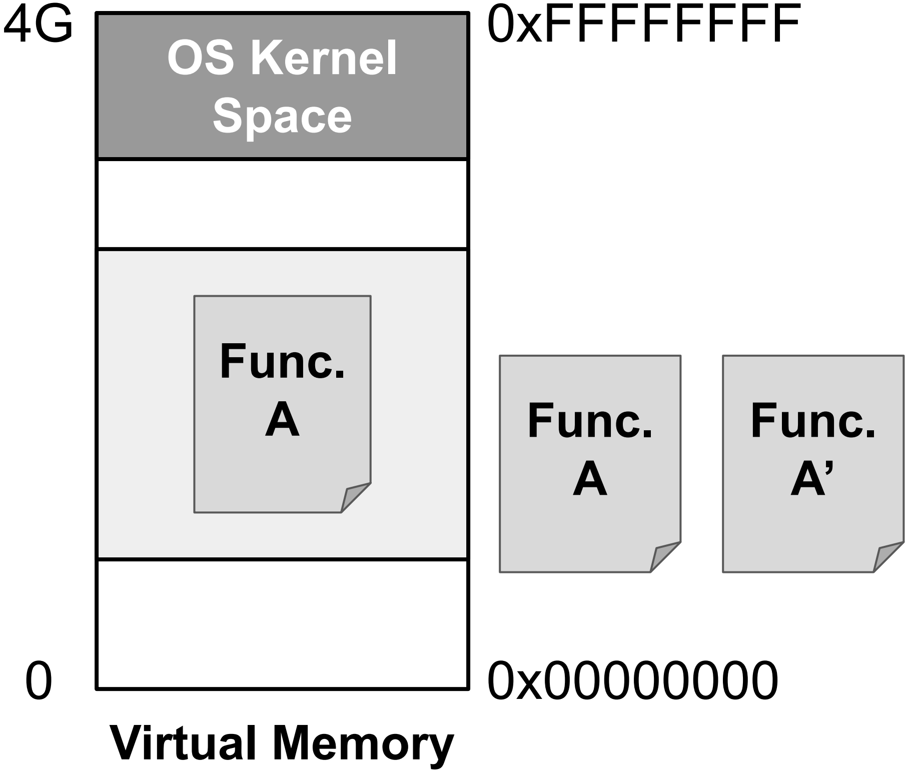
2018

ARM PAC

Timeline

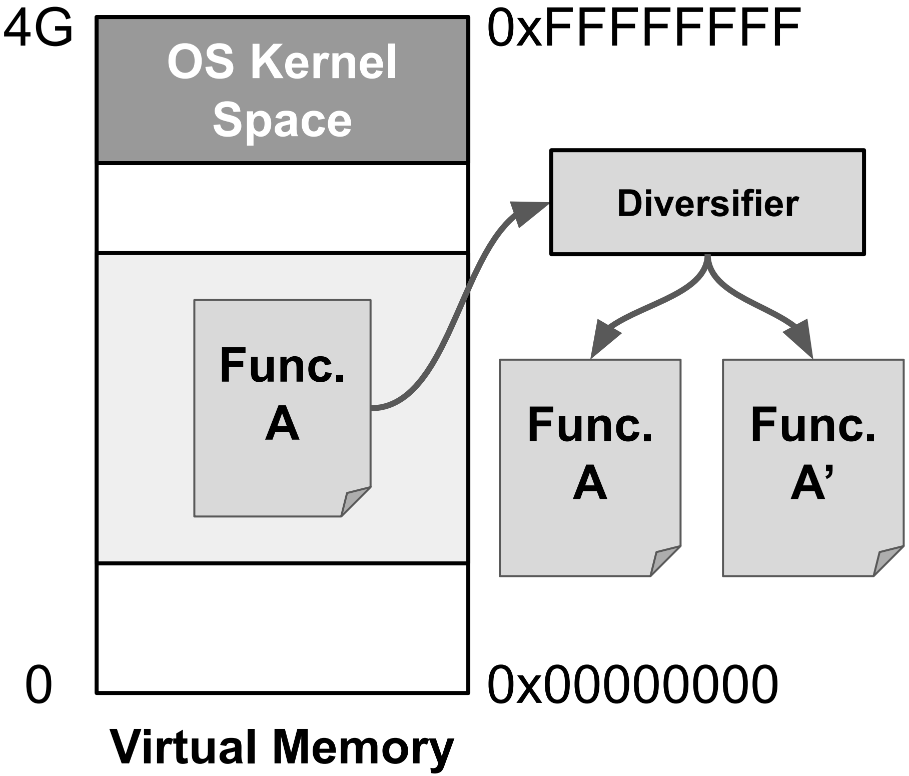
ISOMERON

- Main Idea:
    - Create two diversified variants of each function.



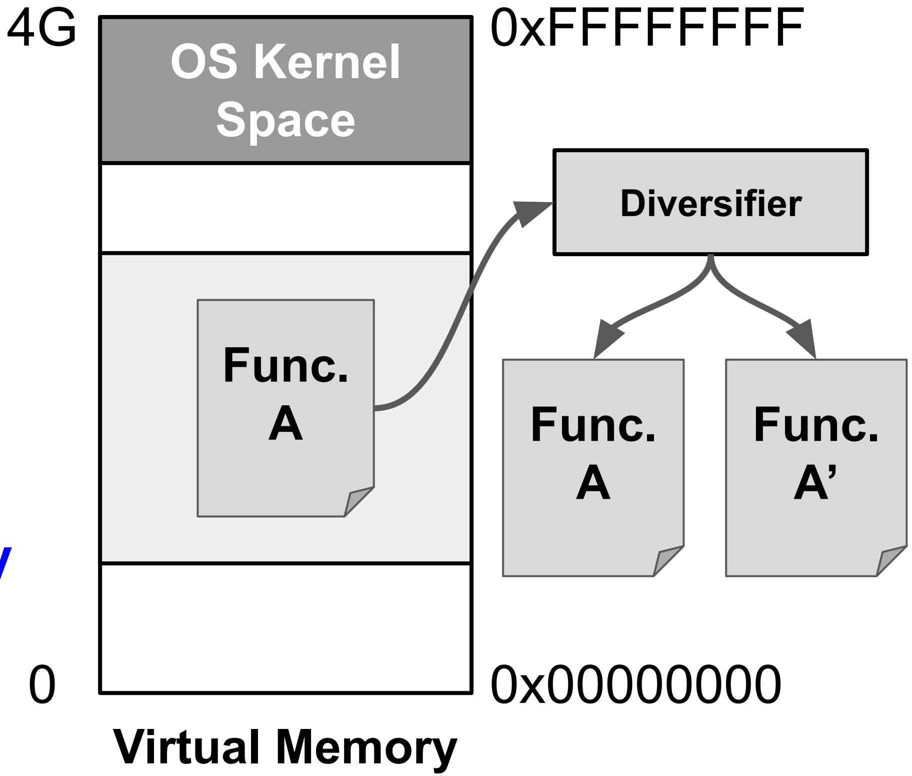
ISOMERON

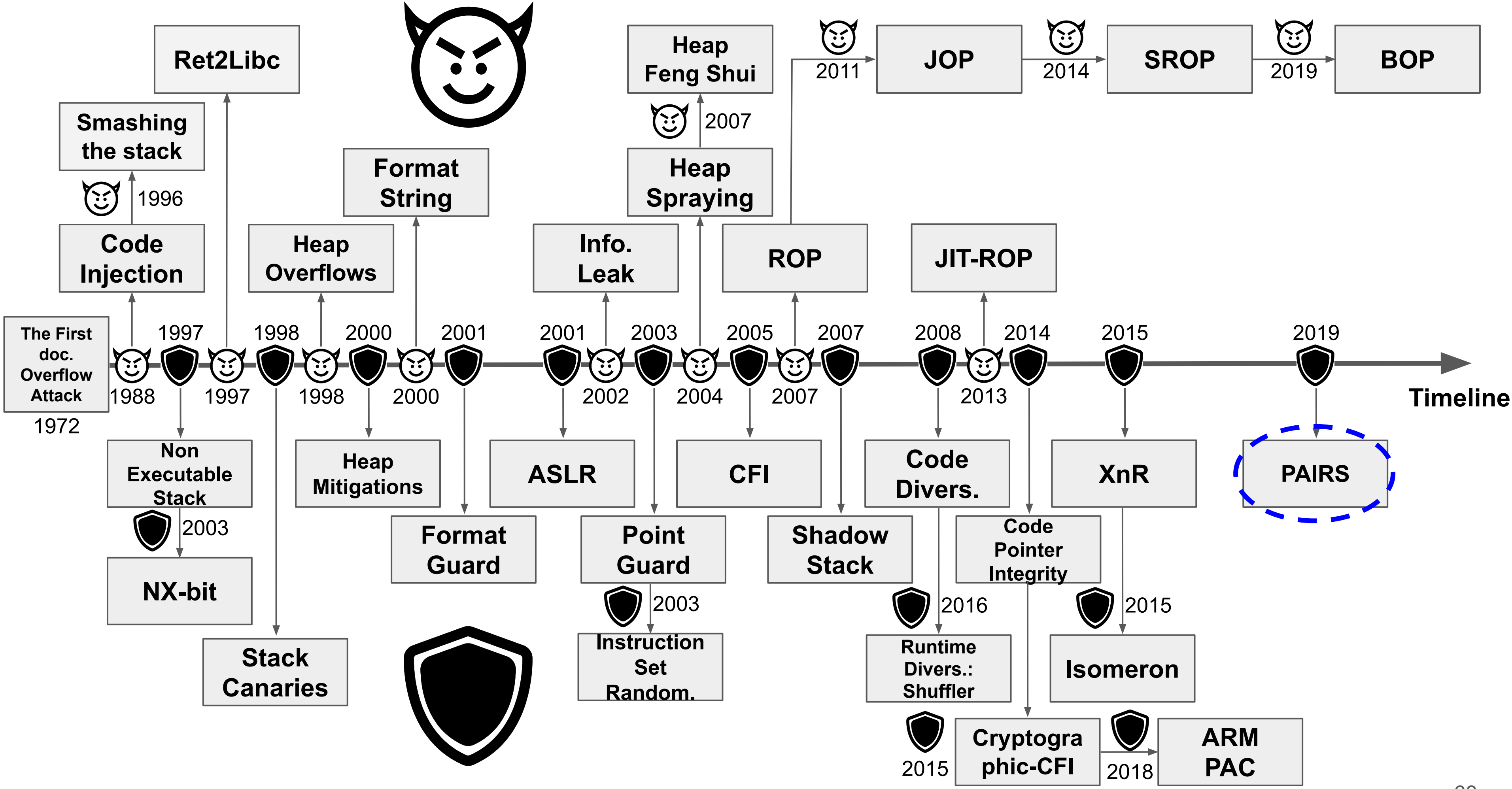
- Main Idea:
  - Create two diversified variants of each function.
  - Pick which variant to execute **randomly** at runtime.



ISOMERON

- Main Idea:
  - Create two diversified variants of each function.
  - Pick which variant to execute randomly at runtime.
- Goal:
  - Prevent JIT-ROP from **reliably** building a gadget chain.





The First doc. Overflow Attack

1972

1988 Code Injection

1996 Smashing the stack

1997 Ret2Libc

1998 Heap Overflows

2000 Format String

2002 Info. Leak

2004 Heap Spraying

2007 Heap Feng Shui

2007 ROP

2011 JOP

2014 SROP

2019 BOP

2013 JIT-ROP

1997 Non Executable Stack

2003 NX-bit

1998 Stack Canaries

2000 Heap Mitigations

2001 Format Guard

2001 ASLR

2003 Point Guard

2003 Instruction Set Random.

2005 CFI

2007 Shadow Stack

2008 Code Divers.

2016 Runtime Divers.: Shuffler

2014 Code Pointer Integrity

2015 Cryptographic-CFI

2018 ARM PAC

2015 XnR

2015 Isomeron

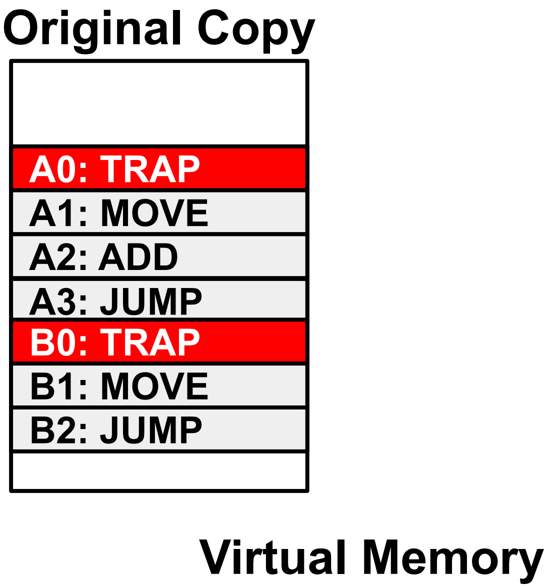
2019 PAIRS

Timeline

[2] Mohamed et. al, PAIRS: Control flow protection using phantom addressed instructions. **[arXiv 2019]**

PAIRS

- Main Idea:
  - ○ Insert TRAP instructions in the beginning of code basic blocks.



Original Copy

A0: TRAP

A1: MOVE

A2: ADD

A3: JUMP

B0: TRAP

B1: MOVE

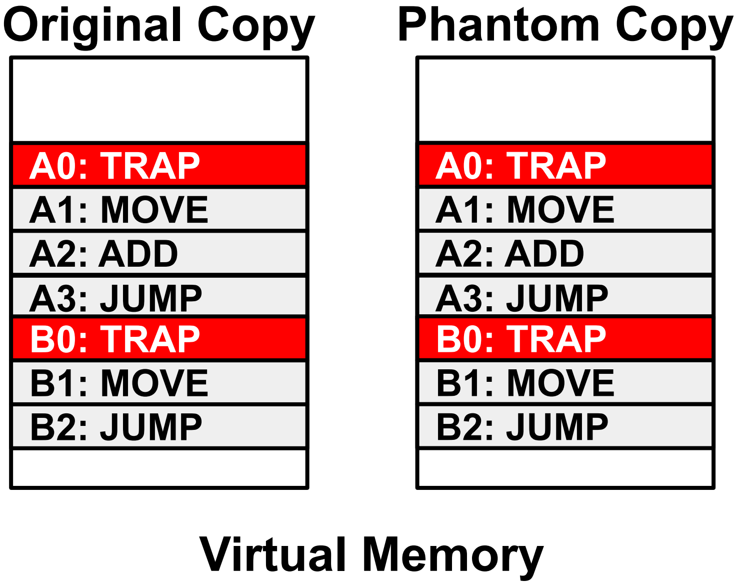
B2: JUMP

Virtual Memory

[2] Mohamed et. al, PAIRS: Control flow protection using phantom addressed instructions. **[arXiv 2019]**

PAIRS

- Main Idea:
  - Insert TRAP instructions in the beginning of code basic blocks.
  - Create two (or more) program copies in virtual memory.



Original Copy

A0: TRAP
A1: MOVE
A2: ADD
A3: JUMP
B0: TRAP
B1: MOVE
B2: JUMP

Phantom Copy

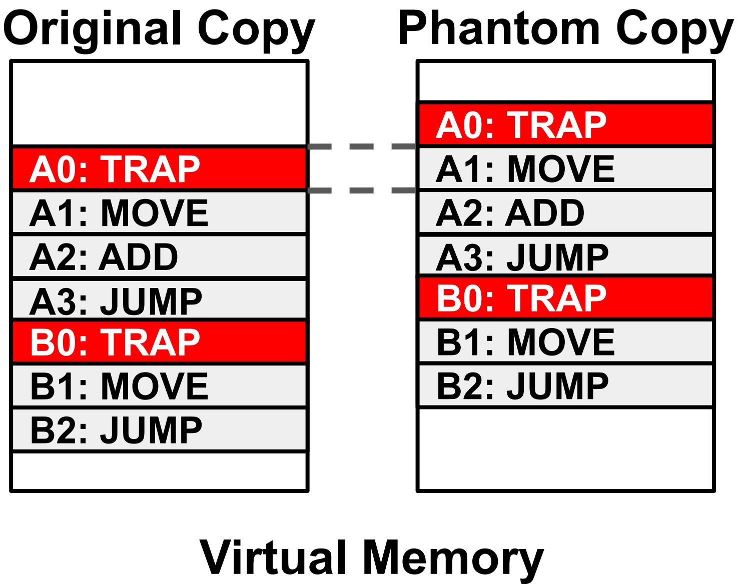
A0: TRAP
A1: MOVE
A2: ADD
A3: JUMP
B0: TRAP
B1: MOVE
B2: JUMP

Virtual Memory

[2] Mohamed et. al, PAIRS: Control flow protection using phantom addressed instructions. **[arXiv 2019]**

PAIRS

- Main Idea:
  - Insert TRAP instructions in the beginning of code basic blocks.
  - Create two (or more) program copies in virtual memory.



Original Copy

A0: TRAP
A1: MOVE
A2: ADD
A3: JUMP
B0: TRAP
B1: MOVE
B2: JUMP

Phantom Copy

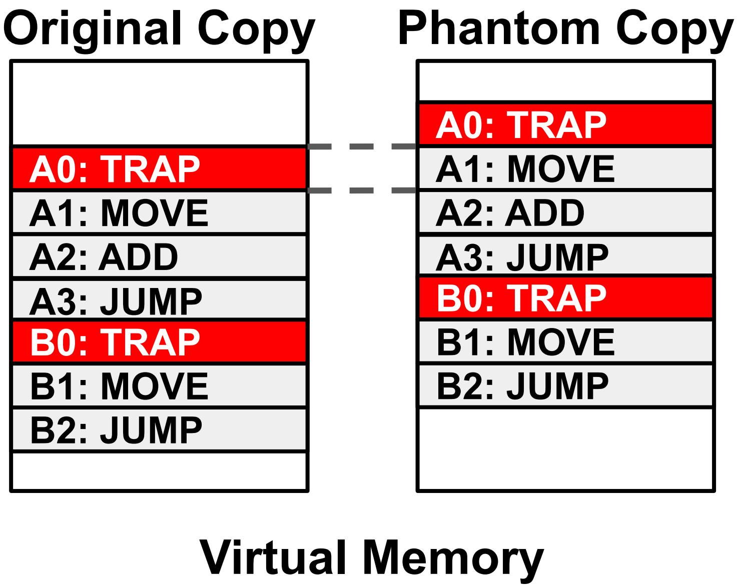
A0: TRAP
A1: MOVE
A2: ADD
A3: JUMP
B0: TRAP
B1: MOVE
B2: JUMP

Virtual Memory

[2] Mohamed et. al, PAIRS: Control flow protection using phantom addressed instructions. **[arXiv 2019]**

PAIRS

- Main Idea:
  - Insert TRAP instructions in the beginning of code basic blocks.
  - Create two (or more) program copies in virtual memory.
  - Randomize the execution between the copies in runtime.



Original Copy

A0: TRAP
A1: MOVE
A2: ADD
A3: JUMP
B0: TRAP
B1: MOVE
B2: JUMP

Phantom Copy

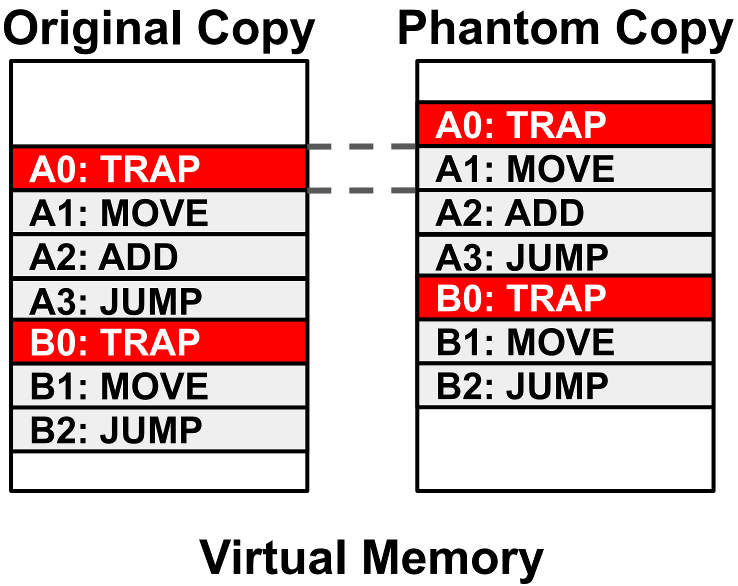
A0: TRAP
A1: MOVE
A2: ADD
A3: JUMP
B0: TRAP
B1: MOVE
B2: JUMP

Virtual Memory

[2] Mohamed et. al, PAIRS: Control flow protection using phantom addressed instructions. **[arXiv 2019]**

PAIRS

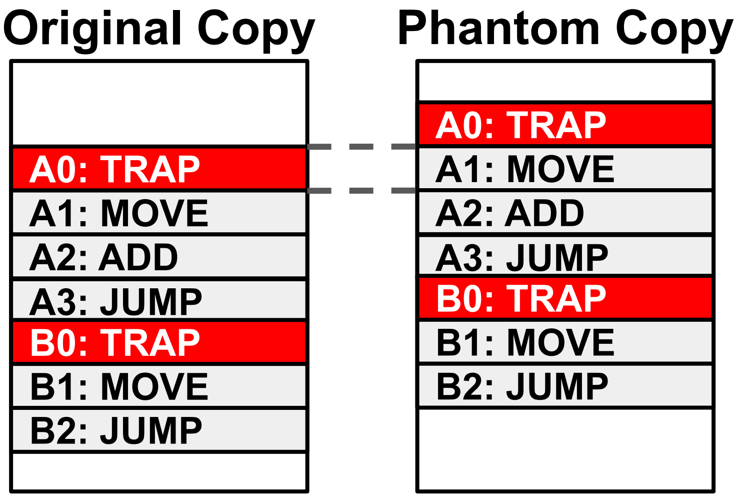
- Main Idea:
  - Insert TRAP instructions in the beginning of code basic blocks.
  - Create two (or more) program copies in virtual memory.
  - Randomize the execution between the copies in runtime.
- Pros and Cons:
  - + Negligible perf. overheads.



[2] Mohamed et. al, PAIRS: Control flow protection using phantom addressed instructions. **[arXiv 2019]**

PAIRS

- Main Idea:
  - Insert TRAP instructions in the beginning of code basic blocks.
  - Create two (or more) program copies in virtual memory.
  - Randomize the execution between the copies in runtime.
- Pros and Cons:
  - + Negligible perf. overheads.
  - - No code pointers protection.



Original Copy

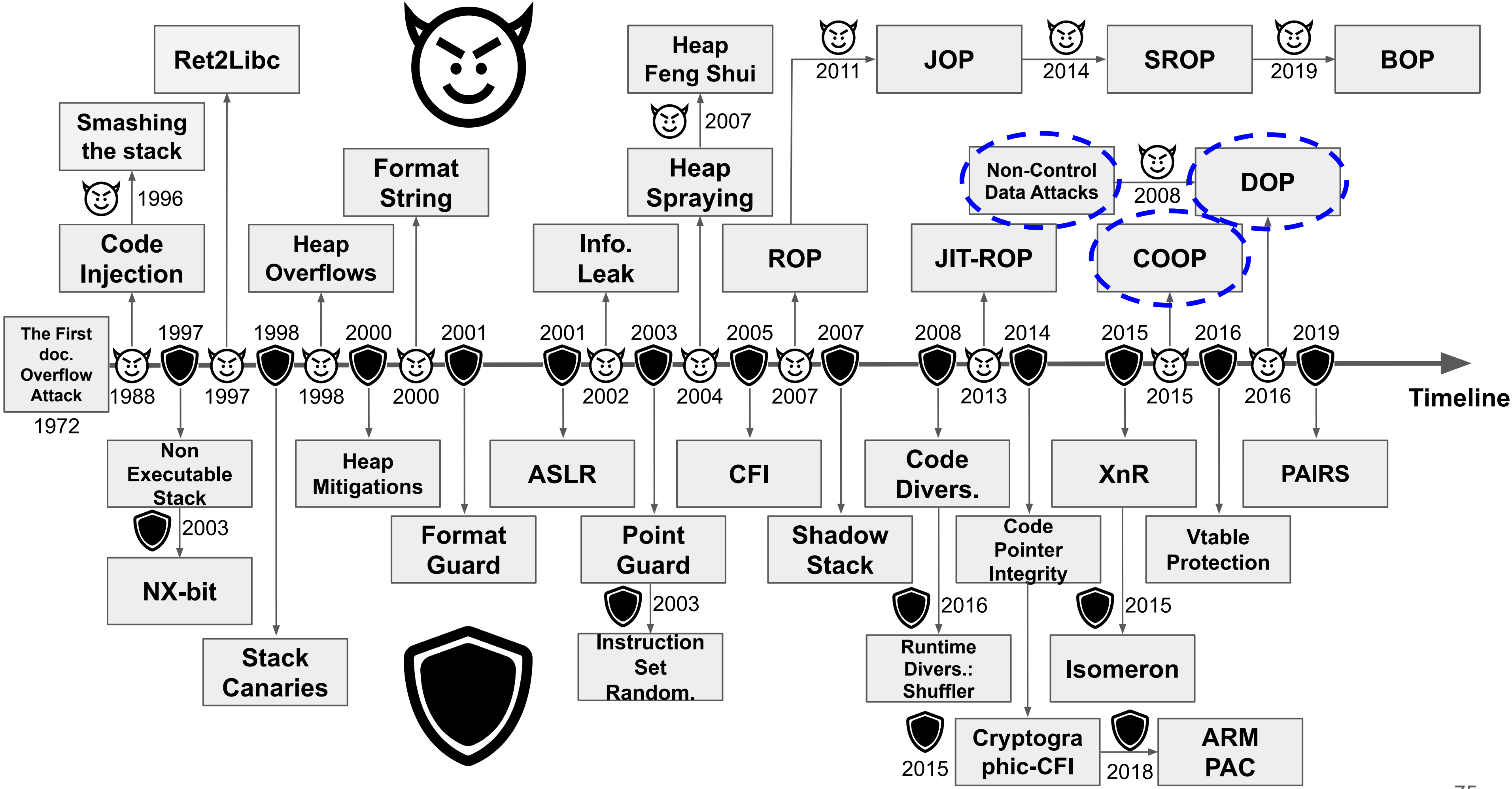
A0: TRAP
A1: MOVE
A2: ADD
A3: JUMP
B0: TRAP
B1: MOVE
B2: JUMP

Phantom Copy

A0: TRAP
A1: MOVE
A2: ADD
A3: JUMP
B0: TRAP
B1: MOVE
B2: JUMP

Virtual Memory

[2] Mohamed et. al, PAIRS: Control flow protection using phantom addressed instructions. **[arXiv 2019]**

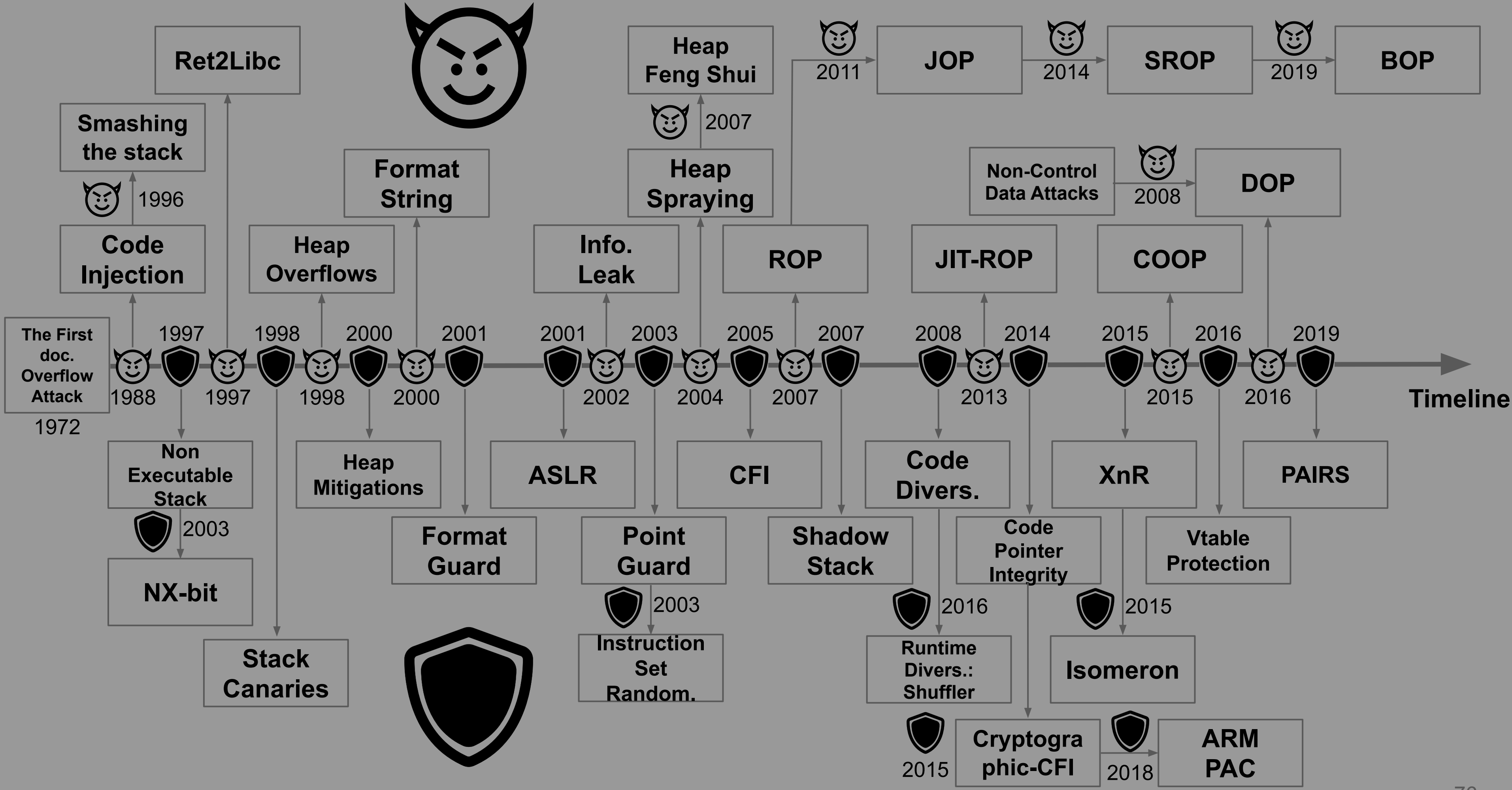


Timeline

The First doc. Overflow Attack
1972

1988 Code Injection
1996 Smashing the stack
1997 Non Executable Stack
2003 NX-bit
1997 Ret2Libc
1998 Stack Canaries
1998 Heap Overflows
2000 Heap Mitigations
2000 Format String
2001 Format Guard
2001 ASLR
2002 Info. Leak
2003 Point Guard
2003 Instruction Set Random.
2004 Heap Spraying
2007 Heap Feng Shui
2005 CFI
2007 ROP
2011 JOP
2014 SROP
2019 BOP
2007 Shadow Stack
2008 Code Divers.
2016 Runtime Divers.: Shuffler
2013 JIT-ROP
2014 Code Pointer Integrity
2015 Cryptographic-CFI
2018 ARM PAC
Non-Control Data Attacks
2008 DOP
2015 XnR
2015 Isomeron
2015 COOP
2016 Vtable Protection
2016 DOP
2019 PAIRS

[10] Cheng et. al., Exploitation techniques and defenses for data-oriented attacks. **[SecDev 2019]**



The First doc. Overflow Attack

1972

1988 Code Injection

1996 Smashing the stack

1997 Non Executable Stack

2003 NX-bit

1997 Ret2Libc

1998 Stack Canaries

1998 Heap Overflows

2000 Heap Mitigations

2000 Format String

2001 Format Guard

2001 ASLR

2002 Info. Leak

2003 Point Guard

2003 Instruction Set Random.

2004 Heap Spraying

2007 Heap Feng Shui

2005 CFI

2007 ROP

2011 JOP

2014 SROP

2019 BOP

2007 Shadow Stack

2008 Code Divers.

2016 Runtime Divers.: Shuffler

2013 JIT-ROP

2014 Code Pointer Integrity

2015 Cryptographic-CFI

2018 ARM PAC

2015 XnR

2015 Isomeron

2015 COOP

2016 Vtable Protection

Non-Control Data Attacks

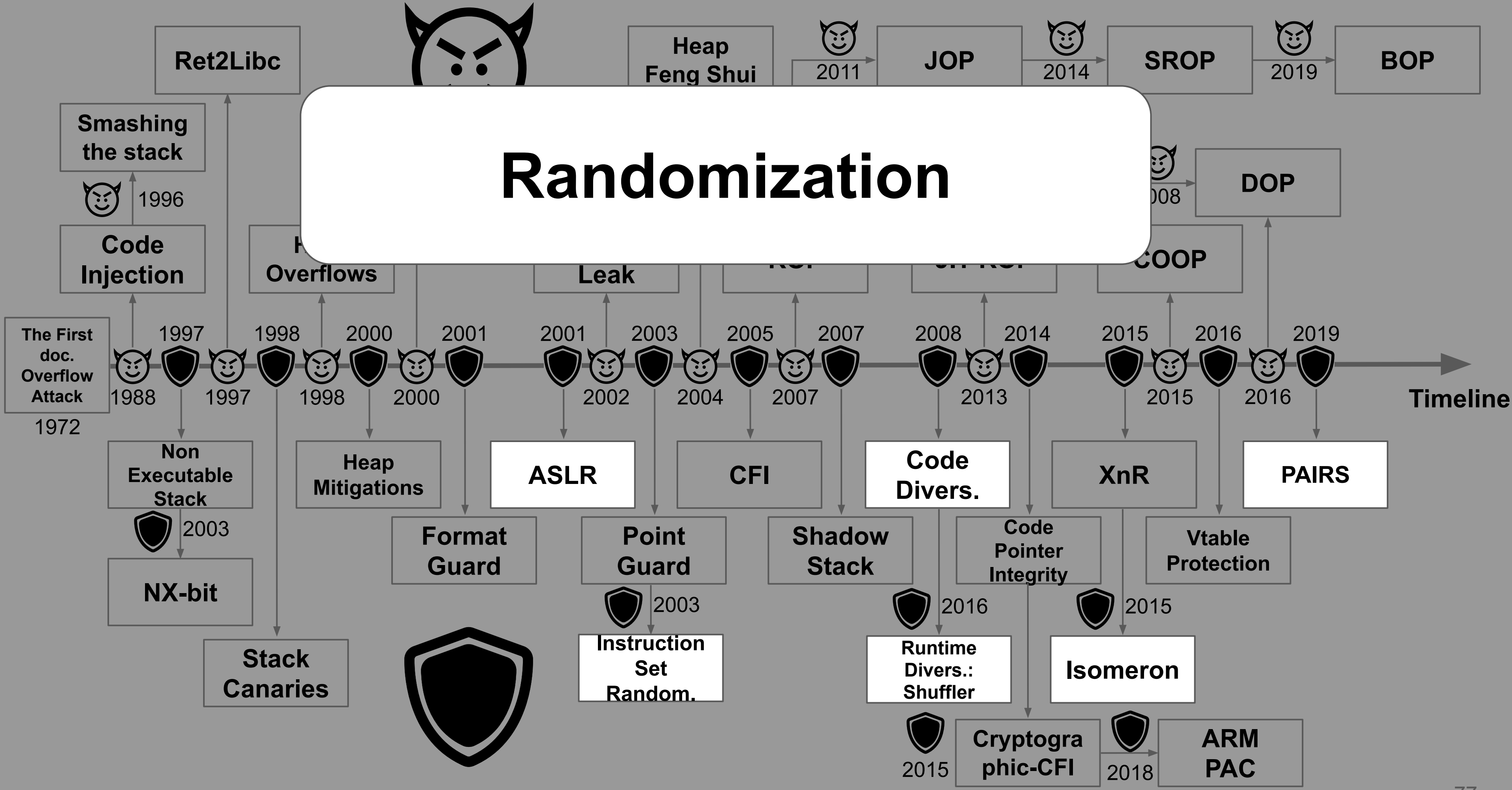
2008 DOP

2016

2019 PAIRS

Timeline

Randomization



The First doc. Overflow Attack
1972

Code Injection
1988

Smashing the stack
1996

Ret2Libc
1997

Non Executable Stack
1997

NX-bit
2003

Stack Canaries
1998

H
Overflows
1998

Heap Mitigations
2000

2000

Format Guard
2001

ASLR
2001

Leak
2002

Point Guard
2003

Instruction Set Random.
2003

Heap Feng Shui
2004

CFI
2005

2007

Shadow Stack
2007

JOP
2011

SROP
2014

BOP
2019

Code Divers.
2008

2013

Code Pointer Integrity
2014

Runtime Divers.: Shuffler
2016

Cryptographic-CFI
2015

ARM PAC
2018

XnR
2015

Isomeron
2015

COOP
2015

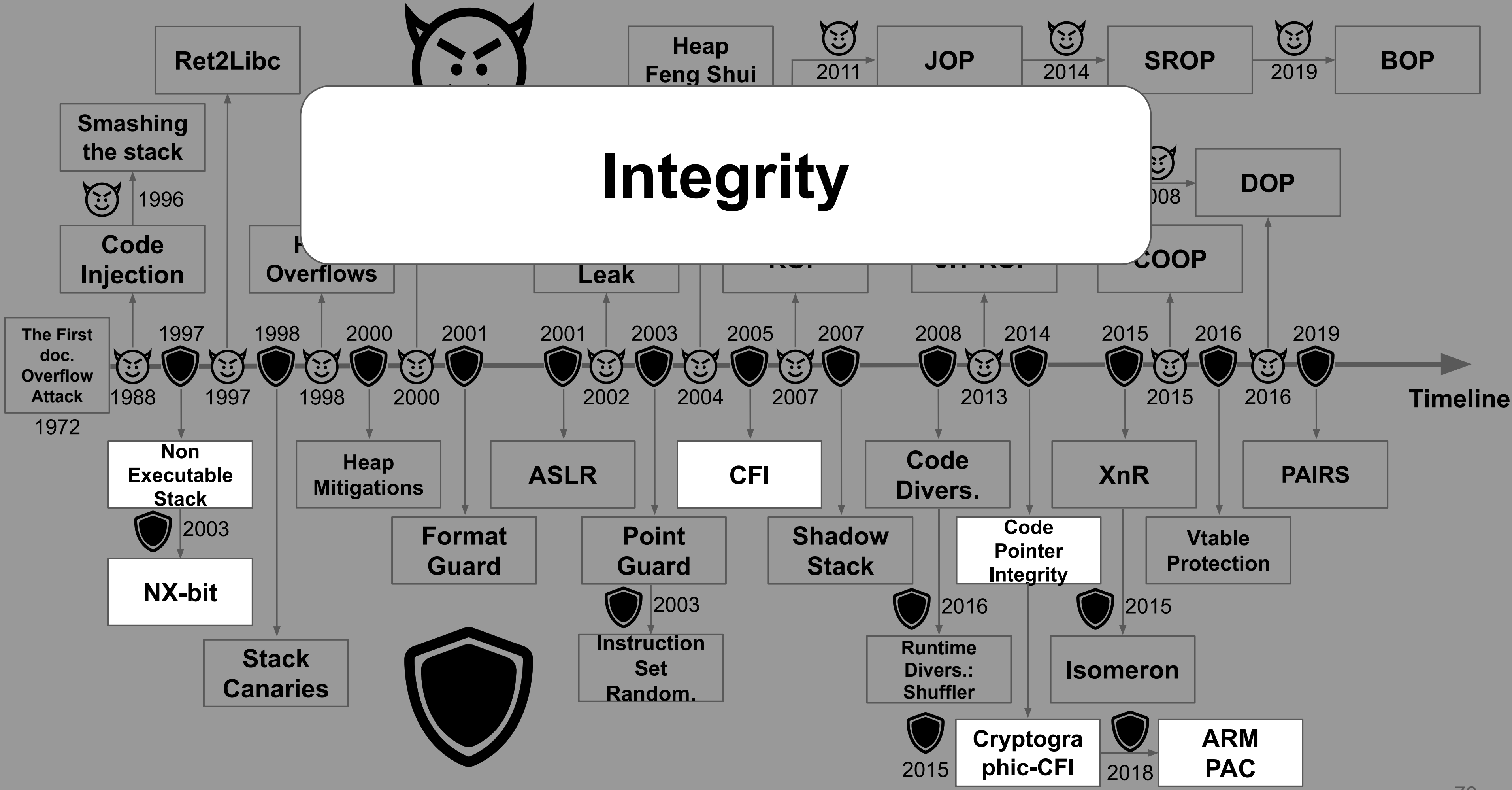
Vtable Protection
2016

DOP
2016
008

PAIRS
2019

Timeline

Integrity



The First doc. Overflow Attack
1972

Code Injection
1988

Smashing the stack
1996

Ret2Libc
1997

H
Overflows
1998

Leak
2002

Heap Feng Shui
2004

2007

2011
JOP

2013

2014
SROP

2019
BOP

COOP
2015

08
DOP
2016

Timeline

Non Executable Stack
1997

2003
NX-bit

Stack Canaries
1998

Heap Mitigations
2000

Format Guard
2001

ASLR
2001

Point Guard
2003

2003
Instruction Set Random.

CFI
2005

Shadow Stack
2007

Code Divers.
2008

2016
Runtime Divers.: Shuffler

Code Pointer Integrity
2014

2015
Cryptographic-CFI

2018
ARM PAC

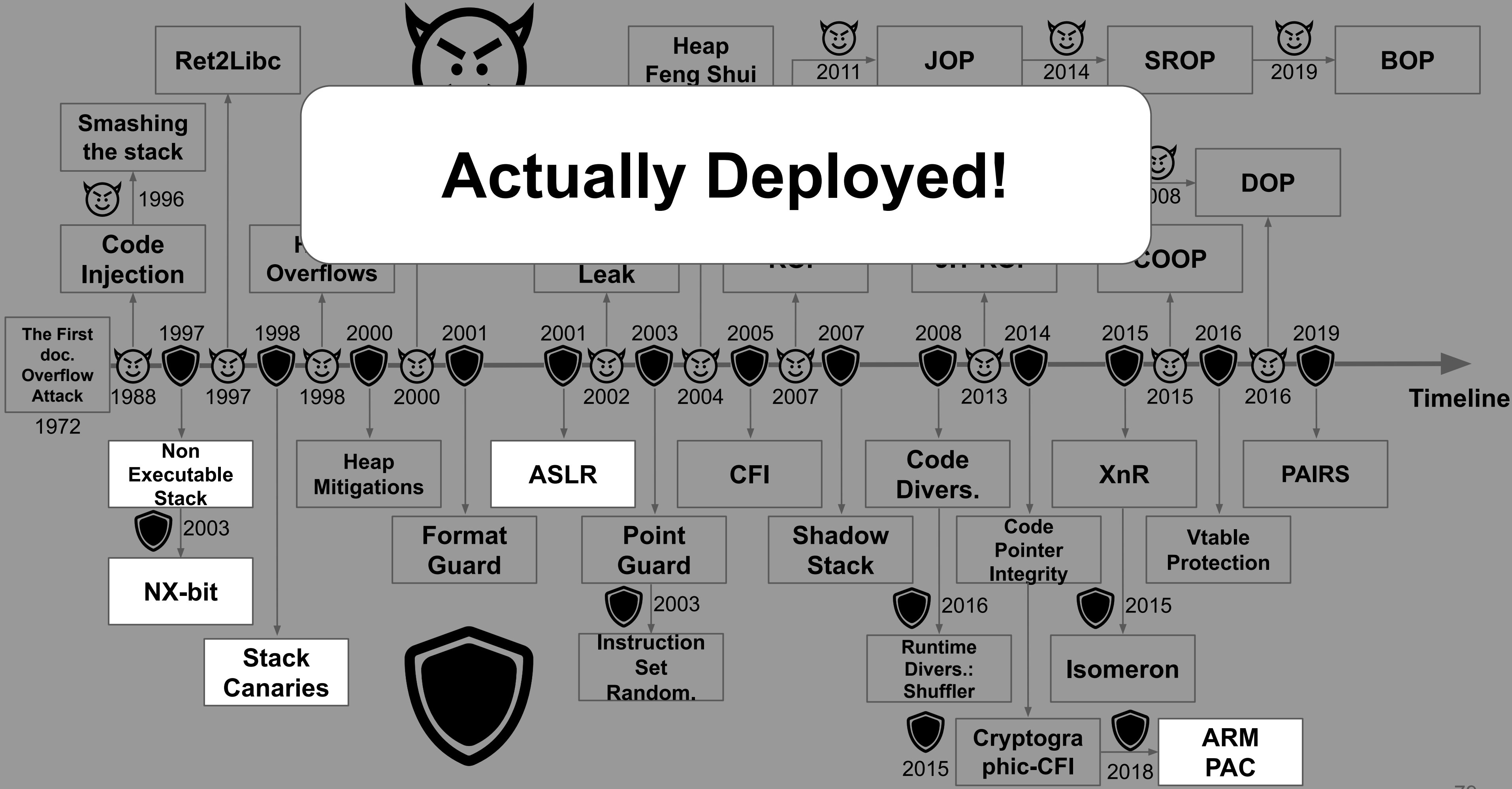
XnR
2015

2015
Isomeron

Vtable Protection
2016

PAIRS
2019

Actually Deployed!



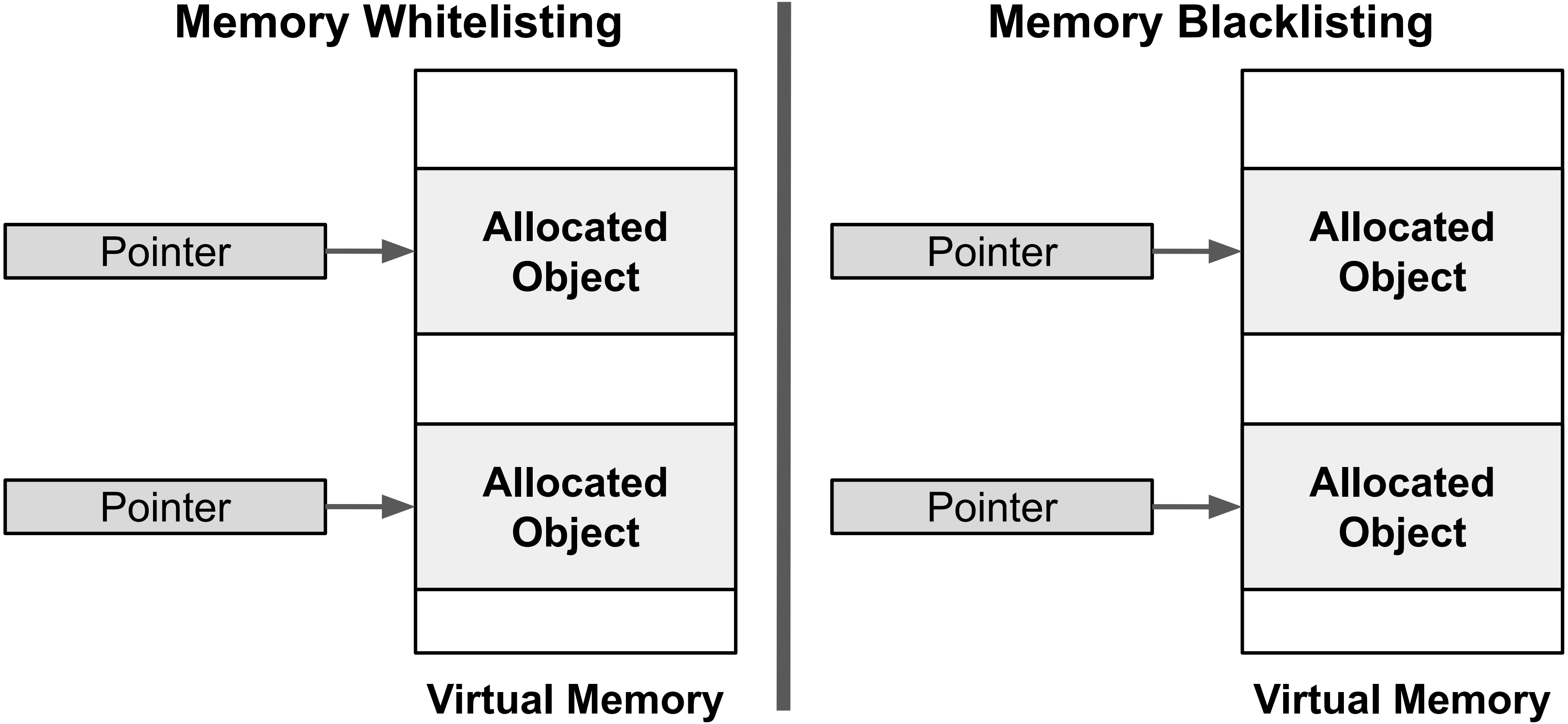
Memory Safety Techniques

MEMORY SAFETY TECHNIQUES

Spatial Memory Safety

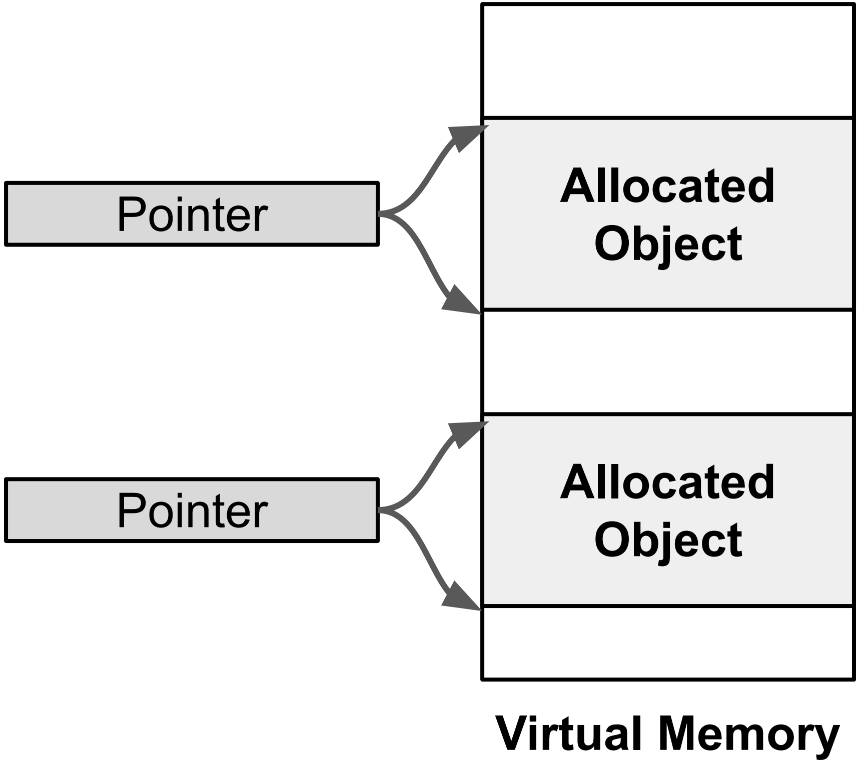
Temporal Memory Safety

SPATIAL MEMORY SAFETY

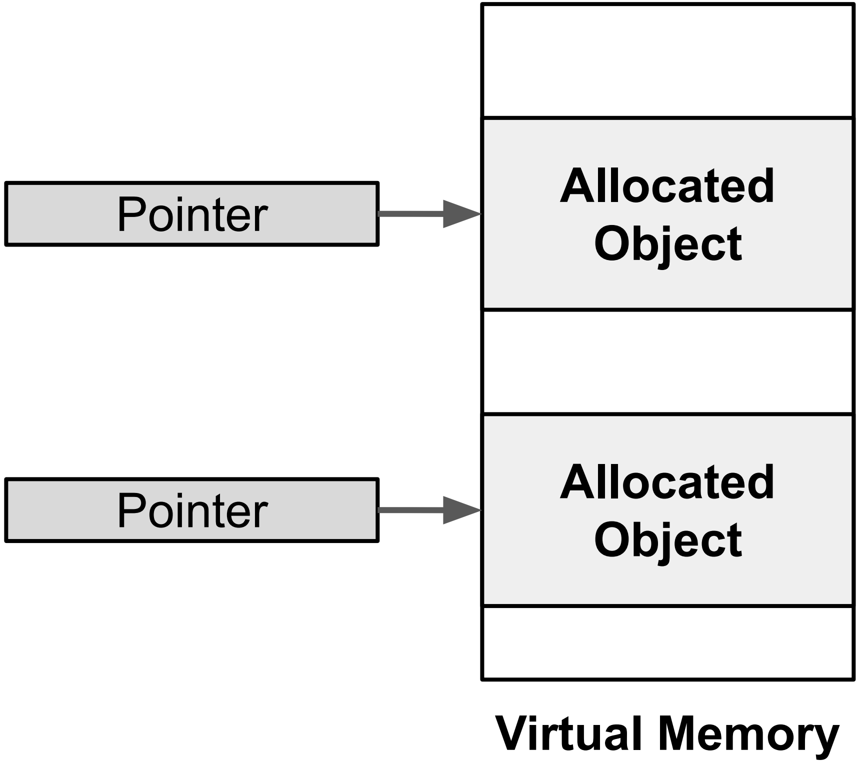


SPATIAL MEMORY SAFETY

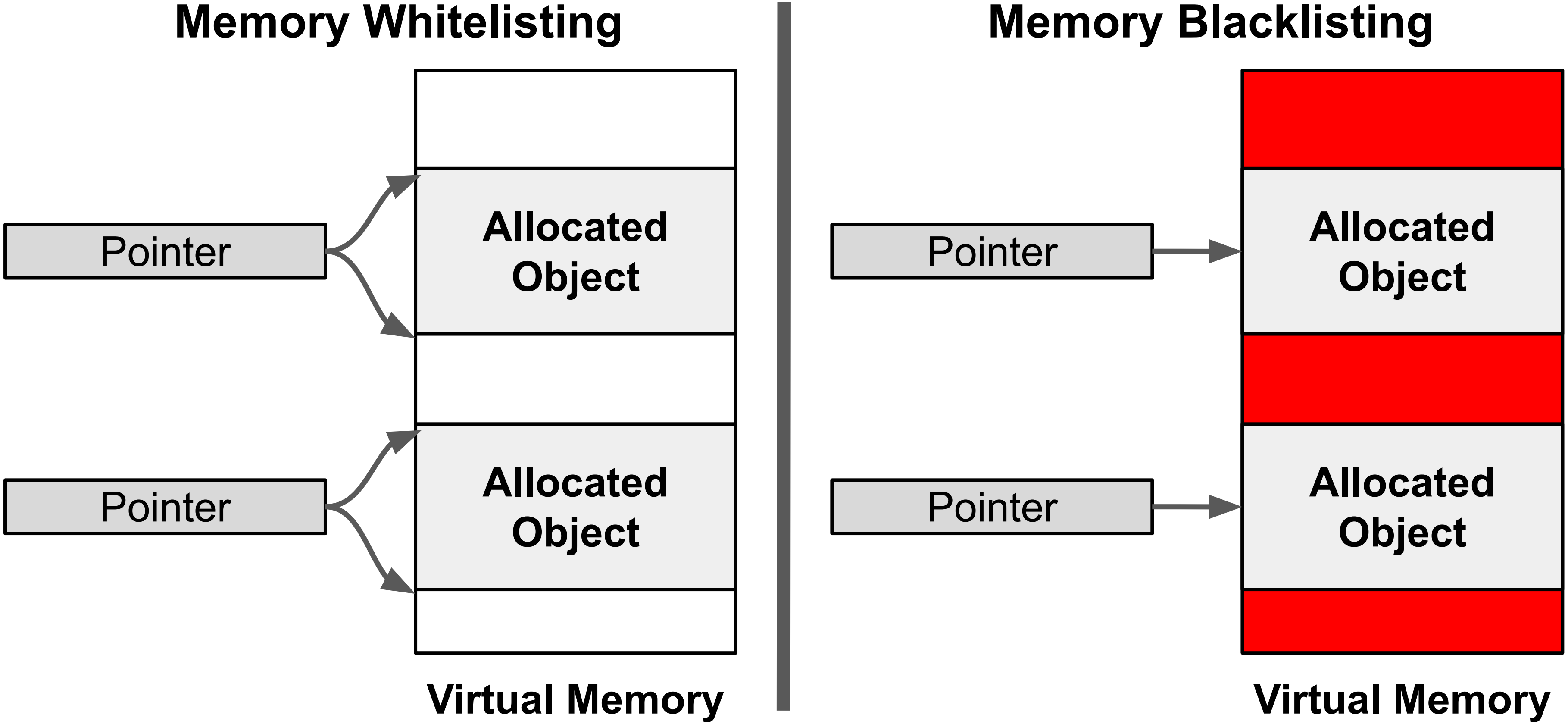
Memory Whitelisting



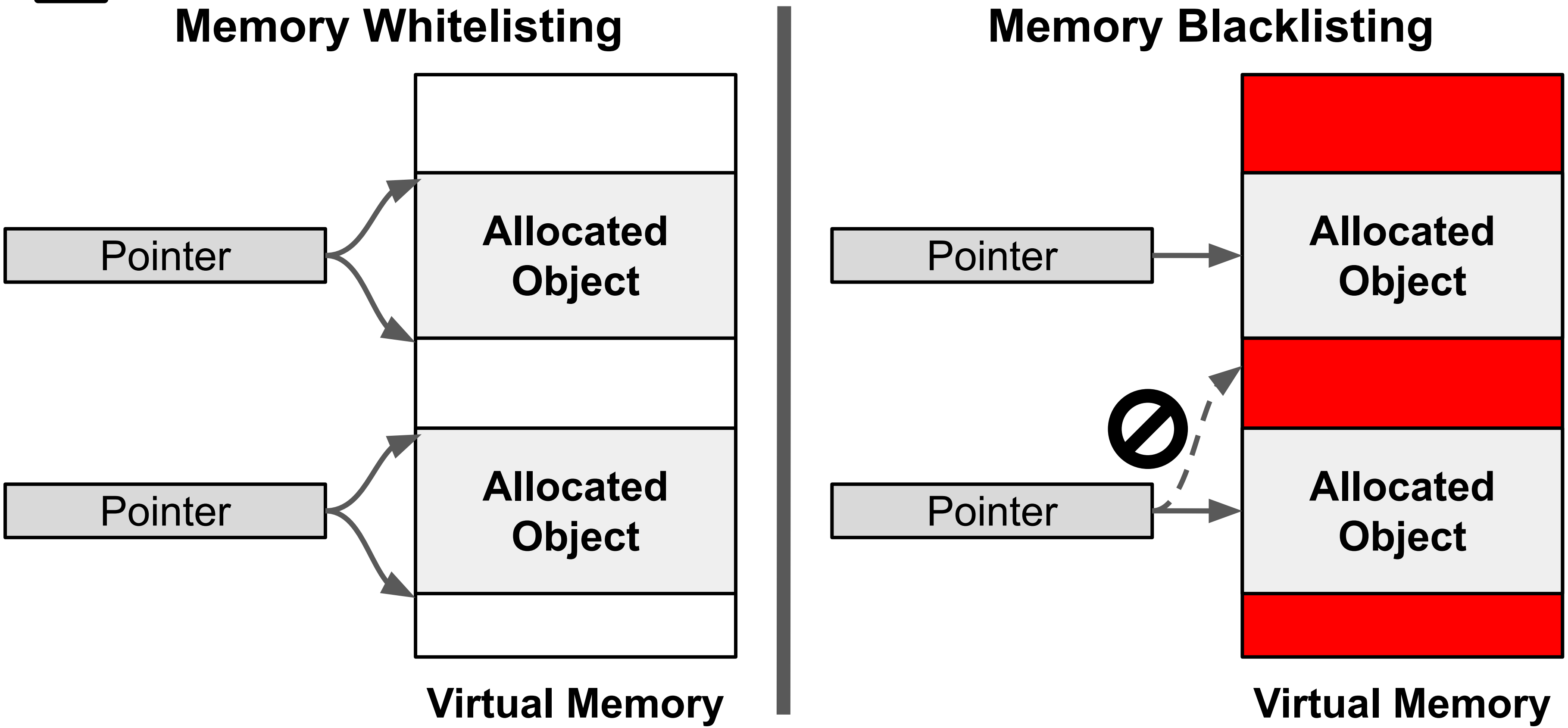
Memory Blacklisting



SPATIAL MEMORY SAFETY



SPATIAL MEMORY SAFETY



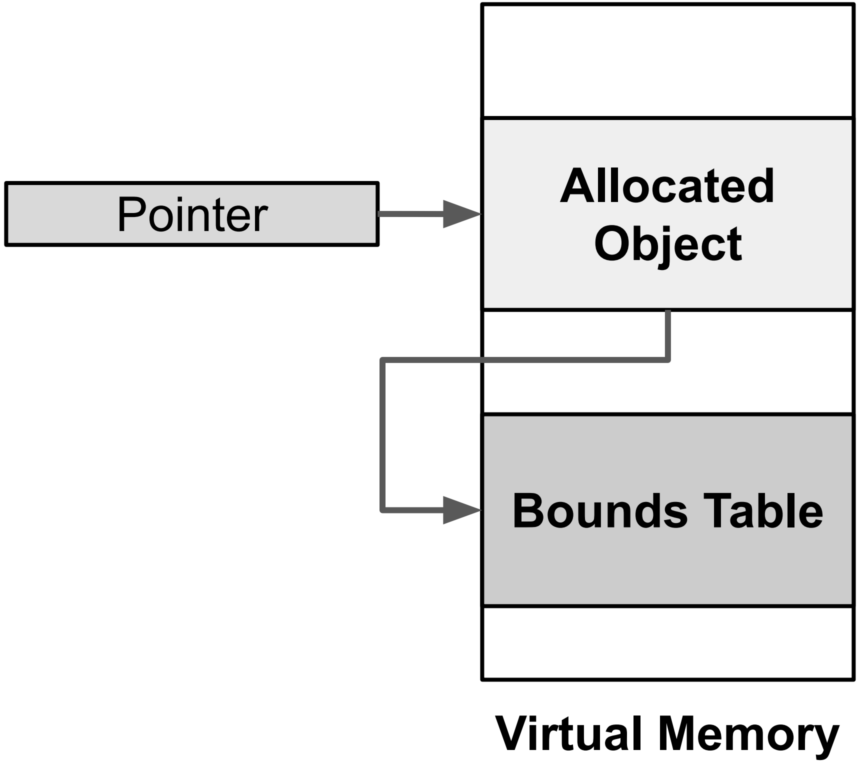
	Whitelisting		**Blacklisting**
	Per-object	**Per-pointer**	
Disjoint Metadata	Compatible C Baggy Bounds	Mondrian M-machine Softbound, Hardbound Watchdog CUP, MPX	Purify Valgrind Dr. Memory Electric Fence ASan
Inlined Metadata	EffectiveSan	(aka Fat Pointers) CHERI Cyclone CheckedC	SafeMem REST CALIFORMS
Co-joined Metadata	ARM Memory Tagging SPARC ADI		
No Metadata	Lowfat s/w	Lowfat h/w	

	Whitelisting		Blacklisting
	Per-object	**Per-pointer**	
Disjoint Metadata	Compatible C Baggy Bounds	Mondrian M-machine Softbound, Hardbound Watchdog CUP, MPX	Purify Valgrind Dr. Memory Electric Fence ASan
Inlined Metadata	EffectiveSan	(aka Fat Pointers) CHERI Cyclone CheckedC	SafeMem REST CALIFORMS
Co-joined Metadata	ARM Memory Tagging SPARC ADI		
No Metadata	Lowfat s/w	Lowfat h/w	

	Whitelisting		Blacklisting
	Per-object	**Per-pointer**	
Disjoint Metadata	Compatible C Baggy Bounds	Mondrian M-machine Softbound, Hardbound Watchdog CUP, MPX	Purify Valgrind Dr. Memory Electric Fence ASan
Inlined Metadata	EffectiveSan	(aka Fat Pointers) CHERI Cyclone CheckedC	SafeMem REST CALIFORMS
Co-joined Metadata	ARM Memory Tagging SPARC ADI		
No Metadata	Lowfat s/w	Lowfat h/w	

WHITELISTING: PER-OBJECT DISJOINT METADATA

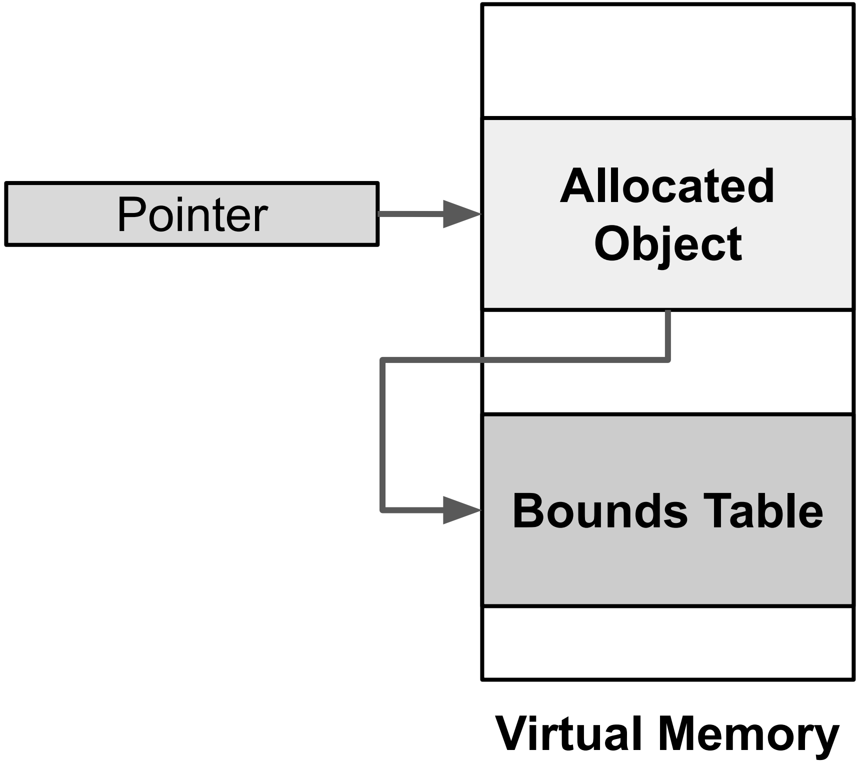
- Main Idea:
  - Record bounds information for each **object** in a bounds table.
  - Check on pointer **arithmetic**.



[19] Jones and Kelly. Backwards-compatible bounds checking for arrays and pointers. **[DEBUG 1997]**

WHITELISTING: PER-OBJECT DISJOINT METADATA

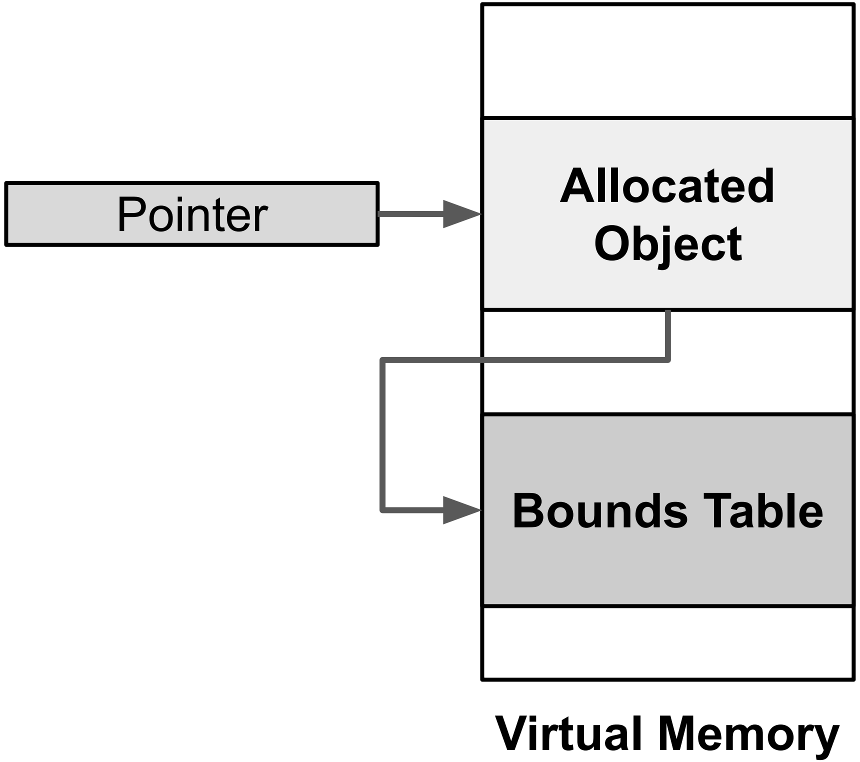
- Main Idea:
  - Record bounds information for each **object** in a bounds table.
  - Check on pointer **arithmetic**.
- Pros and Cons:
  - + Good binary compatibility.



[19] Jones and Kelly. Backwards-compatible bounds checking for arrays and pointers. **[DEBUG 1997]**

WHITELISTING: PER-OBJECT DISJOINT METADATA

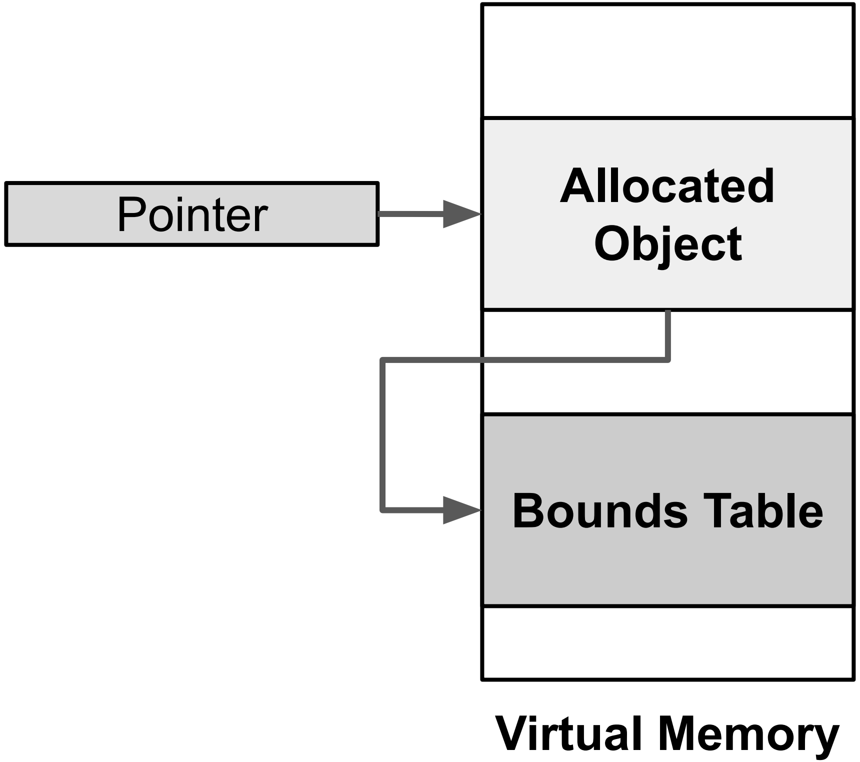
- Main Idea:
  - Record bounds information for each **object** in a bounds table.
  - Check on pointer **arithmetic**.
- Pros and Cons:
  - + Good binary compatibility.
  - - Costly range lookups.



[19] Jones and Kelly. Backwards-compatible bounds checking for arrays and pointers. **[DEBUG 1997]**

WHITELISTING: PER-OBJECT DISJOINT METADATA

- Main Idea:
  - Record bounds information for each **object** in a bounds table.
  - Check on pointer **arithmetic**.
- Pros and Cons:
  - + Good binary compatibility.
  - - Costly range lookups.
  - - No intra-object protection.

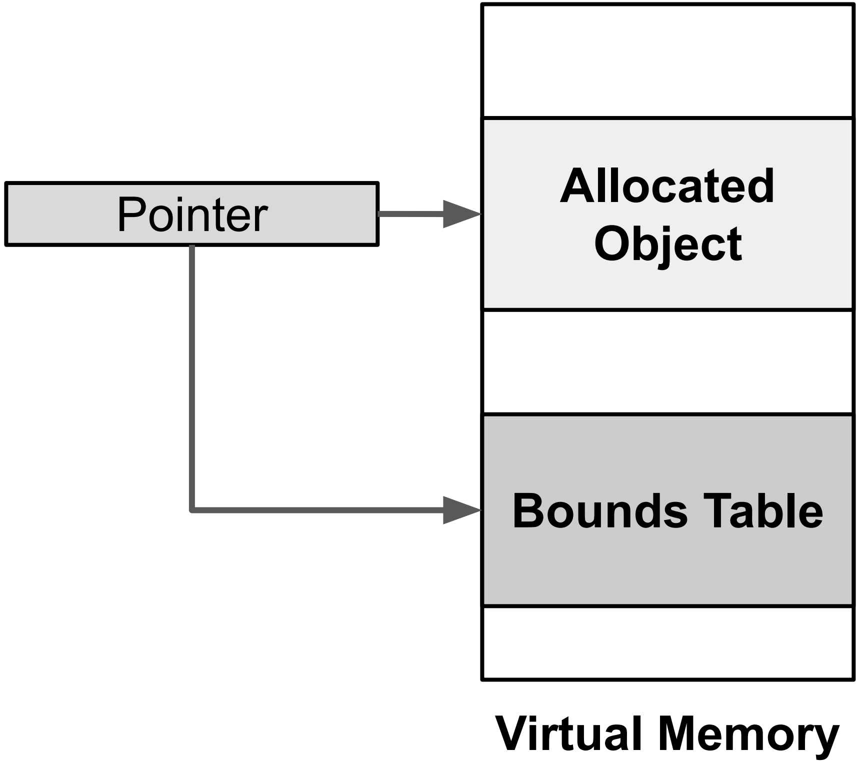


[19] Jones and Kelly. Backwards-compatible bounds checking for arrays and pointers. **[DEBUG 1997]**

	Whitelisting		Blacklisting
	Per-object	**Per-pointer**	
Disjoint Metadata	Compatible C Baggy Bounds	Mondrian M-machine Softbound, Hardbound Watchdog CUP, MPX	Purify Valgrind Dr. Memory Electric Fence ASan
Inlined Metadata	EffectiveSan	(aka Fat Pointers) CHERI Cyclone CheckedC	SafeMem REST CALIFORMS
Co-joined Metadata	ARM Memory Tagging SPARC ADI		
No Metadata	Lowfat s/w	Lowfat h/w	

WHITELISTING: PER-POINTER DISJOINT METADATA

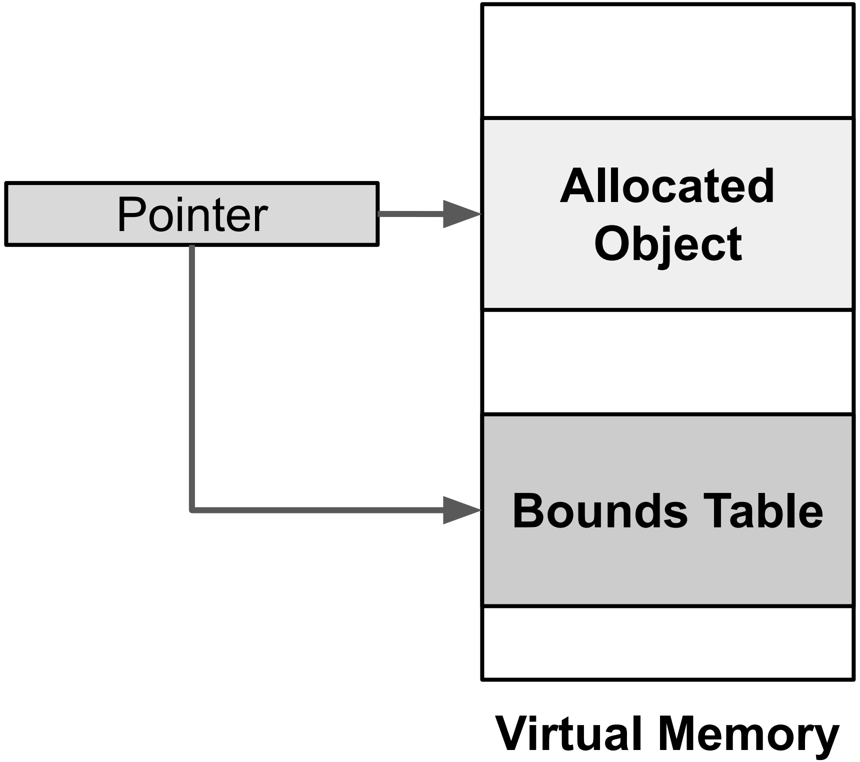
- Main Idea:
  - Record bounds information for each **pointer** in a bounds table.
  - Propagate metadata on pointer arithmetic.
  - Check on pointer **dereference**.



[17] Santosh Nagarakatte. Practical Low-Overhead Enforcement of Memory Safety. **[PhD Thesis 2012]**

WHITELISTING: PER-POINTER DISJOINT METADATA

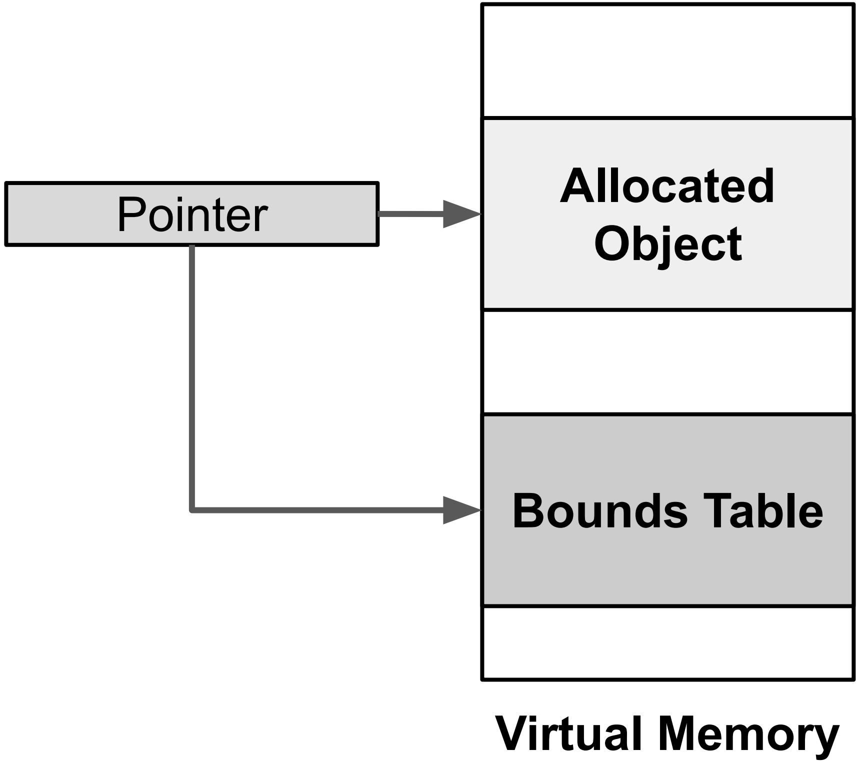
- Main Idea:
  - Record bounds information for each **pointer** in a bounds table.
  - Propagate metadata on pointer arithmetic.
  - Check on pointer **dereference**.
- Pros and Cons:
  - ✚ Lower number of checks.



[17] Santosh Nagarakatte. Practical Low-Overhead Enforcement of Memory Safety. **[PhD Thesis 2012]**

WHITELISTING: PER-POINTER DISJOINT METADATA

- Main Idea:
  - Record bounds information for each **pointer** in a bounds table.
  - Propagate metadata on pointer arithmetic.
  - Check on pointer **dereference**.
- Pros and Cons:
  - + Lower number of checks.
  - + Good binary compatibility.
  - - Problematic for multi-threading.



[17] Santosh Nagarakatte. Practical Low-Overhead Enforcement of Memory Safety. **[PhD Thesis 2012]**

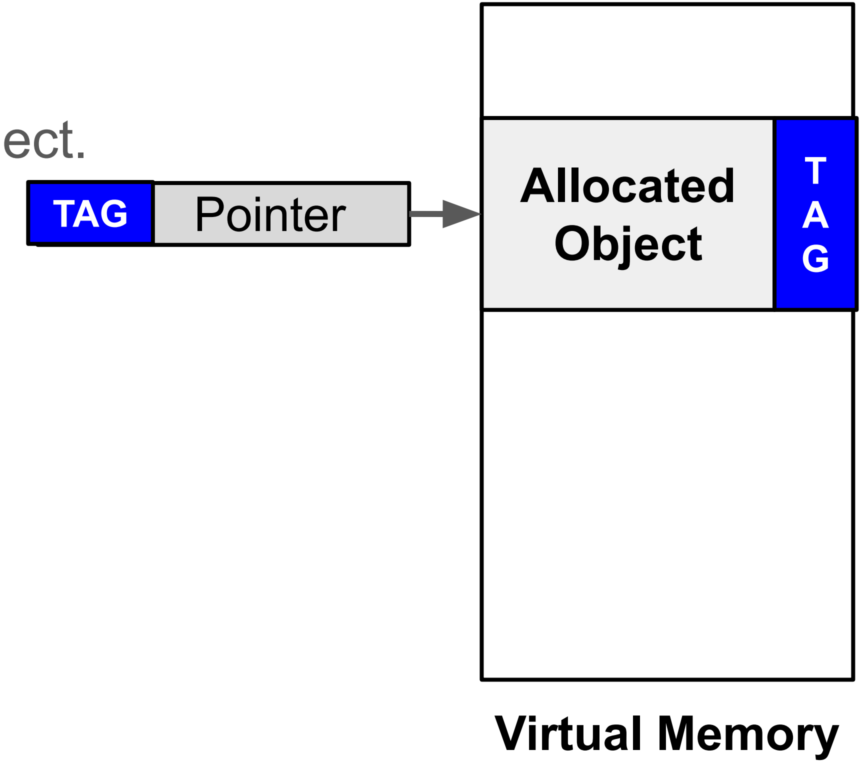
	Whitelisting		Blacklisting
	Per-object	Per-pointer	
Disjoint Metadata	Compatible C Baggy Bounds	Mondrian M-machine Softbound, Hardbound Watchdog CUP, MPX	Purify Valgrind Dr. Memory Electric Fence ASan
Inlined Metadata	EffectiveSan	(aka Fat Pointers) CHERI Cyclone CheckedC	SafeMem REST CALIFORMS
Co-joined Metadata	ARM Memory Tagging SPARC ADI		
No Metadata	Lowfat s/w	Lowfat h/w	

[18] Alexandre Joannou. Optimizing the CHERI capability machine. **[PhD Thesis 2019]**

	Whitelisting		Blacklisting
	Per-object	Per-pointer	
Disjoint Metadata	Compatible C Baggy Bounds	Mondrian M-machine Softbound, Hardbound Watchdog CUP, MPX	Purify Valgrind Dr. Memory Electric Fence ASan
Inlined Metadata	EffectiveSan	(aka Fat Pointers) CHERI Cyclone CheckedC	SafeMem REST CALIFORMS
Co-joined Metadata	ARM Memory Tagging SPARC ADI		
No Metadata	Lowfat s/w	Lowfat h/w	

WHITELISTING: CO-JOINED METADATA

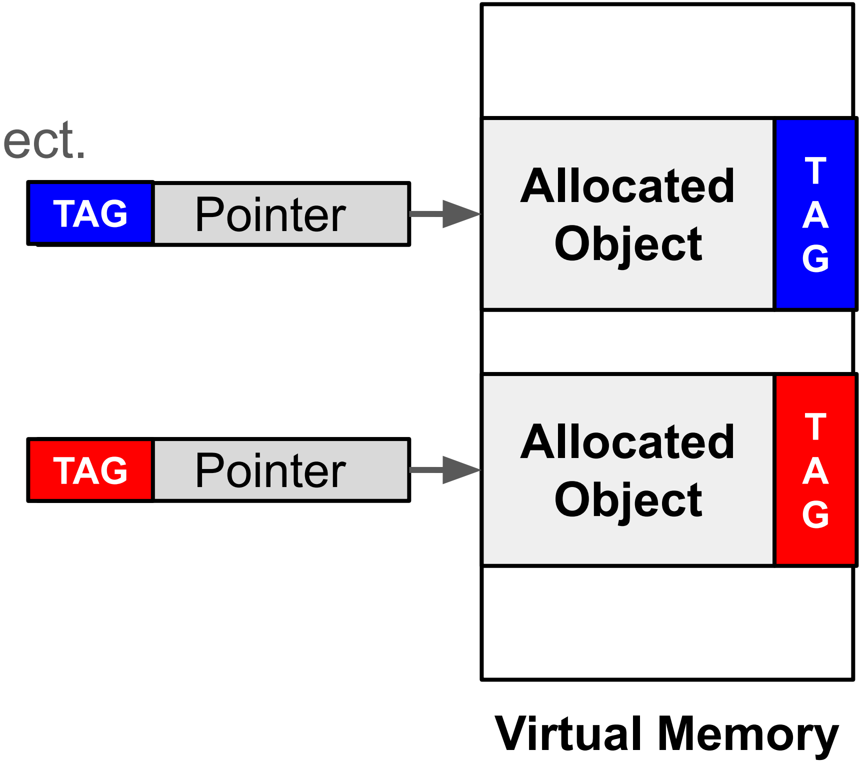
- Main Idea:
  - Assign a tag for each pointer/object.



[34] Serebryany et. al., Memory tagging and how it improves C/C++ memory safety. **[arXiv 2018]**

WHITELISTING: CO-JOINED METADATA

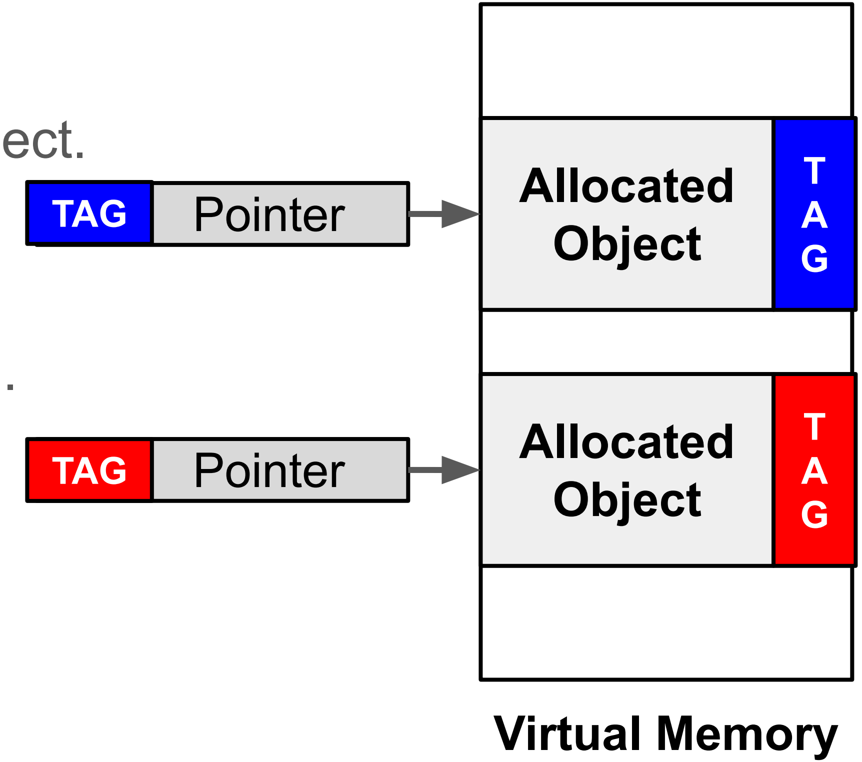
- Main Idea:
  - Assign a tag for each pointer/object.
  - Compare tags on pointer **dereference**.



[34] Serebryany et. al., Memory tagging and how it improves C/C++ memory safety. **[arXiv 2018]**

WHITELISTING: CO-JOINED METADATA

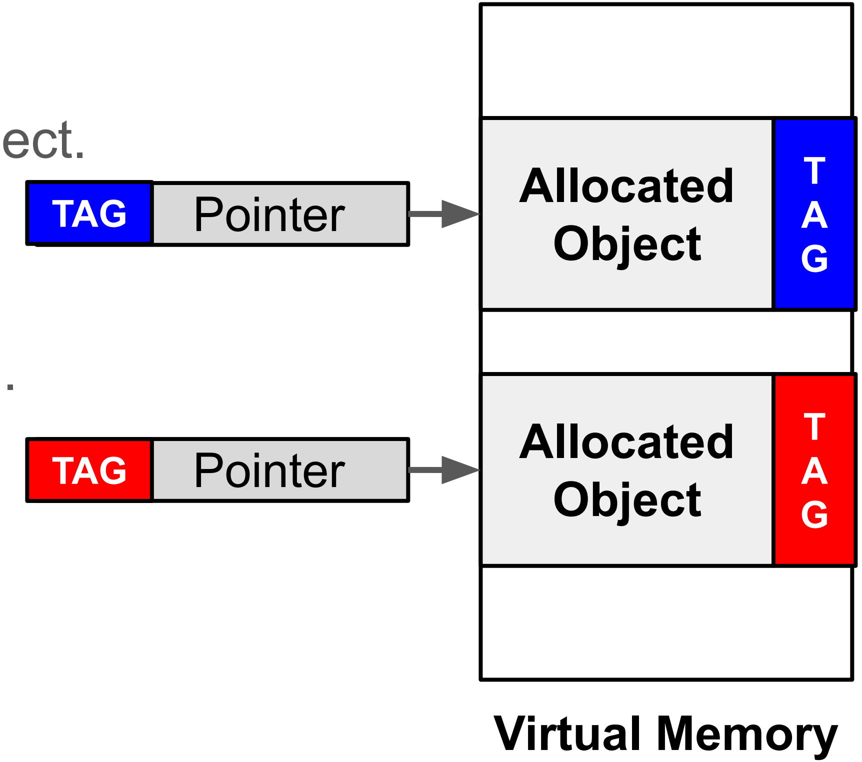
- Main Idea:
  - Assign a tag for each pointer/object.
  - Compare tags on pointer **dereference**.
  - E.g., SPARC ADI and ARM MTE.



[34] Serebryany et. al., Memory tagging and how it improves C/C++ memory safety. **[arXiv 2018]**

WHITELISTING: CO-JOINED METADATA

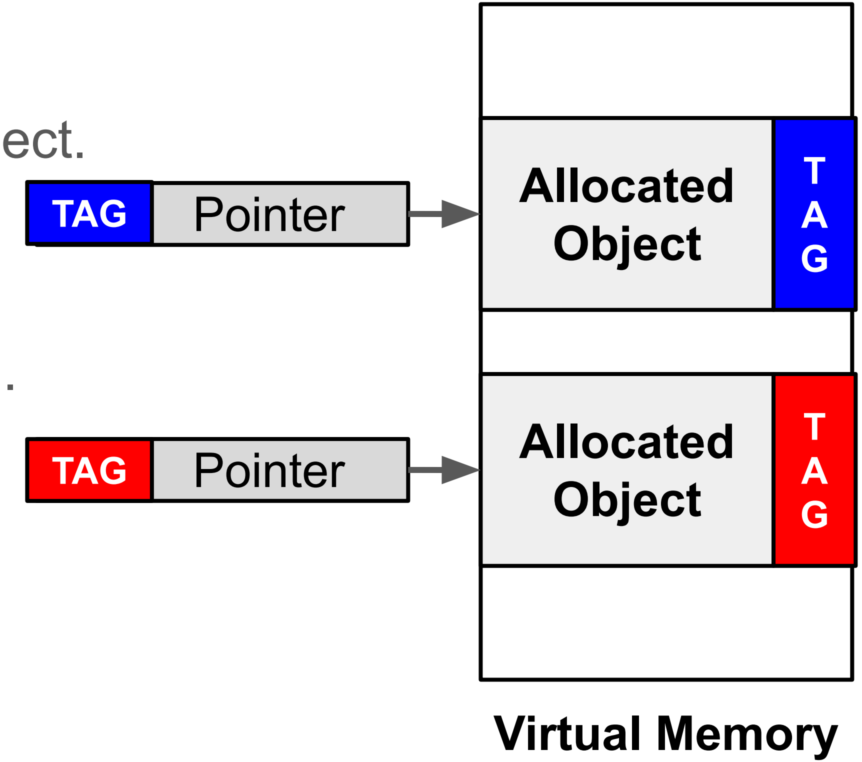
- Main Idea:
  - Assign a tag for each pointer/object.
  - Compare tags on pointer **dereference**.
  - E.g., SPARC ADI and ARM MTE.
- Pros and Cons:
  - + Efficient check in hardware.
  - - Limited entropy.



[33] Bialek et. al., Security analysis of memory tagging. **[Microsoft Research 2020]**

WHITELISTING: CO-JOINED METADATA

- Main Idea:
  - Assign a tag for each pointer/object.
  - Compare tags on pointer **dereference**.
  - E.g., SPARC ADI and ARM MTE.
- Pros and Cons:
  - + Efficient check in hardware.
  - - Limited entropy.
  - - No intra-object protection.
  - - Only for 64-bit systems.

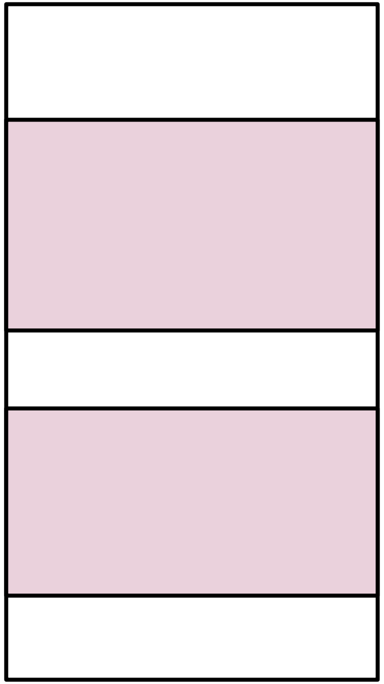


[33] Bialek et. al., Security analysis of memory tagging. **[Microsoft Research 2020]**

	Whitelisting		**Blacklisting**
	Per-object	**Per-pointer**	
Disjoint Metadata	Compatible C Baggy Bounds	Mondrian M-machine Softbound, Hardbound Watchdog CUP, MPX	Purify Valgrind Dr. Memory Electric Fence ASan
Inlined Metadata	EffectiveSan	(aka Fat Pointers) CHERI Cyclone CheckedC	SafeMem REST CALIFORMS
Co-joined Metadata	ARM Memory Tagging SPARC ADI		
No Metadata	Lowfat s/w	Lowfat h/w	

WHITELISTING: NO METADATA (**LOWFAT**)

- Main Idea:
  - Partition the heap into equally-sized regions.

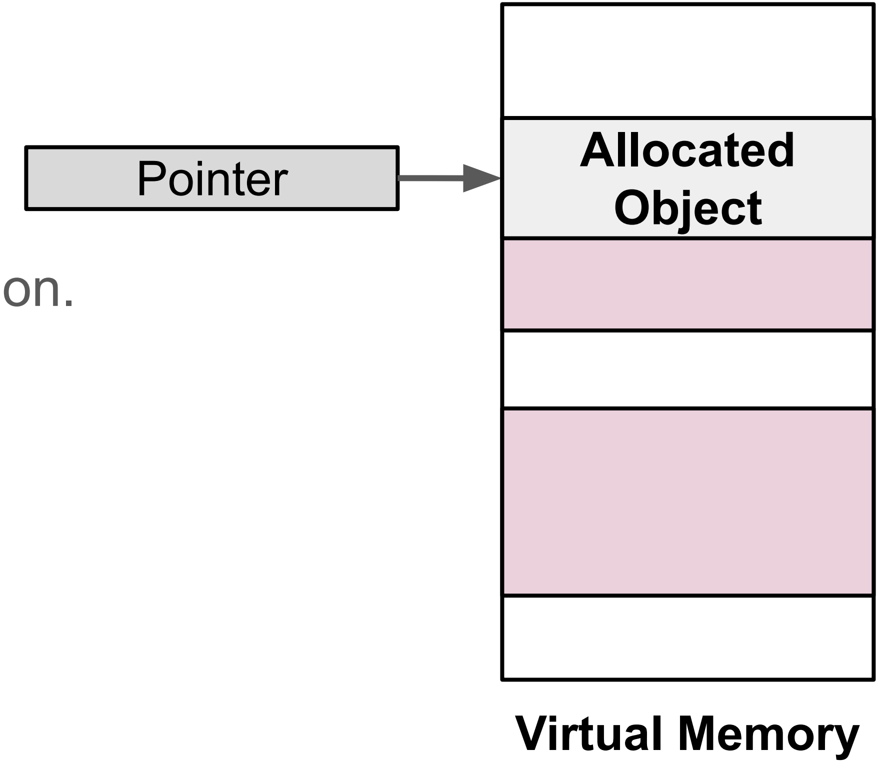


Virtual Memory

[35] Duck & Yap, Heap bounds protection with low fat pointers. **[CC 2016]**

WHITELISTING: NO METADATA (**LOWFAT**)

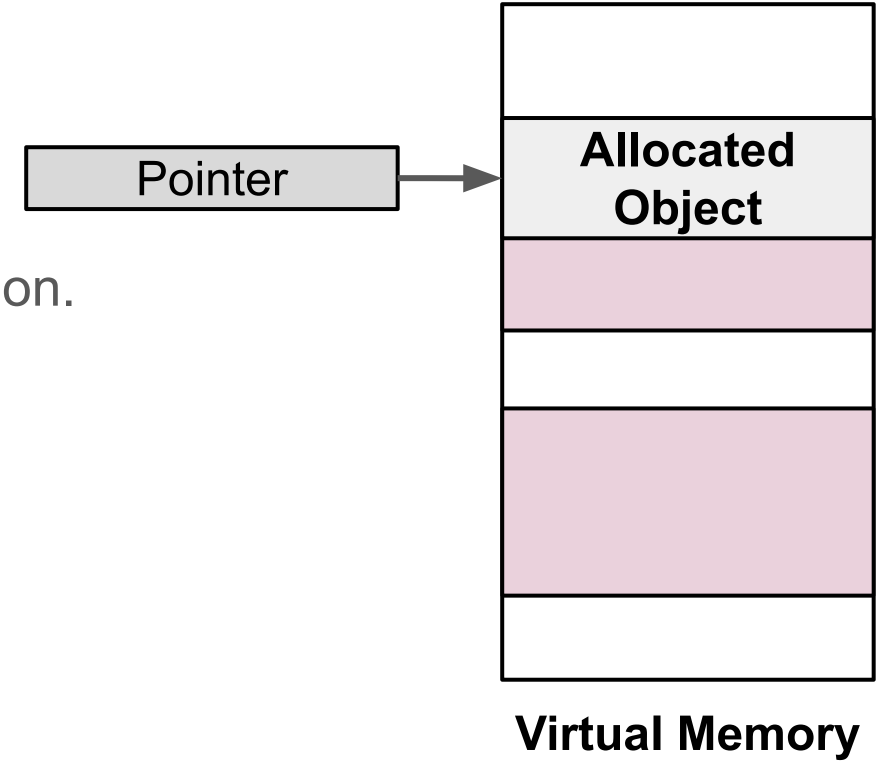
- Main Idea:
  - Partition the heap into equally-sized regions.
  - Use one allocation size per region.



[35] Duck & Yap, Heap bounds protection with low fat pointers. **[CC 2016]**

WHITELISTING: NO METADATA (**LOWFAT**)

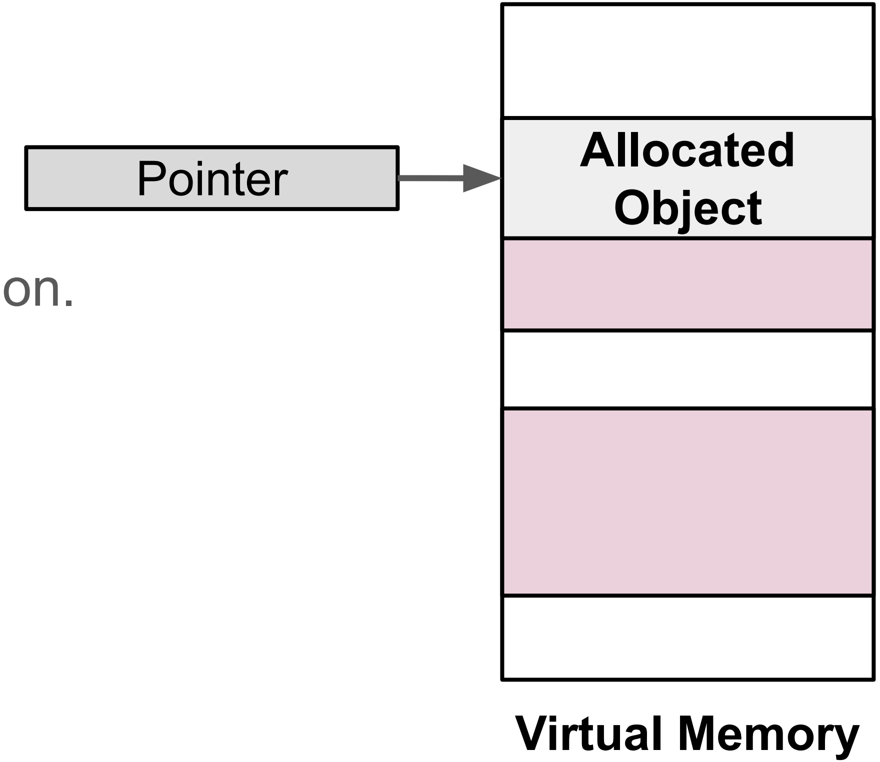
- Main Idea:
  - Partition the heap into equally-sized regions.
  - Use one allocation size per region.
  - Check on pointer **arithmetic**.



[35] Duck & Yap, Heap bounds protection with low fat pointers. **[CC 2016]**

WHITELISTING: NO METADATA (**LOWFAT**)

- Main Idea:
  - Partition the heap into equally-sized regions.
  - Use one allocation size per region.
  - Check on pointer **arithmetic**.
- Pros and Cons:
  - + Good binary compatibility.
  - - Memory fragmentation.
  - - No intra-object protection.



[35] Duck & Yap, Heap bounds protection with low fat pointers. **[CC 2016]**

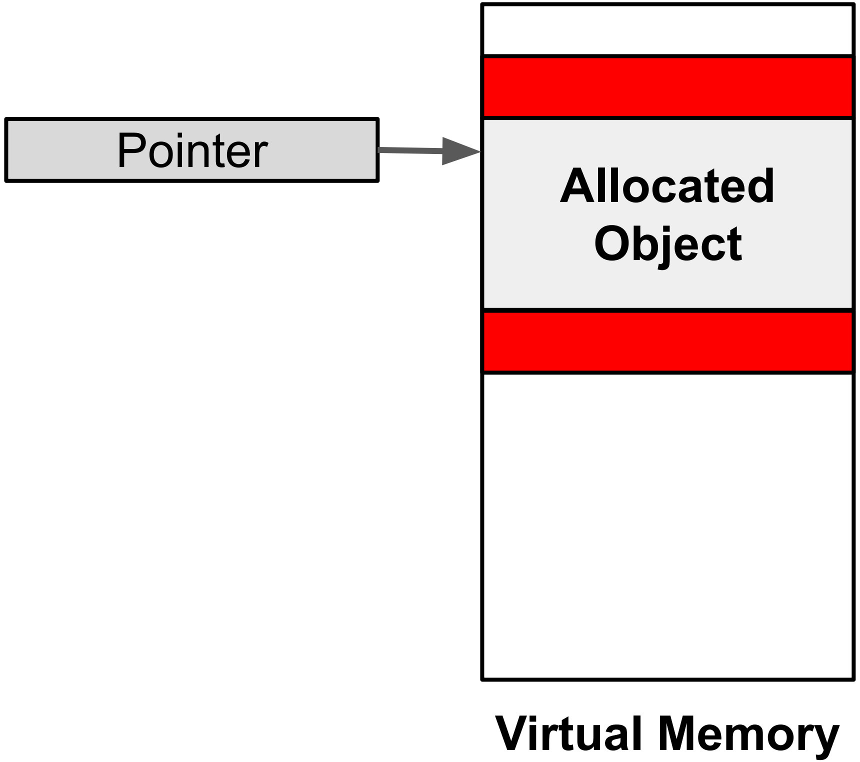
	Whitelisting		Blacklisting
	Per-object	**Per-pointer**	
Disjoint Metadata	Compatible C Baggy Bounds	Mondrian M-machine Softbound, Hardbound Watchdog CUP, MPX	Purify Valgrind Dr. Memory Electric Fence ASan
Inlined Metadata	EffectiveSan	(aka Fat Pointers) CHERI Cyclone CheckedC	SafeMem REST CALIFORMS
Co-joined Metadata	ARM Memory Tagging SPARC ADI		
No Metadata	Lowfat s/w	Lowfat h/w	

[27] Duck & Yap. EffectiveSan: type and memory error detection using dynamically typed C/C++. **[PLDI 2018]**

	Whitelisting		Blacklisting
	Per-object	**Per-pointer**	
Disjoint Metadata	Compatible C Baggy Bounds	Mondrian M-machine Softbound, Hardbound Watchdog CUP, MPX	Purify Valgrind Dr. Memory Electric Fence ASan
Inlined Metadata	EffectiveSan	(aka Fat Pointers) CHERI Cyclone CheckedC	SafeMem REST CALIFORMS
Co-joined Metadata	ARM Memory Tagging SPARC ADI		
No Metadata	Lowfat s/w	Lowfat h/w	

BLACKLISTING: DISJOINT METADATA

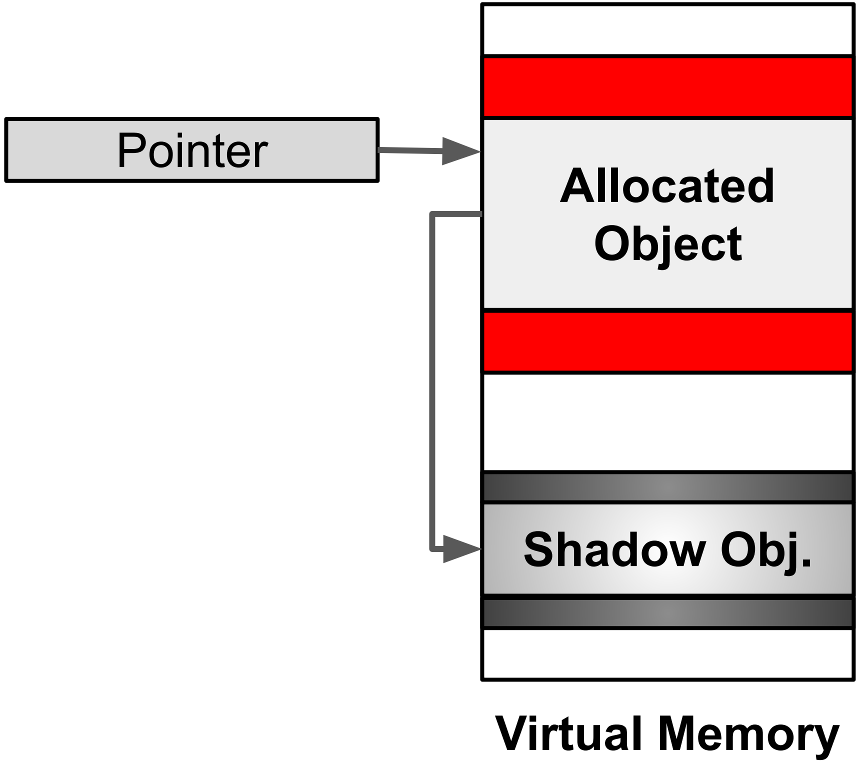
- Main Idea:
  - Guard objects with red zones.



[24] Serebryany, AddressSanitizer: a fast address sanity checker. **[USENIX ATC 2012]**

BLACKLISTING: DISJOINT METADATA

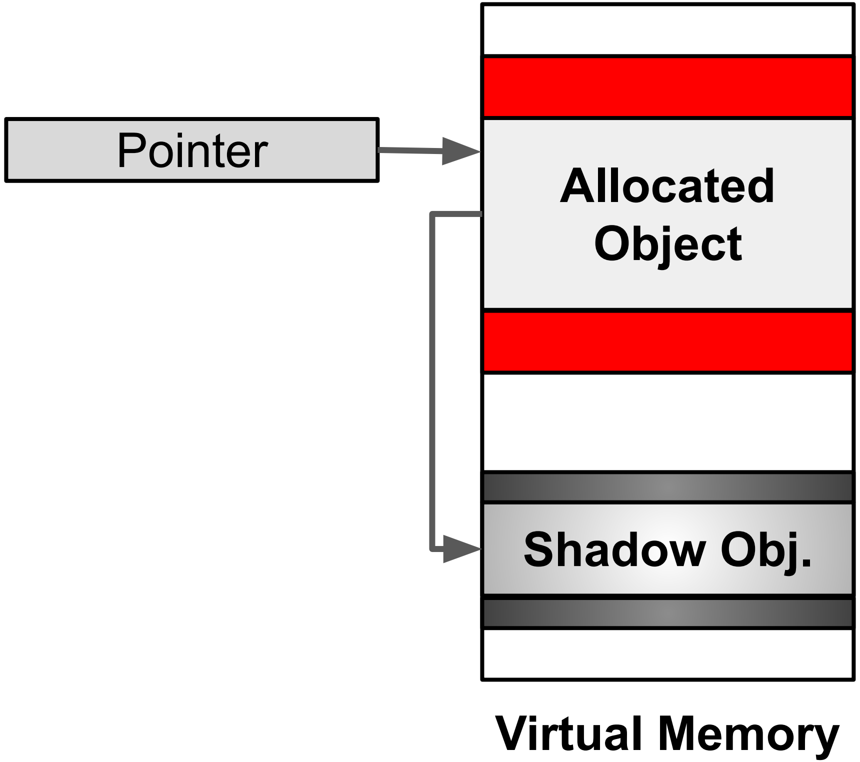
- Main Idea:
  - Guard objects with red zones.
  - Use shadow memory to identify red zone locations.



[24] Serebryany, AddressSanitizer: a fast address sanity checker. **[USENIX ATC 2012]**

BLACKLISTING: DISJOINT METADATA

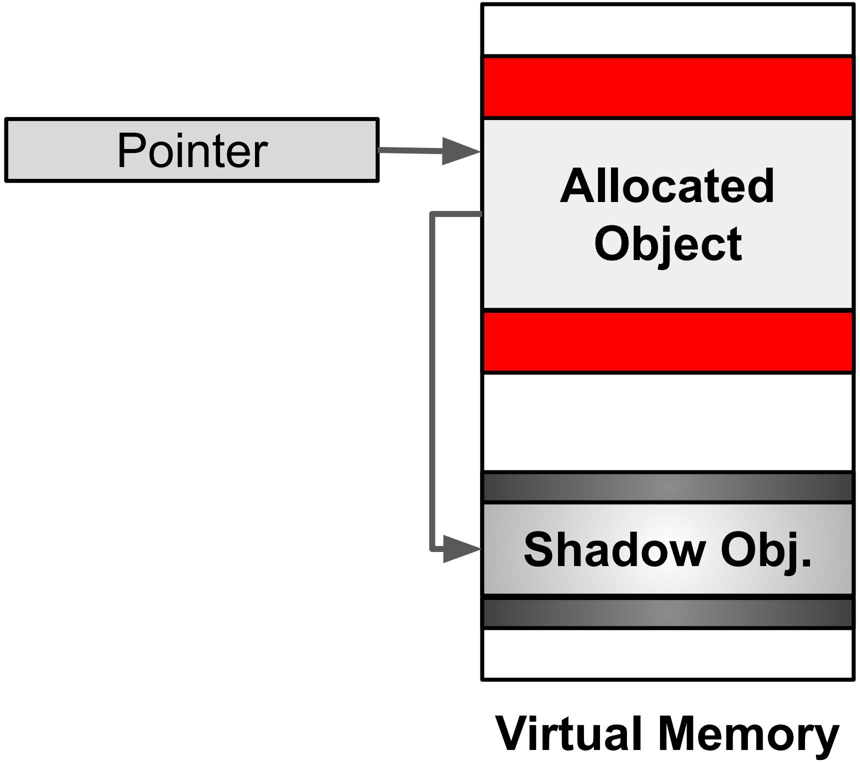
- Main Idea:
  - Guard objects with red zones.
  - Use shadow memory to identify red zone locations.
  - E.g., AddressSanitizer (ASan).



[24] Serebryany, AddressSanitizer: a fast address sanity checker. **[USENIX ATC 2012]**

BLACKLISTING: DISJOINT METADATA

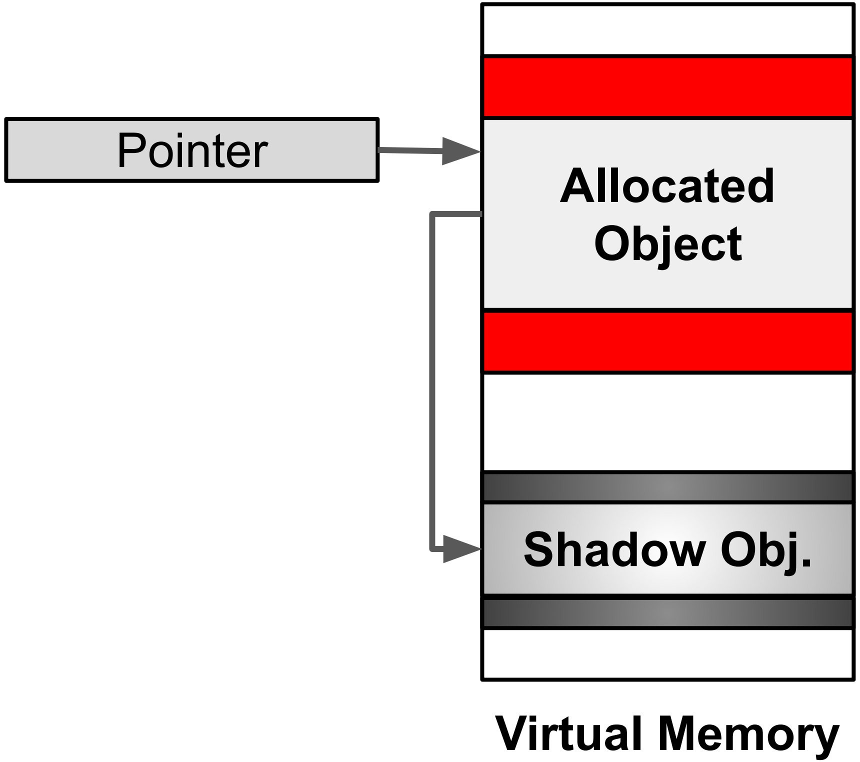
- Main Idea:
  - Guard objects with red zones.
  - Use shadow memory to identify red zone locations.
  - E.g., AddressSanitizer (ASan).
- Pros and Cons:
  - ✚ No metadata propagation.
  - ✚ Good binary compatibility.



[24] Serebryany, AddressSanitizer: a fast address sanity checker. **[USENIX ATC 2012]**

BLACKLISTING: DISJOINT METADATA

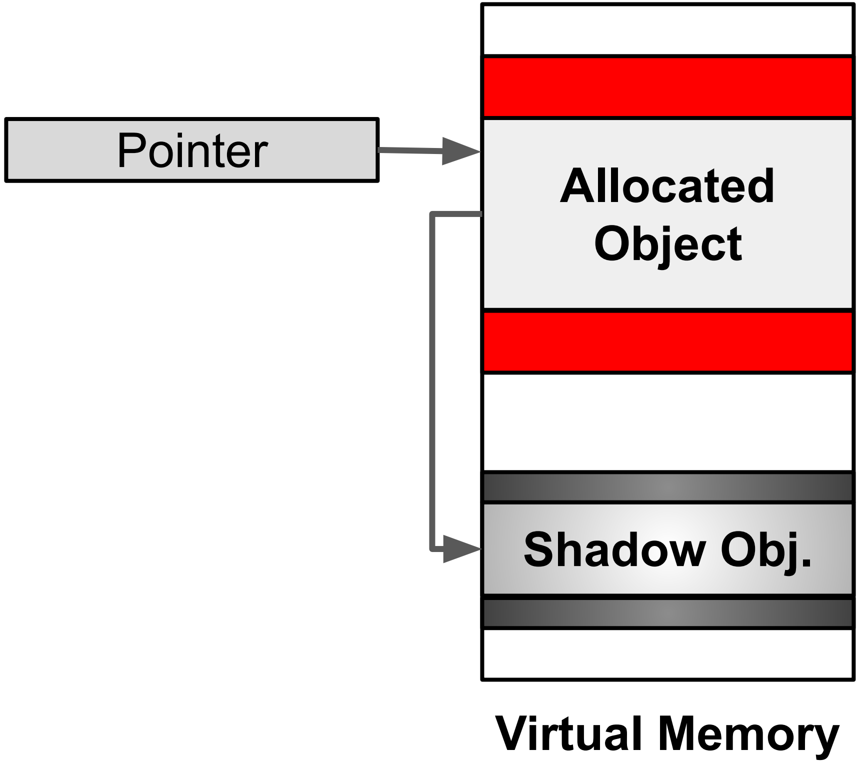
- Main Idea:
  - Guard objects with red zones.
  - Use shadow memory to identify red zone locations.
  - E.g., AddressSanitizer (ASan).
- Pros and Cons:
  - + No metadata propagation.
  - + Good binary compatibility.
  - - High memory footprint.



[24] Serebryany, AddressSanitizer: a fast address sanity checker. **[USENIX ATC 2012]**

BLACKLISTING: DISJOINT METADATA

- Main Idea:
  - Guard objects with red zones.
  - Use shadow memory to identify red zone locations.
  - E.g., AddressSanitizer (ASan).
- Pros and Cons:
  - + No metadata propagation.
  - + Good binary compatibility.
  - - High memory footprint.
  - - Less precise than whitelisting.



[24] Serebryany, AddressSanitizer: a fast address sanity checker. **[USENIX ATC 2012]**

	Whitelisting		Blacklisting
	Per-object	**Per-pointer**	
Disjoint Metadata	Compatible C Baggy Bounds	Mondrian M-machine Softbound, Hardbound Watchdog CUP, MPX	Purify Valgrind Dr. Memory Electric Fence ASan
Inlined Metadata	EffectiveSan	(aka Fat Pointers) CHERI Cyclone CheckedC	SafeMem REST CALIFORMS
Co-joined Metadata	ARM Memory Tagging SPARC ADI		
No Metadata	Lowfat s/w	Lowfat h/w	

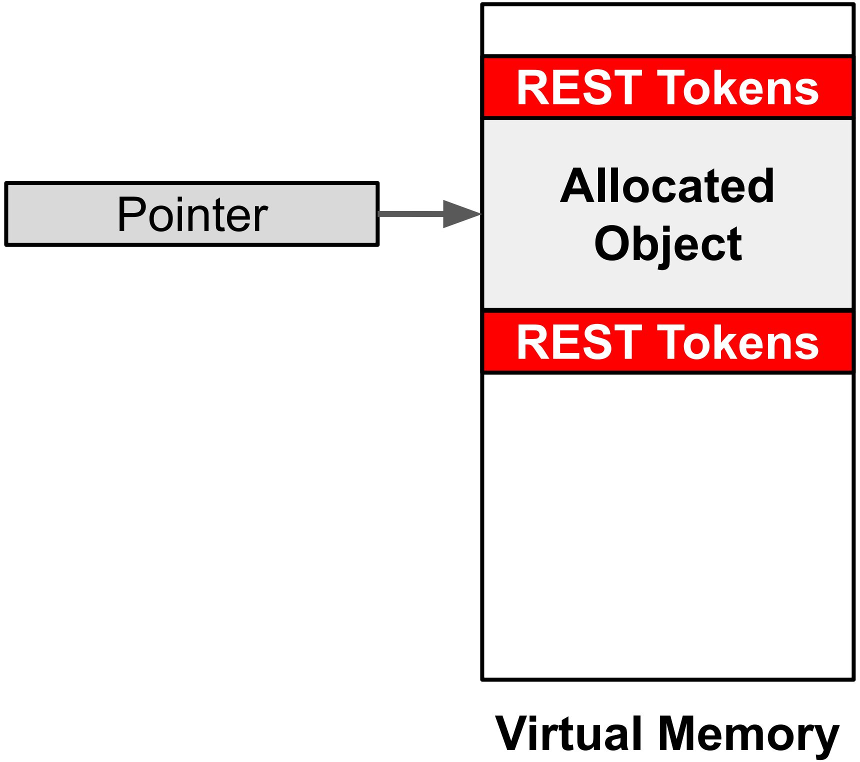
	Whitelisting		Blacklisting
	Per-object	**Per-pointer**	
Disjoint Metadata	Compatible C Baggy Bounds	Mondrian M-machine Softbound, Hardbound Watchdog CUP, MPX	Purify Valgrind Dr. Memory Electric Fence ASan
Inlined Metadata	EffectiveSan	(aka Fat Pointers) CHERI Cyclone CheckedC	**SafeMem** REST CALIFORMS
Co-joined Metadata	ARM Memory Tagging SPARC ADI		
No Metadata	Lowfat s/w	Lowfat h/w	

[29] Qin et. al., SafeMem: exploiting ECC-memory for detecting memory leaks. **[HPCA 2005]**

	Whitelisting		**Blacklisting**
	Per-object	**Per-pointer**	
Disjoint Metadata	Compatible C Baggy Bounds	Mondrian M-machine Softbound, Hardbound Watchdog CUP, MPX	Purify Valgrind Dr. Memory Electric Fence ASan
Inlined Metadata	EffectiveSan	(aka Fat Pointers) CHERI Cyclone CheckedC	SafeMem **REST** **CALIFORMS**
Co-joined Metadata	ARM Memory Tagging SPARC ADI		
No Metadata	Lowfat s/w	Lowfat h/w	

BLACKLISTING: INLINED METADATA (REST)

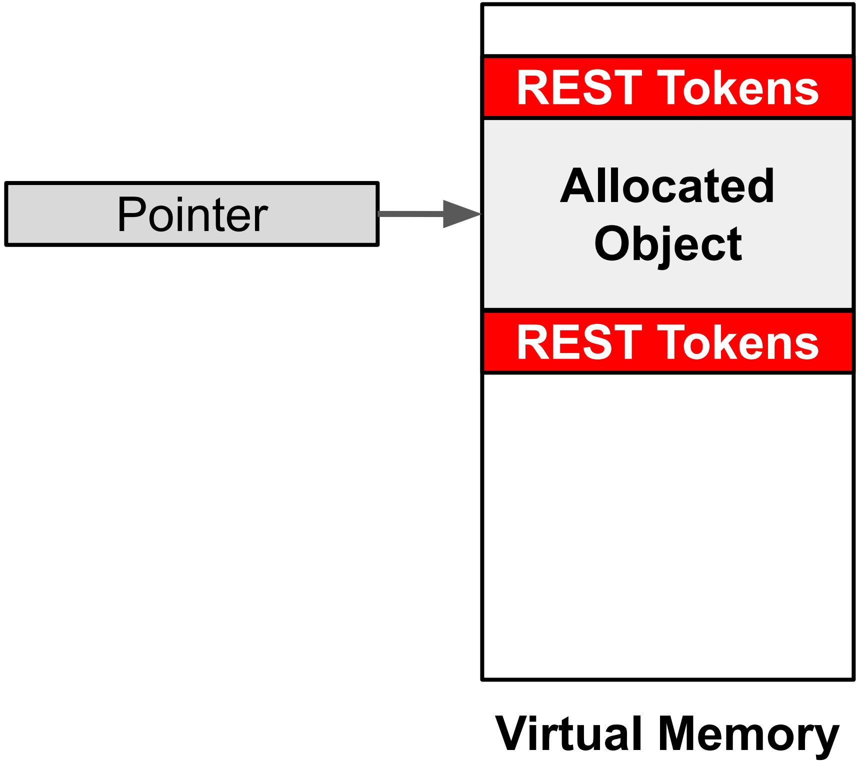
- Main Idea:
  - Store a unique value (Token) in locations to be blacklisted.
  - Issue an exception when a regular load/store touches them.



[31] Kanad Sinha and Simha Sethumadhavan. Practical memory safety with REST. **[ISCA 2018]**

BLACKLISTING: INLINED METADATA (**REST**)

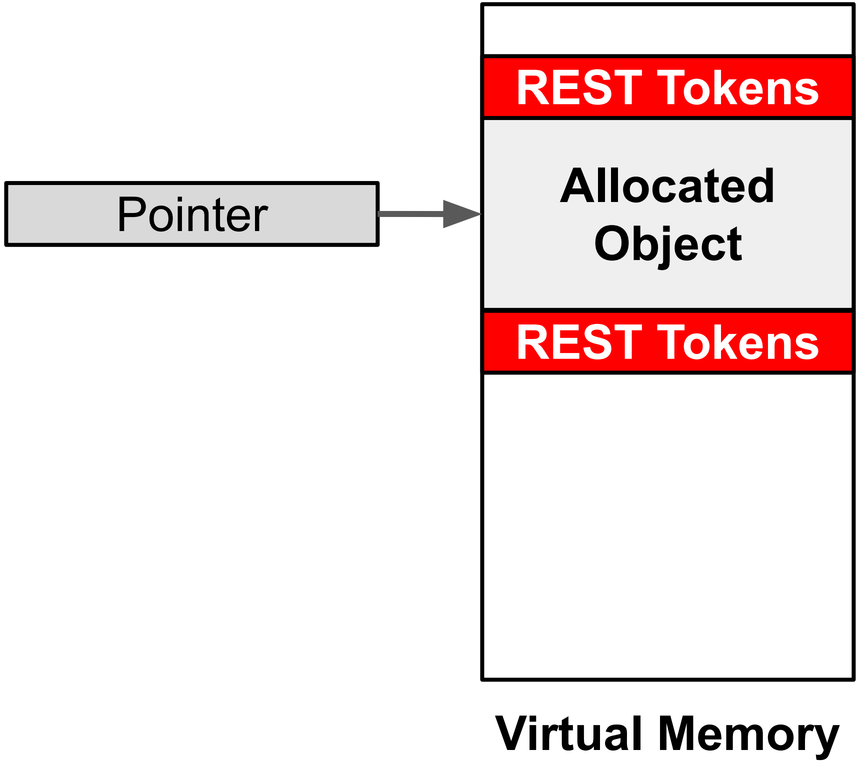
- Main Idea:
  - Store a unique value (Token) in locations to be blacklisted.
  - Issue an exception when a regular load/store touches them.
- Pros and Cons:
  - + Negligible perf. overheads.
  - + Good binary compatibility.



[31] Kanad Sinha and Simha Sethumadhavan. Practical memory safety with REST. **[ISCA 2018]**

BLACKLISTING: INLINED METADATA (**REST**)

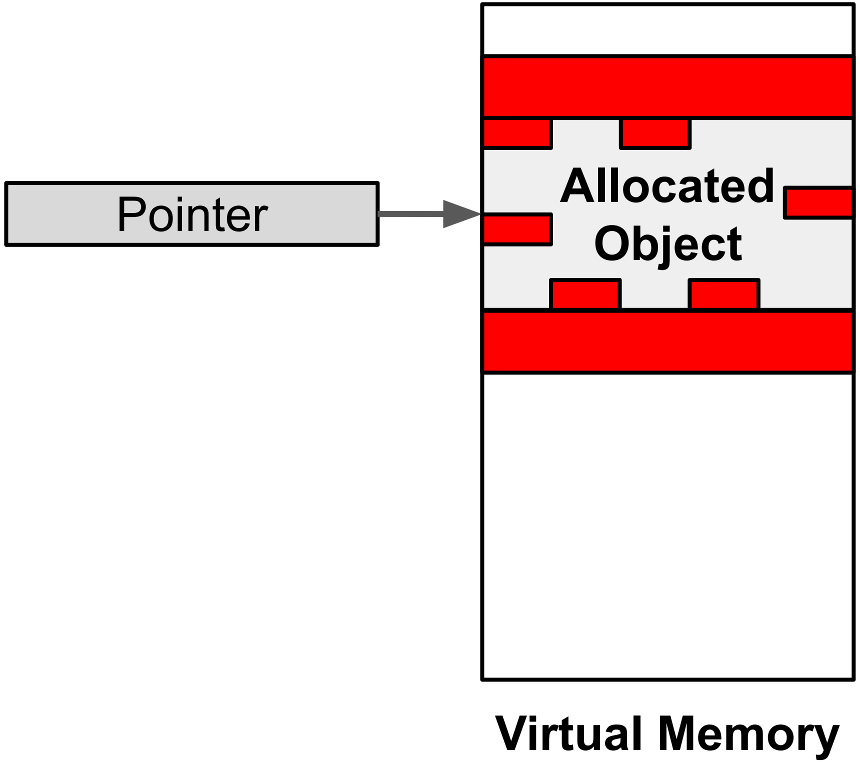
- Main Idea:
  - Store a unique value (Token) in locations to be blacklisted.
  - Issue an exception when a regular load/store touches them.
- Pros and Cons:
  - + Negligible perf. overheads.
  - + Good binary compatibility.
  - - No intra-object protection.



[31] Kanad Sinha and Simha Sethumadhavan. Practical memory safety with REST. **[ISCA 2018]**

BLACKLISTING: INLINED METADATA (CALIFORMS)

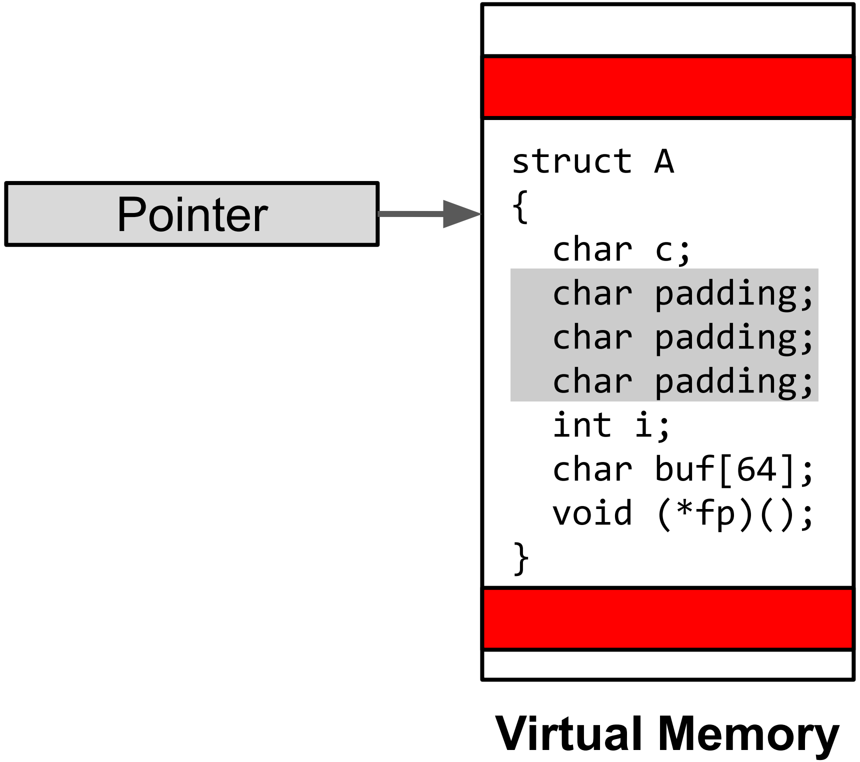
- Main Idea:
  - Use natural padding bytes in structs to store the metadata.



[1] Sasaki et. al., Practical byte-granular memory blacklisting using Califorms. **[MICRO 2019]**

BLACKLISTING: INLINED METADATA (**CALIFORMS**)

- Main Idea:
  - Use natural padding bytes in structs to store the metadata.



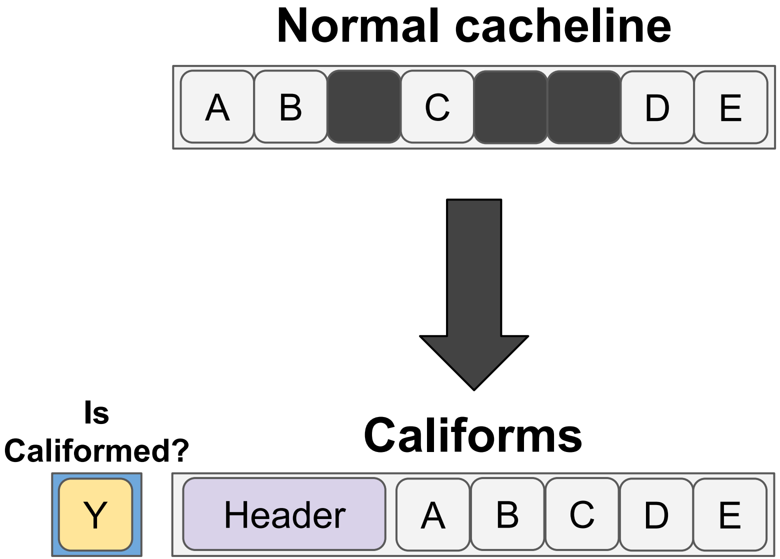
```
struct A
{
  char c;
  char padding;
  char padding;
  char padding;
  int i;
  char buf[64];
  void (*fp)();
}
```

Virtual Memory

[1] Sasaki et. al., Practical byte-granular memory blacklisting using Califorms. **[MICRO 2019]**

BLACKLISTING: INLINED METADATA (**CALIFORMS**)

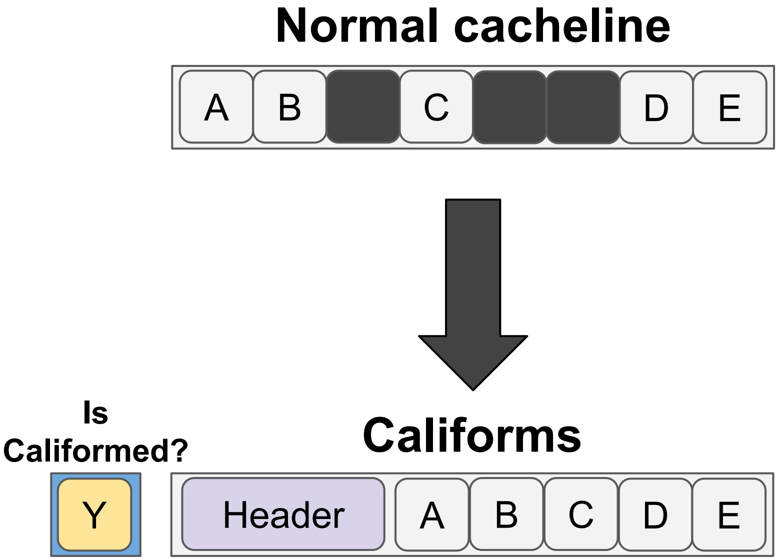
- Main Idea:
  - Use natural padding bytes in structs to store the metadata.
  - Only 1-bit of metadata is needed per each 64B cacheline.



[1] Sasaki et. al., Practical byte-granular memory blacklisting using Califorms. **[MICRO 2019]**

BLACKLISTING: INLINED METADATA (**CALIFORMS**)

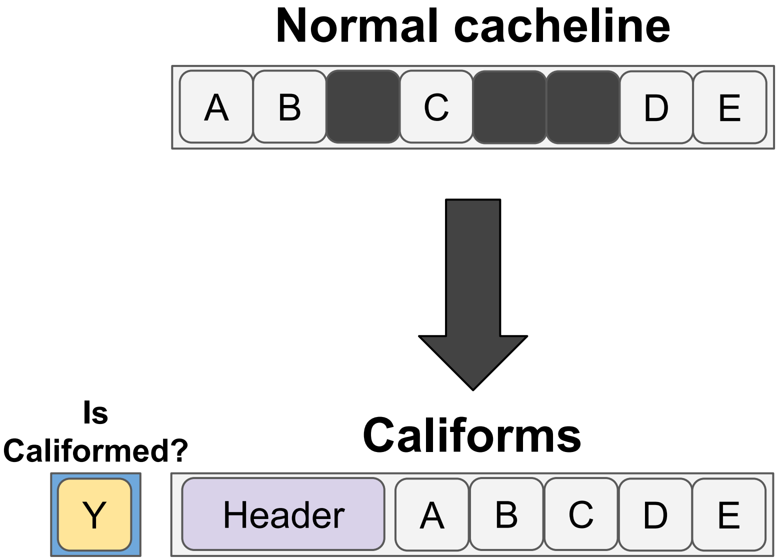
- Main Idea:
  - Use natural padding bytes in structs to store the metadata.
  - Only 1-bit of metadata is needed per each 64B cacheline.
- Pros and Cons:
  - ✚ Intra-object protection.
  - ✚ Negligible perf. overheads.



[1] **[1]** Sasaki et. al., Practical byte-granular memory blacklisting using Califorms. **[MICRO 2019]**

BLACKLISTING: INLINED METADATA (CALIFORMS)

- Main Idea:
  - Use natural padding bytes in structs to store the metadata.
  - Only 1-bit of metadata is needed per each 64B cacheline.
- Pros and Cons:
  - + Intra-object protection.
  - + Negligible perf. overheads.
  - - Same paddings layout for objects of the same type.



[1] Sasaki et. al., Practical byte-granular memory blacklisting using Califorms. **[MICRO 2019]**

	Whitelisting		Blacklisting	Randomized Allocators
	Per-object	Per-pointer		
Disjoint Metadata	Compatible C Baggy Bounds	Mondrian M-machine Softbound Hardbound Watchdog CUP, MPX	Purify Valgrind Dr. Memory Electric Fence ASan	Diehard FreeGuard Guarder
Inlined Metadata	EffectiveSan	(aka Fat Pointers) CHERI Cyclone CheckedC	SafeMem REST CALIFORMS	
Co-joined Metadata	ARM Memory Tagging SPARC ADI			
No Metadata	Lowfat s/w	Lowfat h/w		

[20] Berger and Zorn, DieHard: Probabilistic memory safety for unsafe languages. **[PLDI 2006]**

MEMORY SAFETY TECHNIQUES

Spatial Memory Safety

Temporal Memory Safety

Naive Solution	Never use `Free()`

Naive Solution	Never use `Free()`

High memory consumption

& memory leaks!

Naive Solution		Never use `Free()`
Garbage Collection (GC)	**Regular**	Hardware Accelerated GC
	Conservative	MarkUs

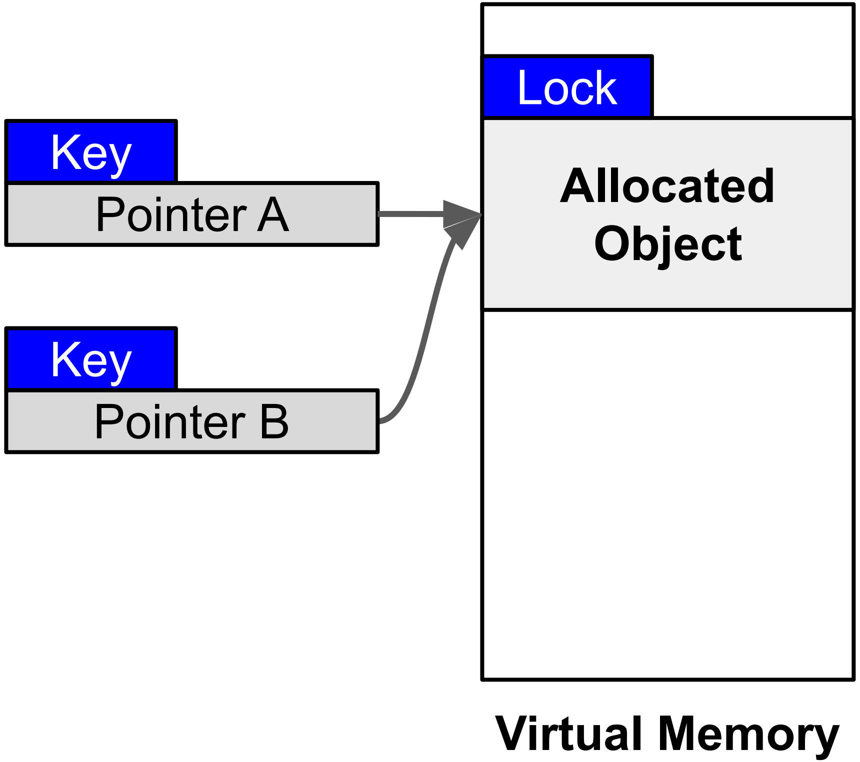
[37] Mass et. al., A hardware accelerator for tracing garbage collection. **[ISCA 2018]**

Naive Solution		Never use `Free()`
Garbage Collection (GC)	**Regular**	Hardware Accelerated GC
	Conservative	MarkUs
Memory Quarantining		Valgrind, ASan, REST, Califorms, CHERIvoke

Naive Solution			Never use `Free()`
Garbage Collection (GC)	**Regular**		Hardware Accelerated GC
	Conservative		MarkUs
Memory Quarantining			Valgrind, ASan, REST, Califorms, CHERIvoke
Lock & Key	**Explicit**	**Change Lock**	CETS, CUP
	Implicit	**Change Lock**	Electric Fence, Oscar
		Revoke key	DangNull, DangSan, BOGO

LOCK & KEY: EXPLICIT LOCK CHANGE (**CETS**)

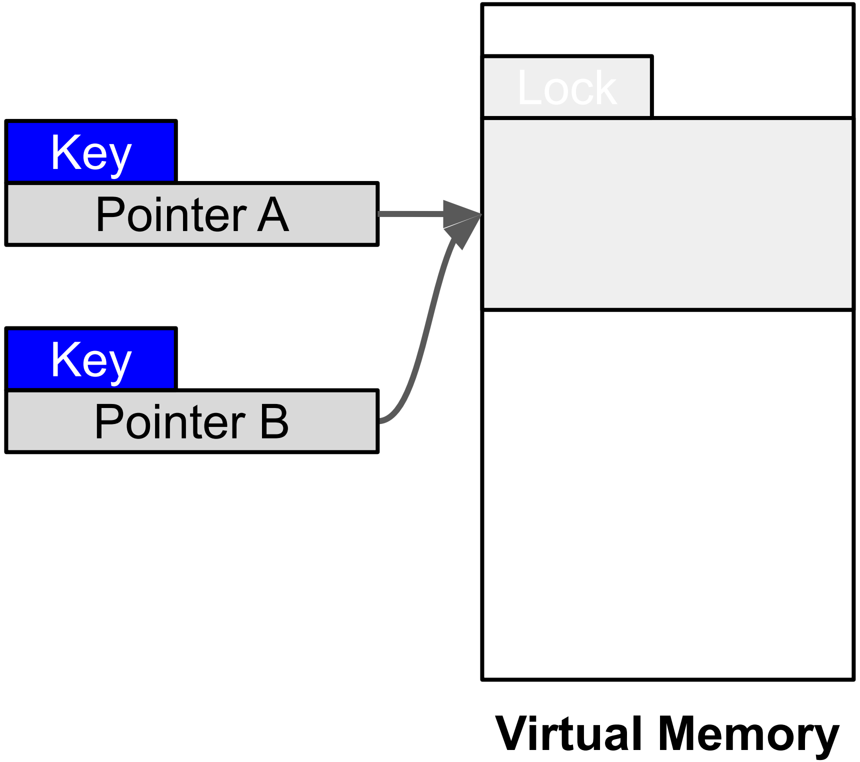
- Main Idea:
  - Use unique Lock per object.
  - Pass Lock to pointers as a key.
  - Propagate keys on pointer **arithmetic**.
  - Check on pointer **dereference**.



[17] Santosh Nagarakatte. Practical Low-Overhead Enforcement of Memory Safety. **[PhD Thesis 2012]**

LOCK & KEY: EXPLICIT LOCK CHANGE (**CETS**)

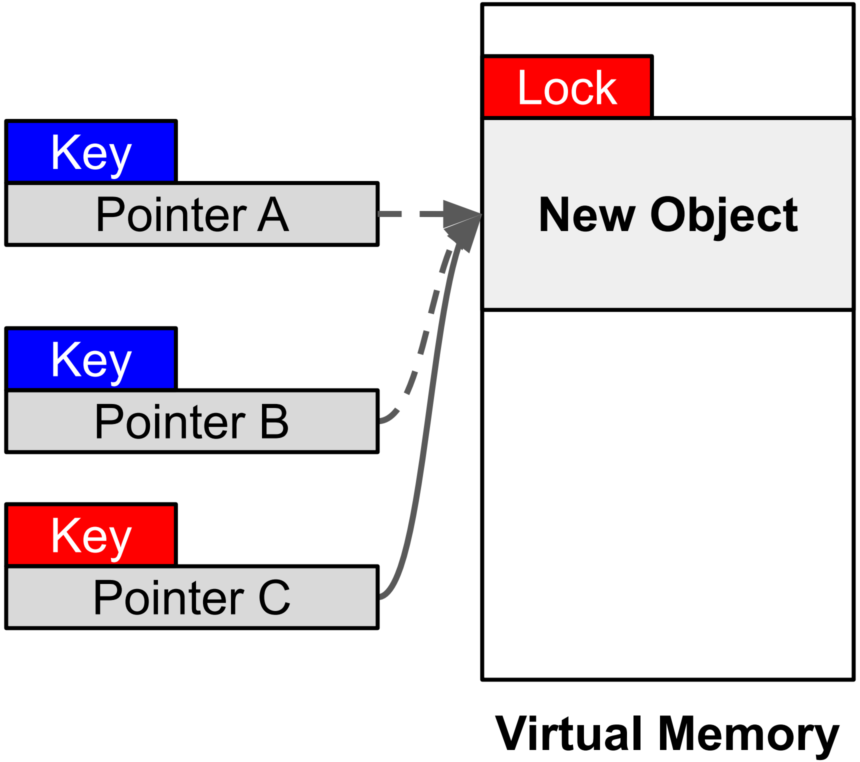
- Main Idea:
  - Use unique Lock per object.
  - Pass Lock to pointers as a key.
  - Propagate keys on pointer **arithmetic**.
  - Check on pointer **dereference**.



[17] Santosh Nagarakatte. Practical Low-Overhead Enforcement of Memory Safety. **[PhD Thesis 2012]**

LOCK & KEY: EXPLICIT LOCK CHANGE (CETS)

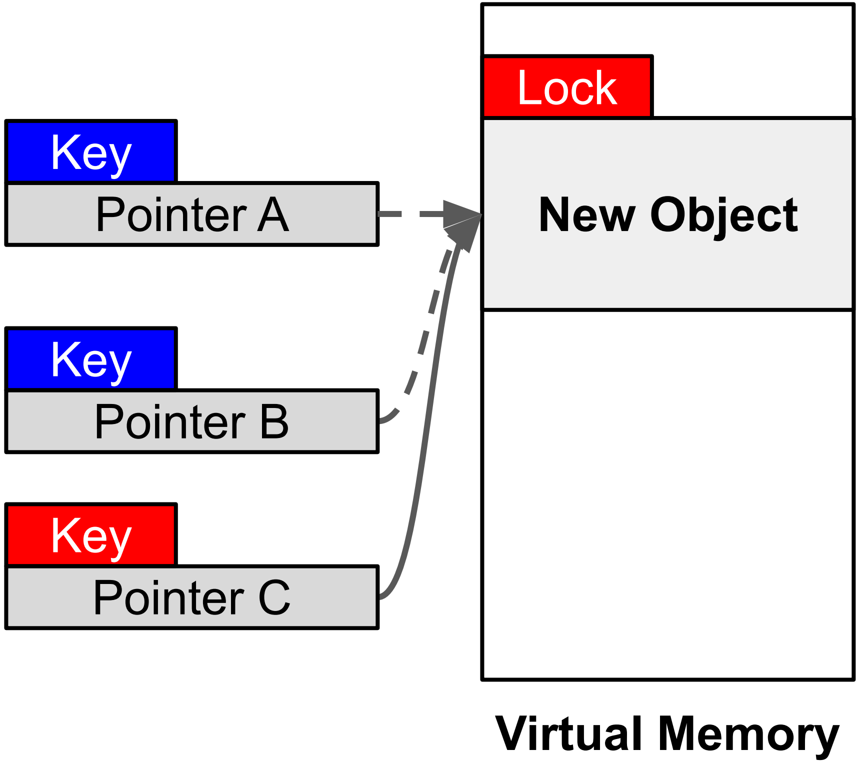
- Main Idea:
  - Use unique Lock per object.
  - Pass Lock to pointers as a key.
  - Propagate keys on pointer **arithmetic**.
  - Check on pointer **dereference**.



[17] Santosh Nagarakatte. Practical Low-Overhead Enforcement of Memory Safety. **[PhD Thesis 2012]**

LOCK & KEY: EXPLICIT LOCK CHANGE (**CETS**)

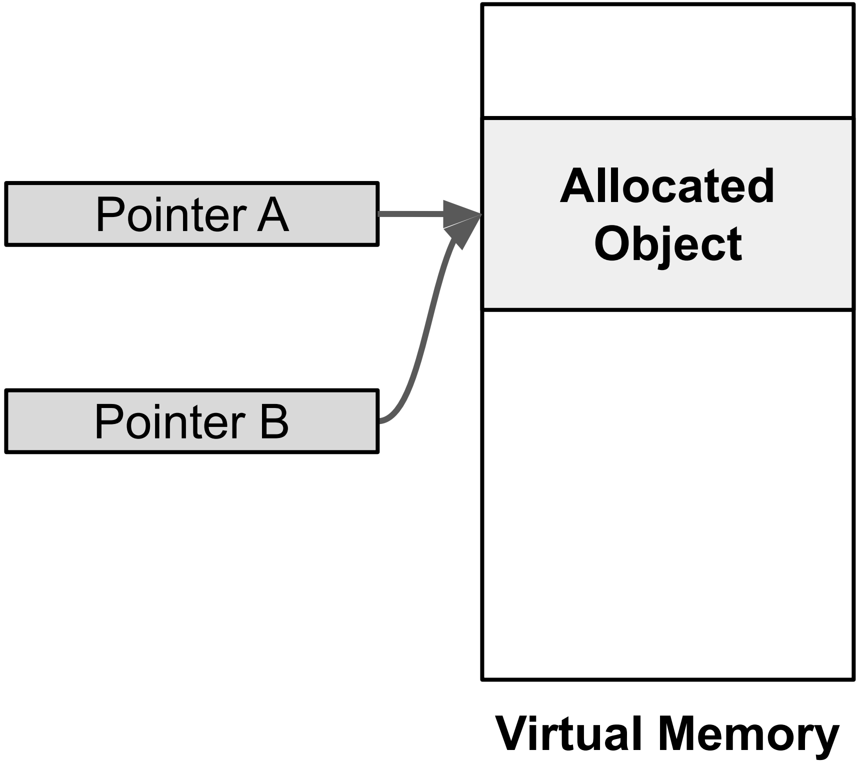
- Main Idea:
  - Use unique Lock per object.
  - Pass Lock to pointers as a key.
  - Propagate keys on pointer **arithmetic**.
  - Check on pointer **dereference**.
- Pros and Cons:
  - + Simple bounds checking.
  - - High performance overheads.



[17] Santosh Nagarakatte. Practical Low-Overhead Enforcement of Memory Safety. **[PhD Thesis 2012]**

LOCK & KEY: IMPLICIT LOCK CHANGE

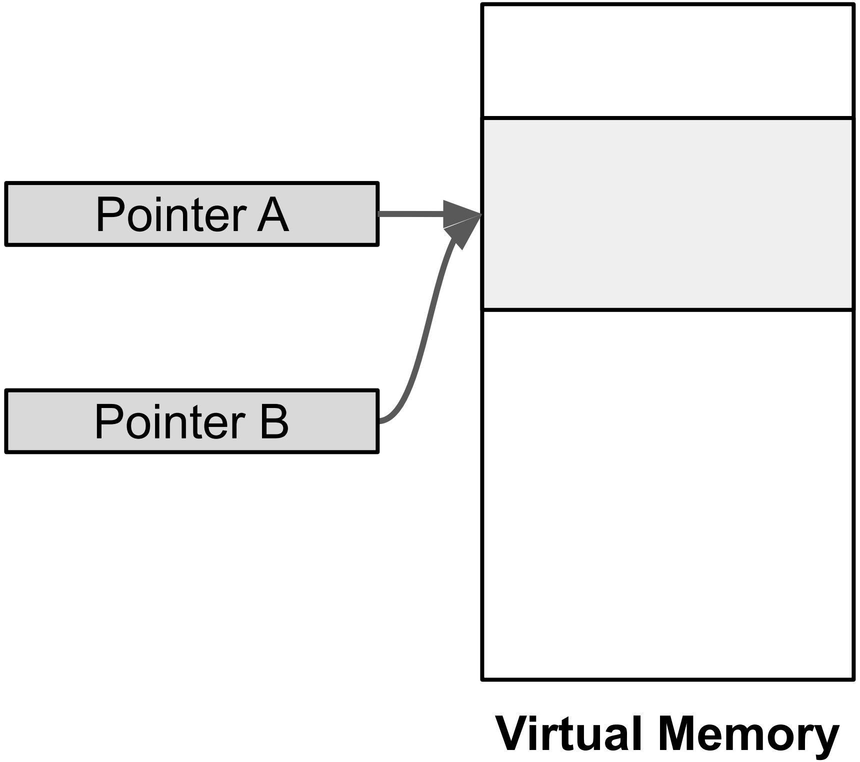
- Main Idea:
  - ○ Use object address as a lock.



[39] Dang et. al., Oscar: A practical page-permissions-based scheme. **[USENIX 2017]**

LOCK & KEY: IMPLICIT LOCK CHANGE

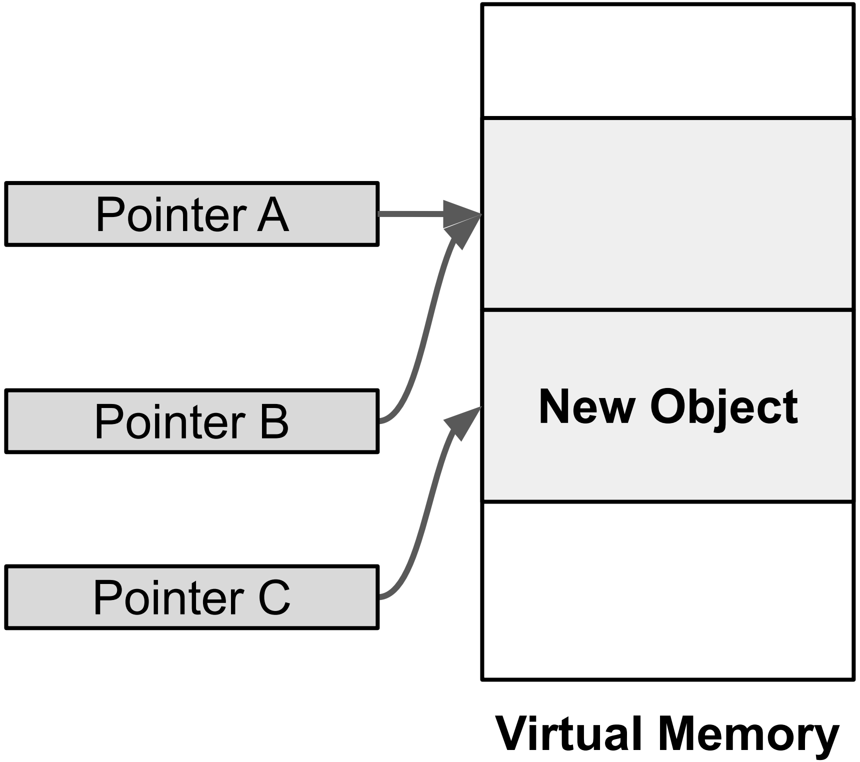
- Main Idea:
  - Use object address as a lock.
  - Mark virtual addresses as inaccessible upon free.



[39] Dang et. al., Oscar: A practical page-permissions-based scheme. **[USENIX 2017]**

LOCK & KEY: IMPLICIT LOCK CHANGE

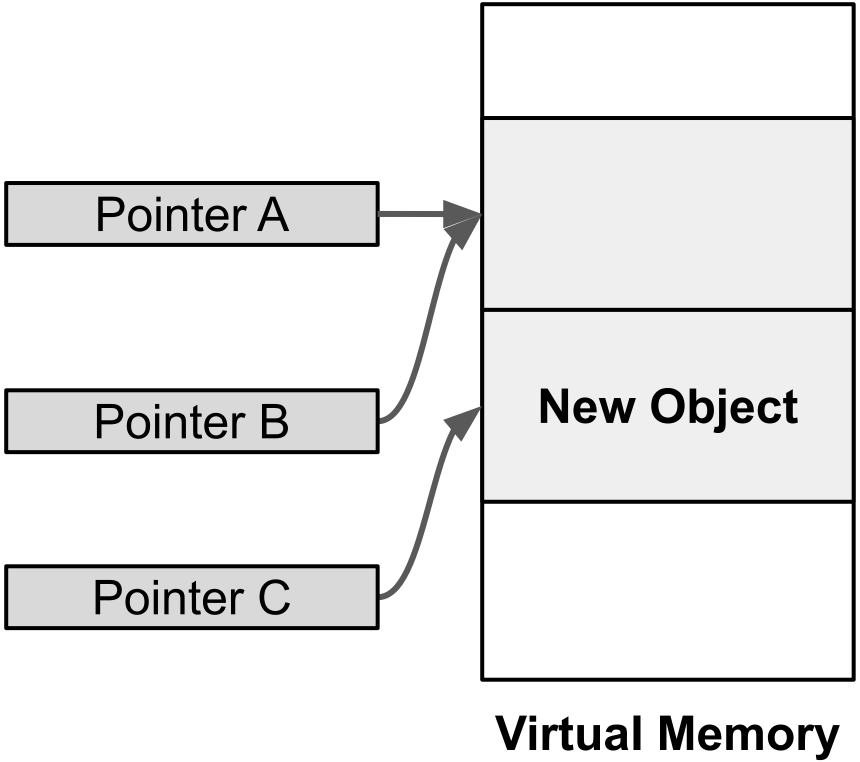
- Main Idea:
  - Use object address as a lock.
  - Mark virtual addresses as inaccessible upon free.
  - Never reuse virtual addresses.



[39] **Dang et. al., Oscar: A practical page-permissions-based scheme. [USENIX 2017]**

LOCK & KEY: IMPLICIT LOCK CHANGE

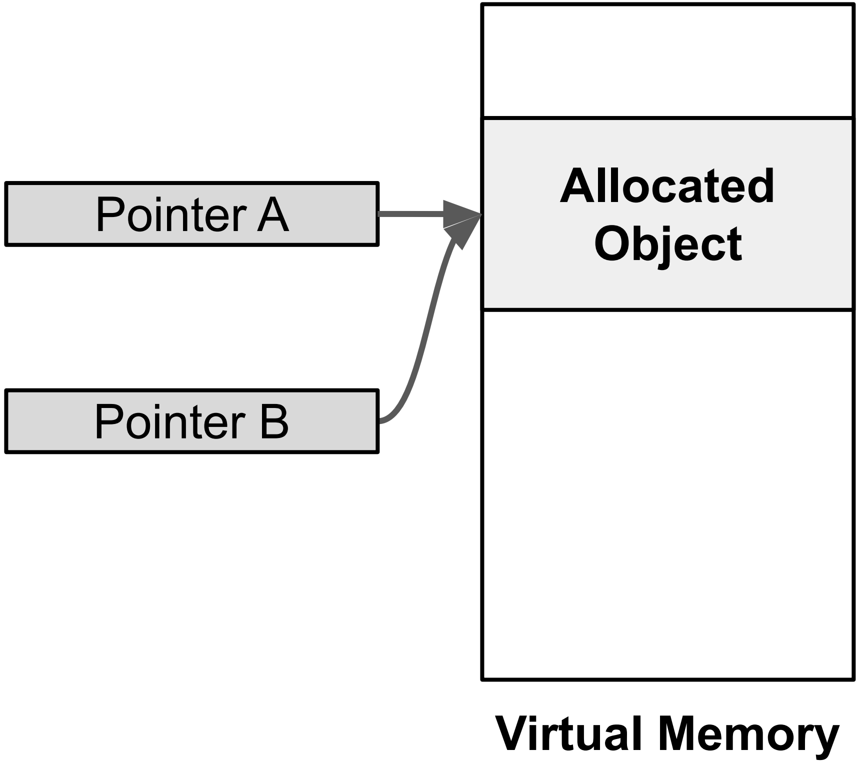
- Main Idea:
  - Use object address as a lock.
  - Mark virtual addresses as inaccessible upon free.
  - Never reuse virtual addresses.
- Pros and Cons:
  - + No per-pointer overheads.
  - - High TLB overheads.



[39] Dang et. al., Oscar: A practical page-permissions-based scheme. **[USENIX 2017]**

LOCK & KEY: IMPLICIT KEY REVOCATION

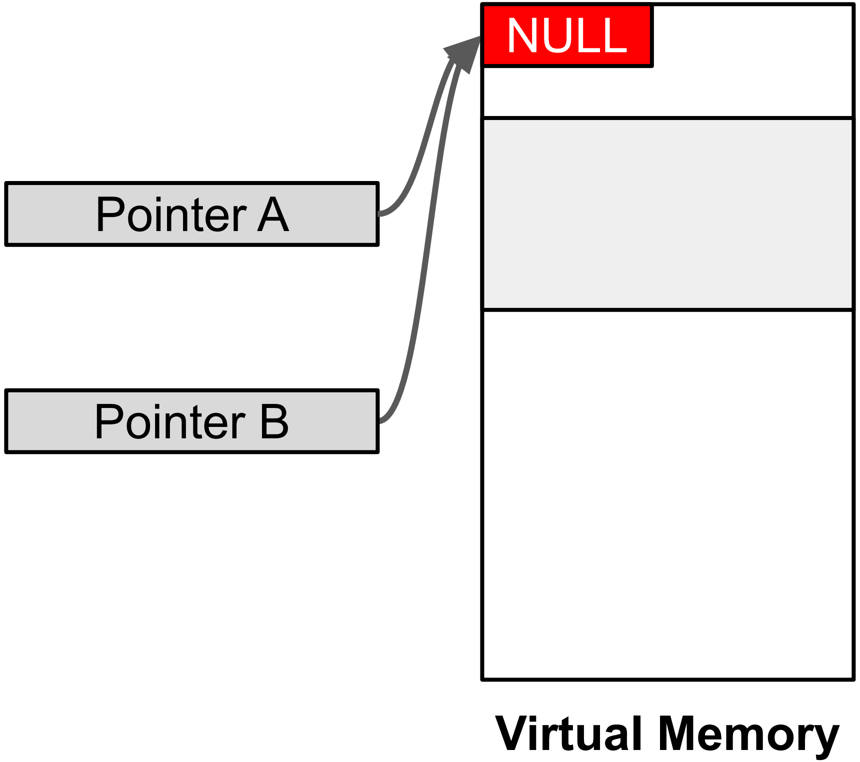
- Main Idea:
  - Use pointer value as a key.



[5] Song et. al., SoK: Sanitizing for security. **[S&P 2019]**

LOCK & KEY: IMPLICIT KEY REVOCATION

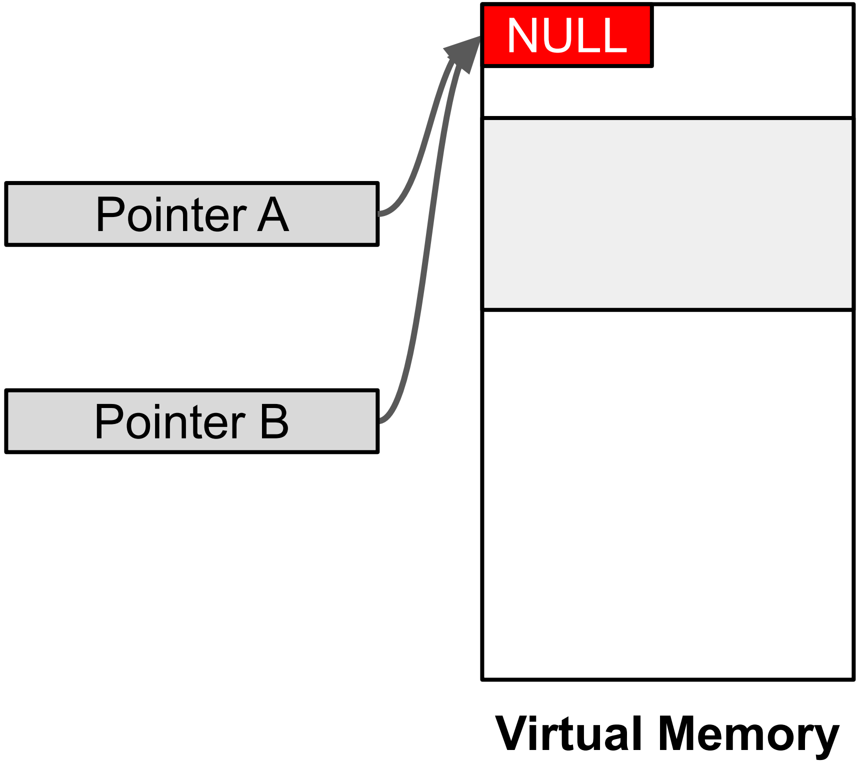
- Main Idea:
  - Use pointer value as a key.
  - Nullify pointers upon `Free()`.



[5] Song et. al., SoK: Sanitizing for security. **[S&P 2019]**

LOCK & KEY: IMPLICIT KEY REVOCATION

- Main Idea:
  - Use pointer value as a key.
  - Nullify pointers upon `Free()`.
- Limitations:
  - Overheads are proportional to the number of pointers.
  - May miss dangling pointers stored in registers.

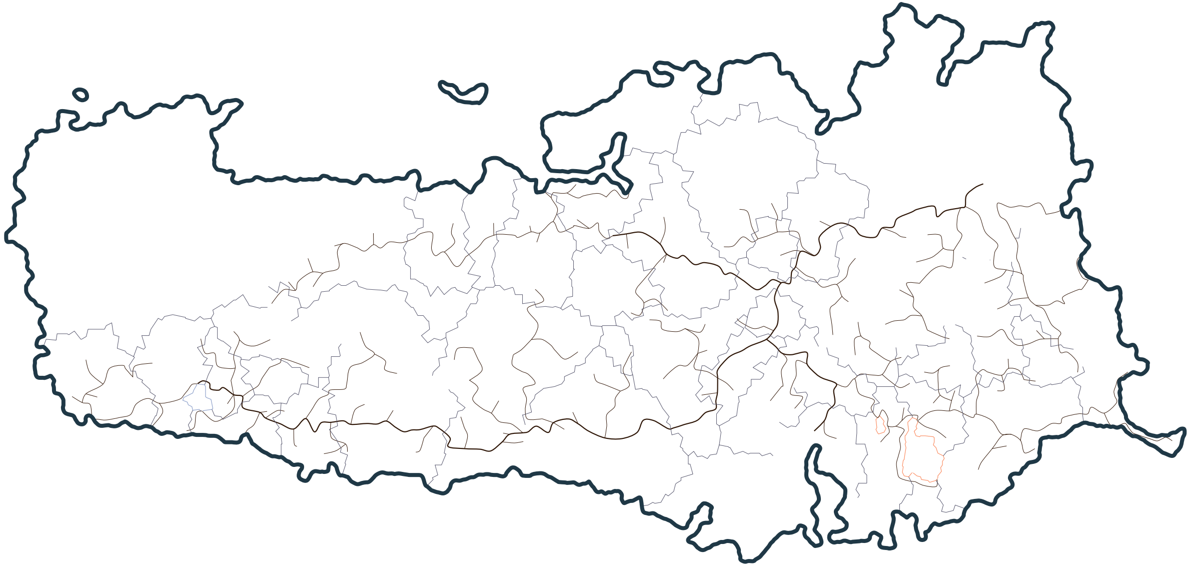


[5] Song et. al., SoK: Sanitizing for security. **[S&P 2019]**

Naive Solution			Never use `Free()`
Garbage Collection (GC)	**Regular**		Hardware Accelerated GC
	Conservative		MarkUs
Memory Quarantining			Valgrind, ASan, REST, Califorms, CHERIvoke
Lock & Key	**Explicit**	**Change Lock**	CETS, CUP
	Implicit	**Change Lock**	Electric Fence, Oscar
		Revoke key	DangNull, DangSan, BOGO

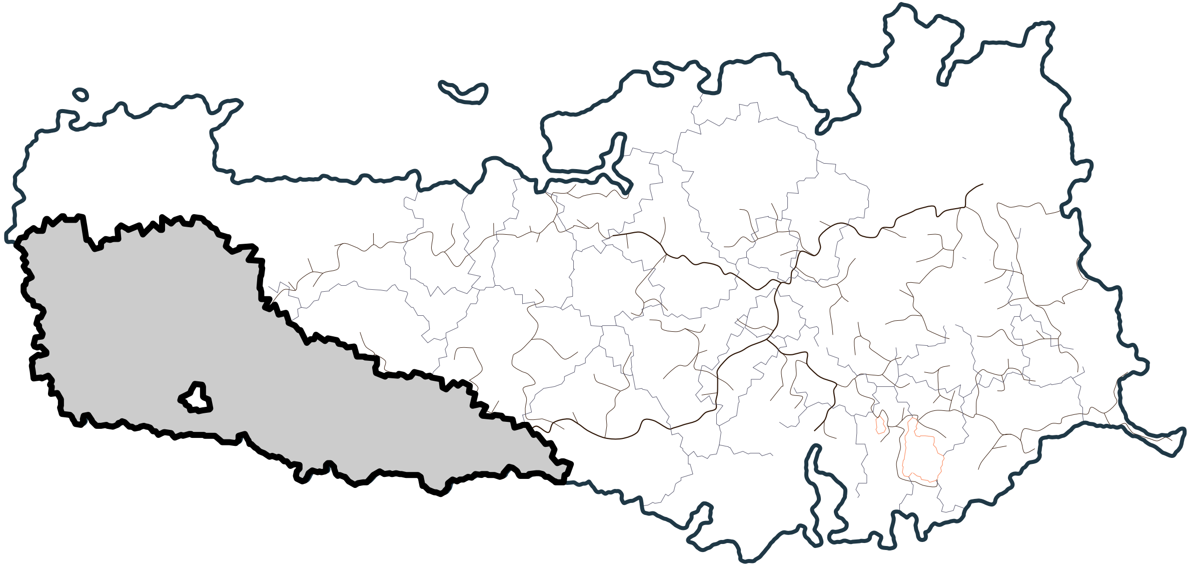
Future Work Map

FUTURE WORK MAP



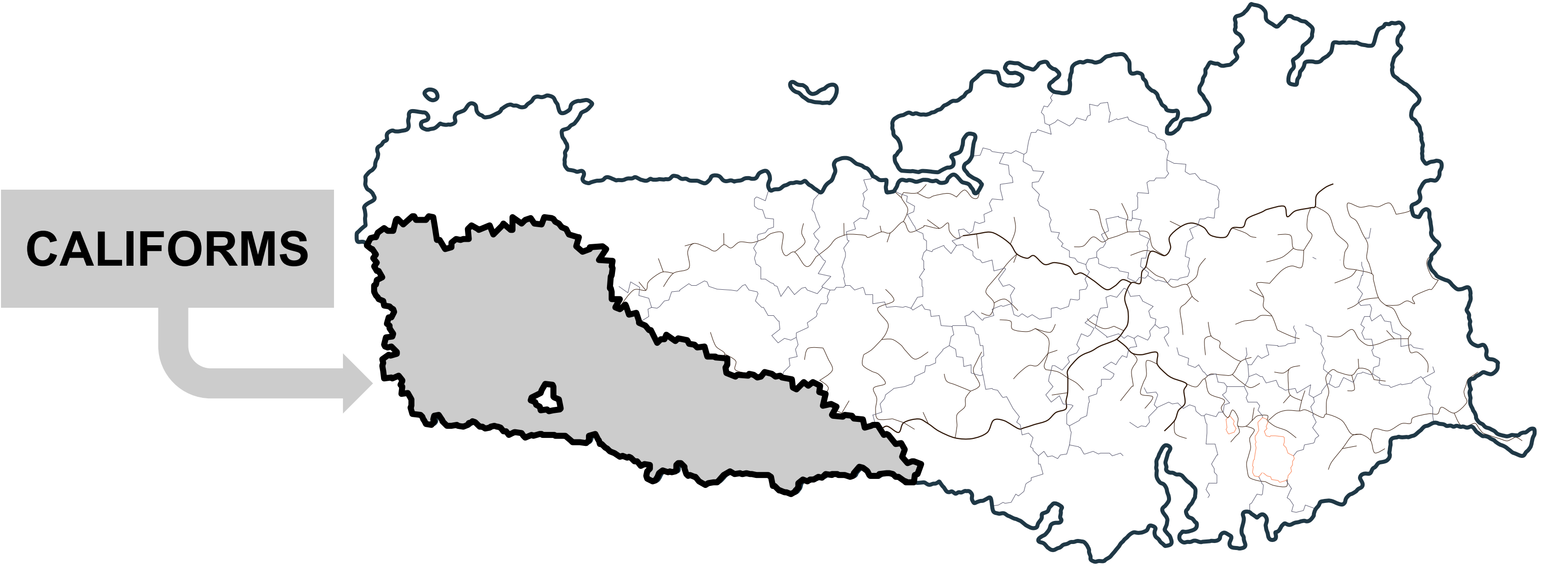
Created via: https://azgaar.github.io/Fantasy-Map-Generator/

LOW-COST MEMORY SAFETY SOLUTIONS



Created via: https://azgaar.github.io/Fantasy-Map-Generator/

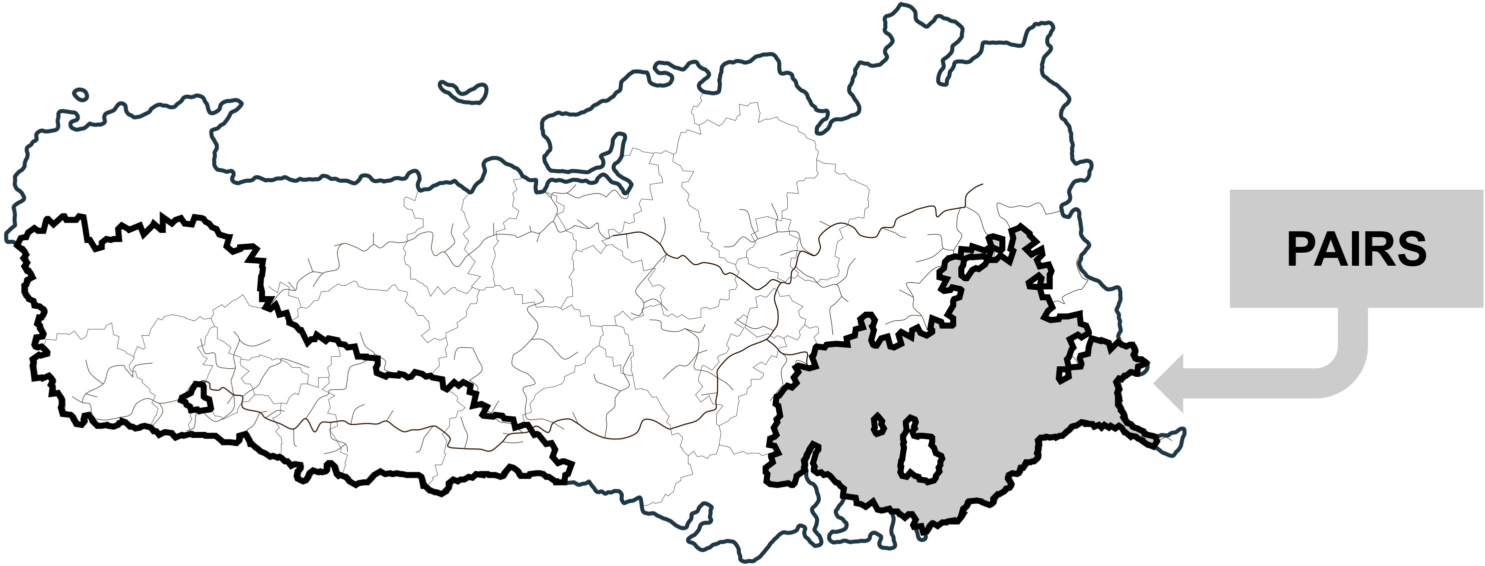
LOW-COST MEMORY SAFETY SOLUTIONS



Hiroshi Sasaki, Miguel A. Arroyo, **Mohamed Tarek Ibn Ziad**, Koustubha Bhat, Kanad Sinha, and Simha Sethumadhavan, Practical byte-granular memory blacklisting using Califorms. **[MICRO 2019] [IEEE Micro Top Picks Honorable Mention]**

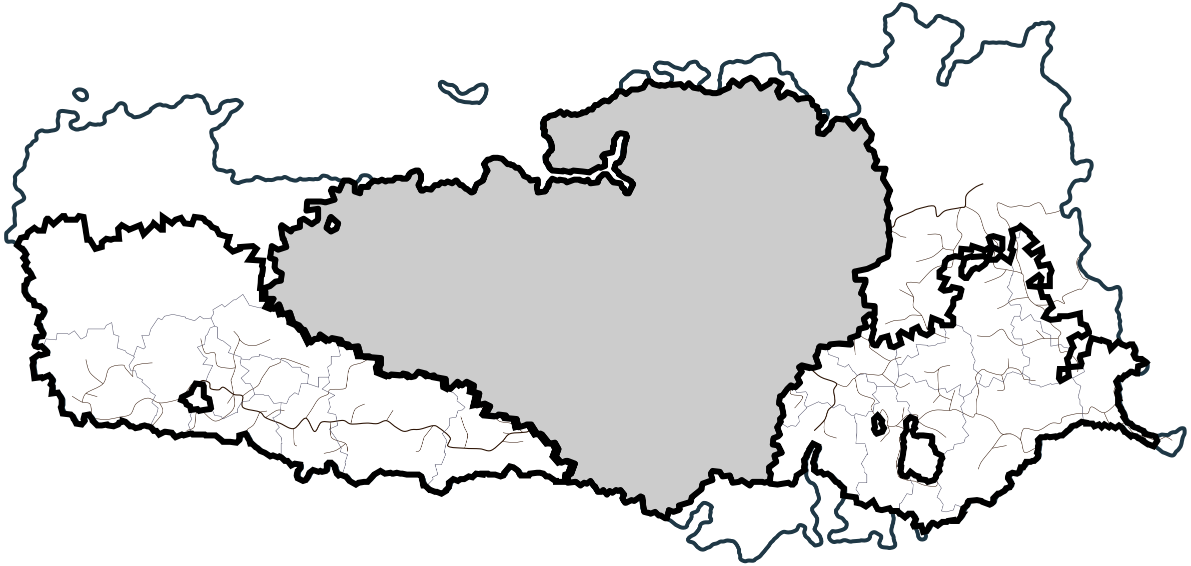
PROTECTING NON-64 BIT SYSTEMS

PROTECTING NON-64 BIT SYSTEMS

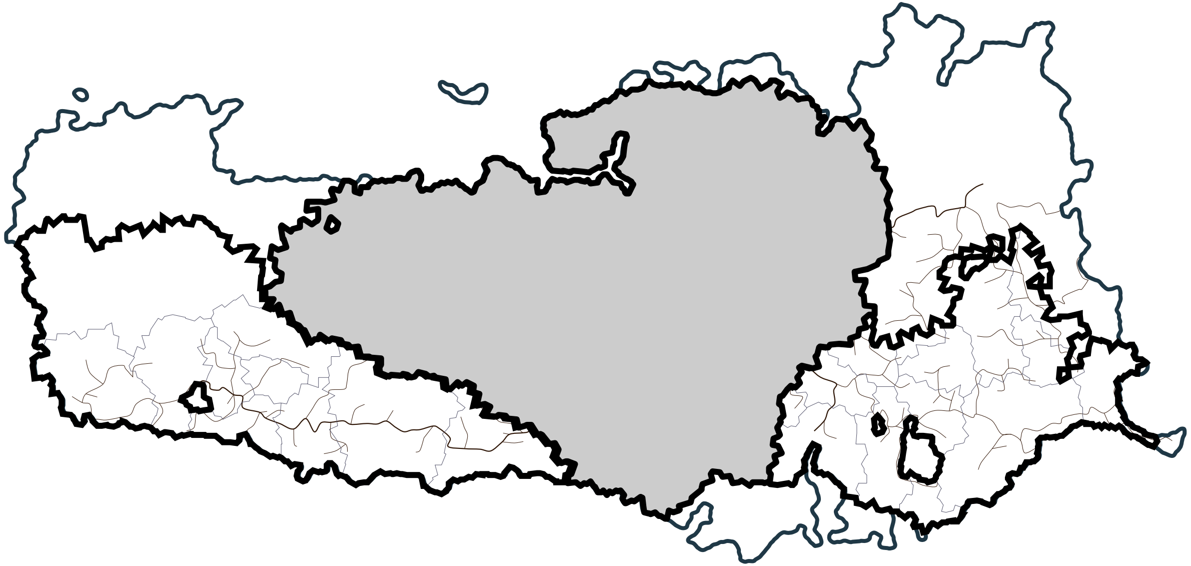


Mohamed Tarek Ibn Ziad and Evgeny Manzhosov, Practical Software Security on Heterogeneous Systems on Chips. **[Qualcomm Innovation Fellowship Finalists 2020]**

COHESIVE MEMORY SAFETY SOLUTIONS



COHESIVE MEMORY SAFETY SOLUTIONS



Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, and Simha Sethumadhavan,
SPAM: Stateless Permutation of Application Memory. [**Submitted to USENIX 2020**]

Why is memory safety still a concern?

Mohamed (Tarek Ibn Ziad) Hassan

Ph.D. Candidacy Exam
April 9th, 2020.

Why is memory safety still a concern?

Mohamed (Tarek Ibn Ziad) Hassan

QUESTIONS?

https://www.cs.columbia.edu/~mtarek/

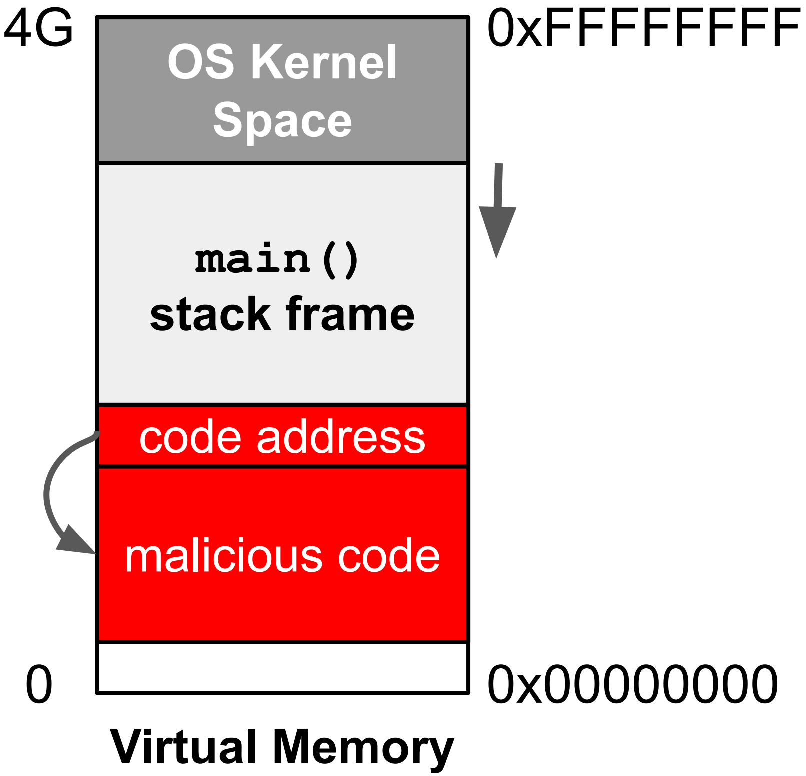
@M_TarekIbnZiad

Slide Intentionally Left Blank

Supplementary Slides

😈 CODE INJECTION

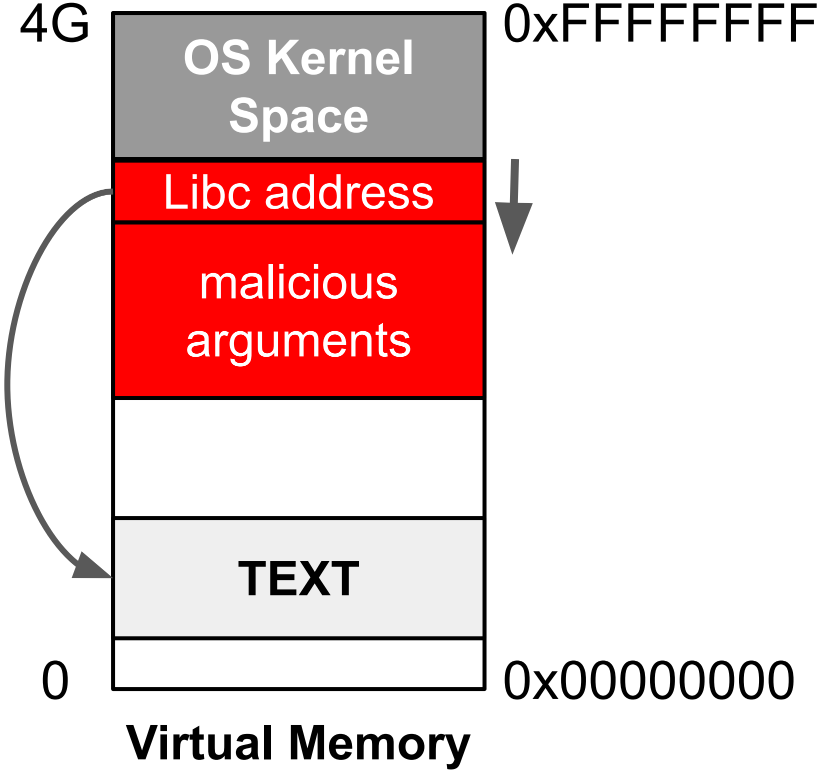
```
void main (int argc, char **argv) {
   ...
   vulnerable(argv[1]);
   ...
}
void vulnerable(char *str1){
   char str2[100];
   strcpy(str2,str1);
   return;
}
```



[4] Szekeres et. al, SoK: Eternal war in memory. **[S&P 2013]**

😈 RETURN-TO-LIBC

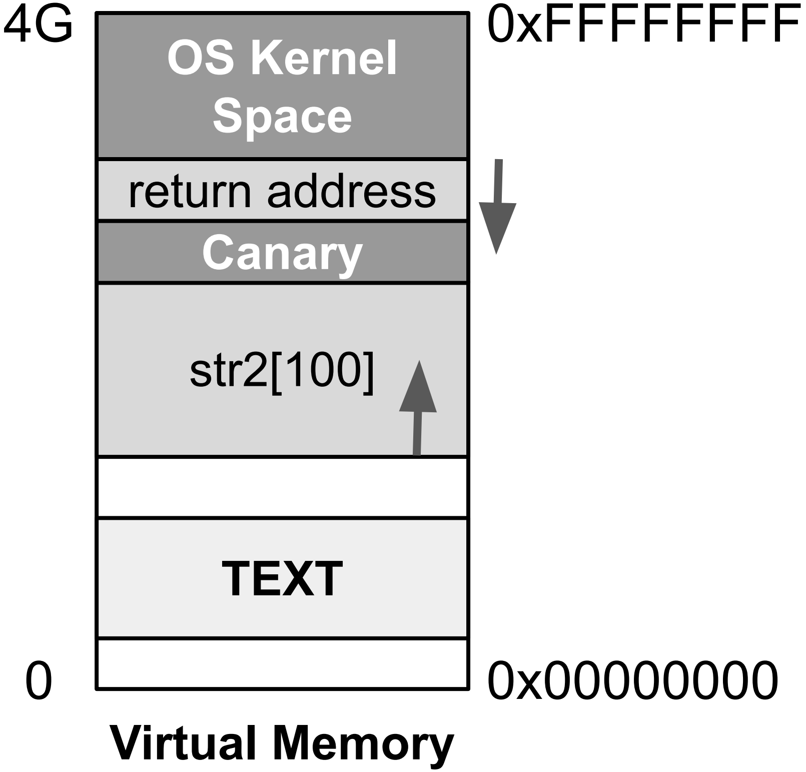
- Attack payload:
  - An address to libc function.
  - Function arguments.
- Limitations:
  - Execute whole functions.
  - Cannot target functions with ‘00’ byte in address.



[3] Van der Veen et. al., Memory errors: The past, the present, and the future. [RAID 2012]

STACK CANARIES

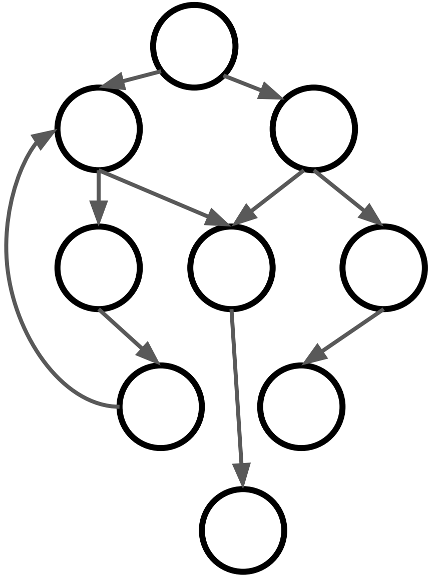
- Main Idea:
  - Insert unique variables on the stack.
  - Check their contents upon function return.
- Limitations:
  - Only detect continuous writes.
  - No read protection.



[3] Van der Veen et. al., Memory errors: The past, the present, and the future. **[RAID 2012]**

CONTROL FLOW INTEGRITY (CFI)

- Main Idea:
  - Construct a pre-defined CFG.
    - Statically with point-to analysis.
    - Dynamically with profiling.
  - Enforce it at runtime.
- Limitations:
  - Over-approximation.
  - Modularity.

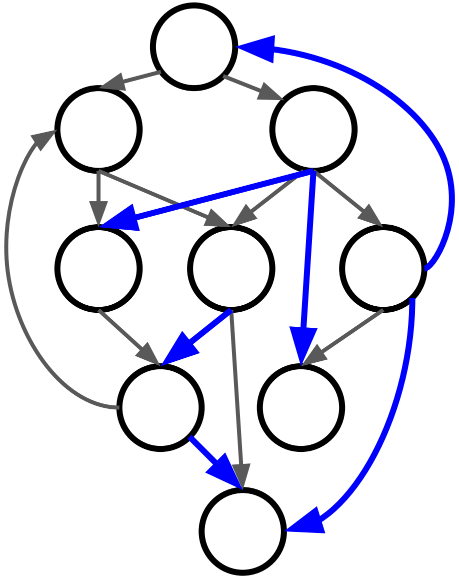


CFG

[12] Burow et. al, Control-flow integrity: Precision, security, and performance. **[ACM Surveys 2017]**

CONTROL FLOW INTEGRITY (CFI)

- Main Idea:
  - Construct a pre-defined CFG.
    - Statically with point-to analysis.
    - Dynamically with profiling.
  - Enforce it at runtime.
- Limitations:
  - **Over-approximation.**
  - Modularity.

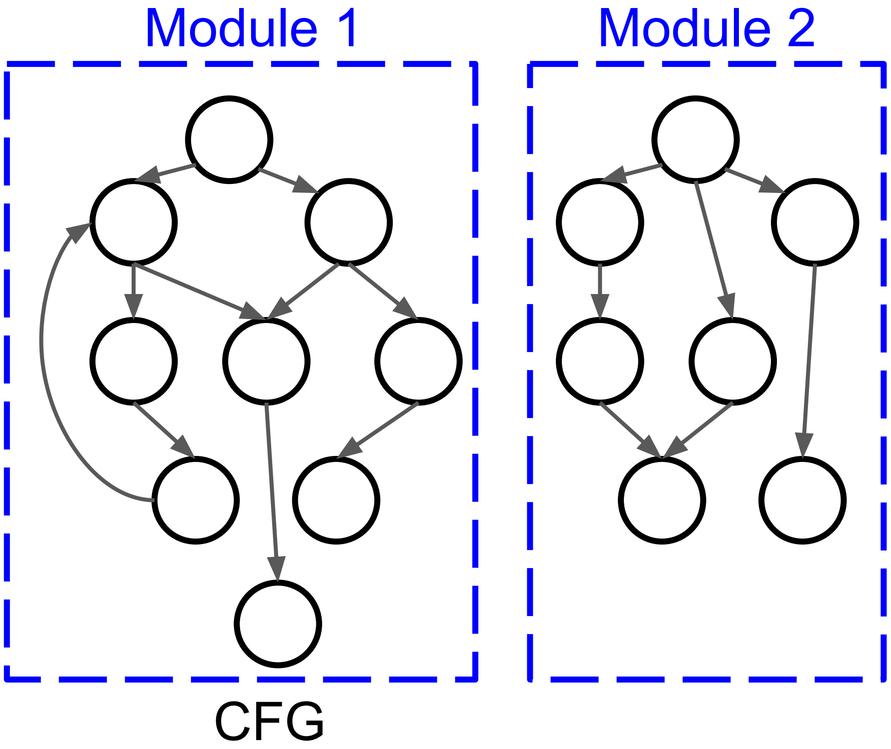


CFG

[12] Burow et. al, Control-flow integrity: Precision, security, and performance. **[ACM Surveys 2017]**

CONTROL FLOW INTEGRITY (CFI)

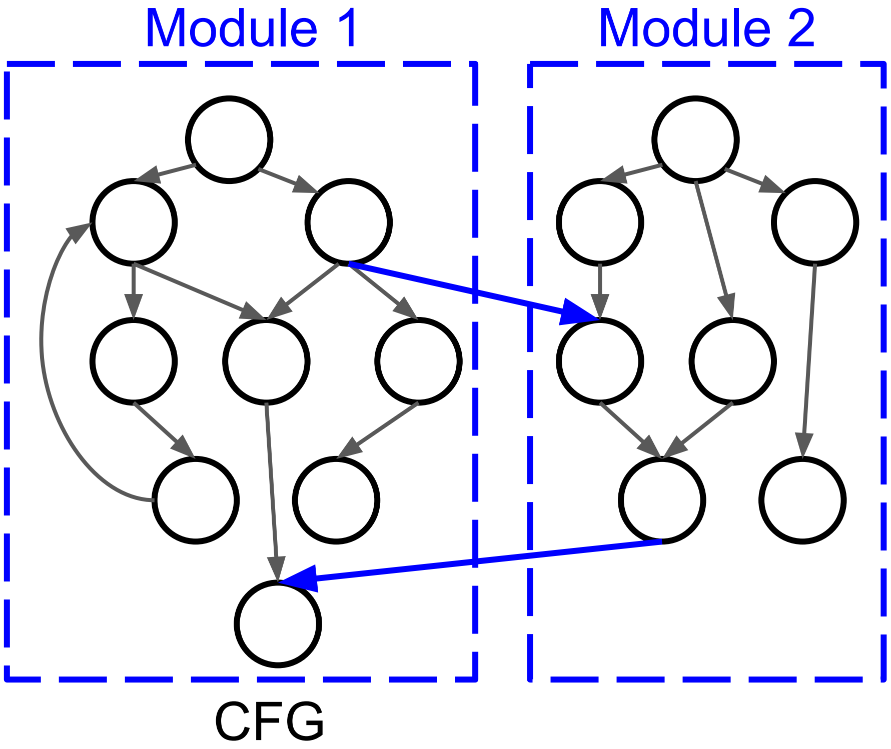
- Main Idea:
    - Construct a pre-defined CFG.
        - Statically with point-to analysis.
        - Dynamically with profiling.
    - Enforce it at runtime.
- Limitations:
    - Over-approximation.
    - **Modularity.**



[12] Burow et. al, Control-flow integrity: Precision, security, and performance. **[ACM Surveys 2017]**

CONTROL FLOW INTEGRITY (CFI)

- Main Idea:
  - Construct a pre-defined CFG.
    - Statically with point-to analysis.
    - Dynamically with profiling.
  - Enforce it at runtime.
- Limitations:
  - Over-approximation.
  - **Modularity.**

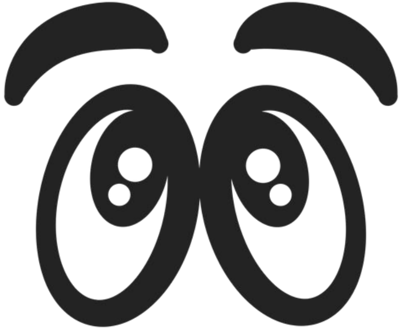


[12] Burow et. al, Control-flow integrity: Precision, security, and performance. **[ACM Surveys 2017]**

Counterfeit Object Oriented Programming

[9] Schuster et. al, Counterfeit object-oriented programming. **[S&P 2015]**

What are C++ Virtual Pointers?

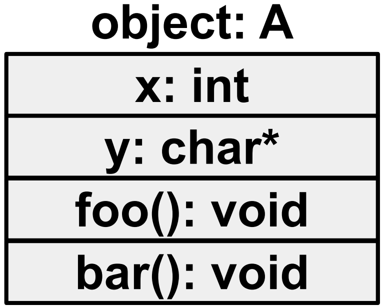


C++ CONCEPTS: OBJECT-ORIENTED

```
class A {
  public:
    int x;
    char *y;

    void foo();
    void bar();
}
```

C++ CONCEPTS: OBJECT-ORIENTED

```
class A {
  public:
    int x;
    char *y;

    void foo();
    void bar();
}
```



object: A

x: int

y: char*

foo(): void

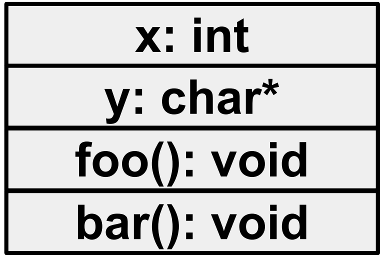
bar(): void

C++ CONCEPTS: INHERITANCE

```
class A {
  public:
    int x;
    char *y;

    void foo();
    void bar();
}
```

```
class B : public A {
  public:
    int z;




}
```

object: A



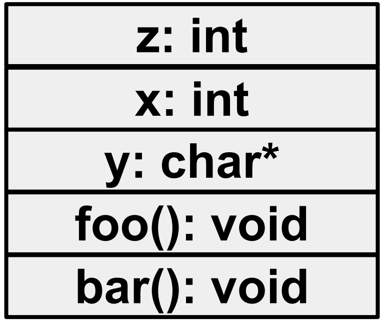
x: int

y: char*

foo(): void

bar(): void

object: B



z: int

x: int

y: char*

foo(): void

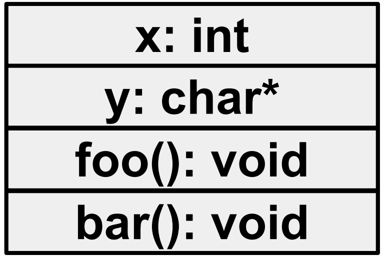
bar(): void

C++ CONCEPTS: POLYMORPHISM

```
class A {
  public:
    int x;
    char *y;

    void foo();
    virtual void bar();
}
```

```
class B : public A {
  public:
    int z;


    void bar();
}
```

object: A



x: int

y: char*

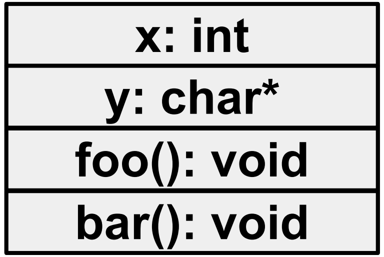
foo(): void

bar(): void

C++ CONCEPTS: COMPILER

```
class A {
  public:
    int x;
    char *y;

    void foo();
    virtual void bar();
}
```

object: A



x: int

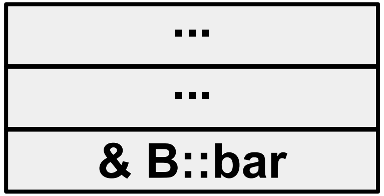
y: char*

foo(): void

bar(): void

```
class B : public A {
  public:
    int z;



     void bar();
}
```

B::vTable



...

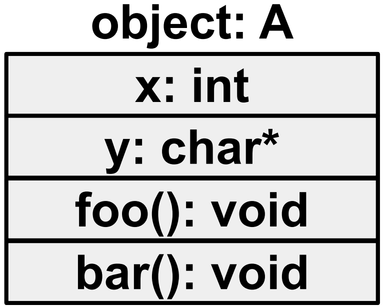
...

& B::bar

C++ CONCEPTS: COMPILER

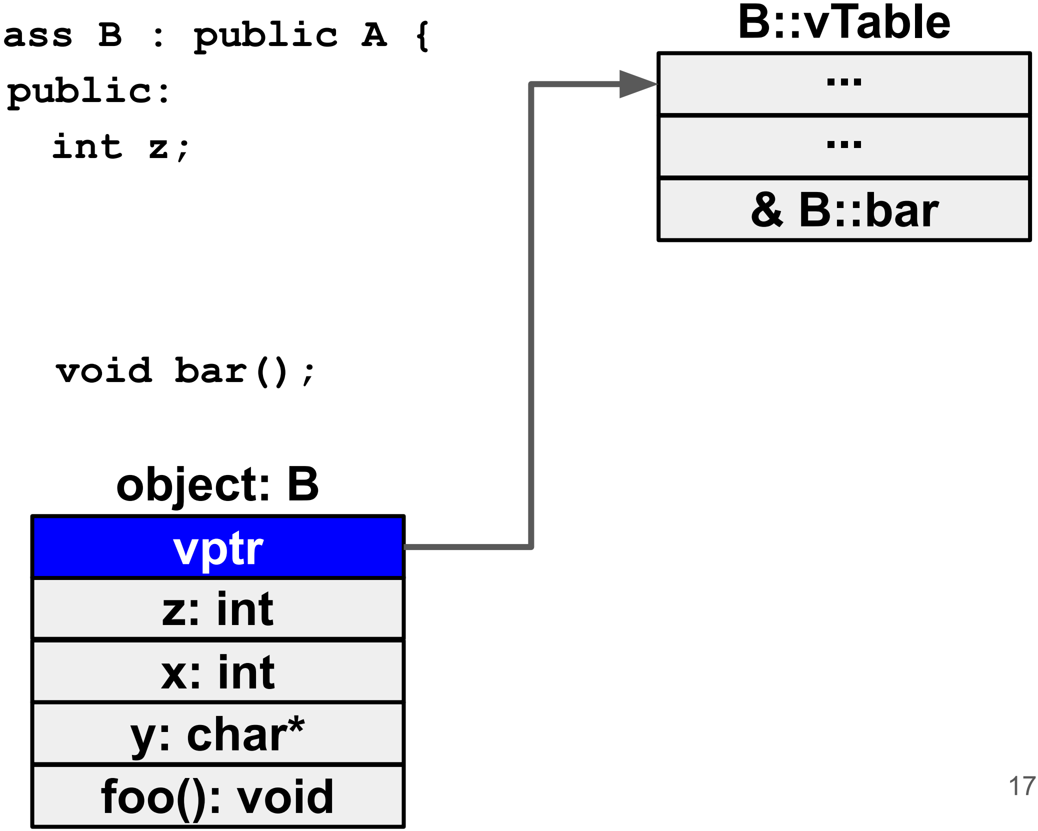
```
class A {
  public:
    int x;
    char *y;

    void foo();
    virtual void bar();
}
```

```
class B : public A {
  public:
    int z;



    void bar();
}
```



object: A

x: int
y: char*
foo(): void
bar(): void



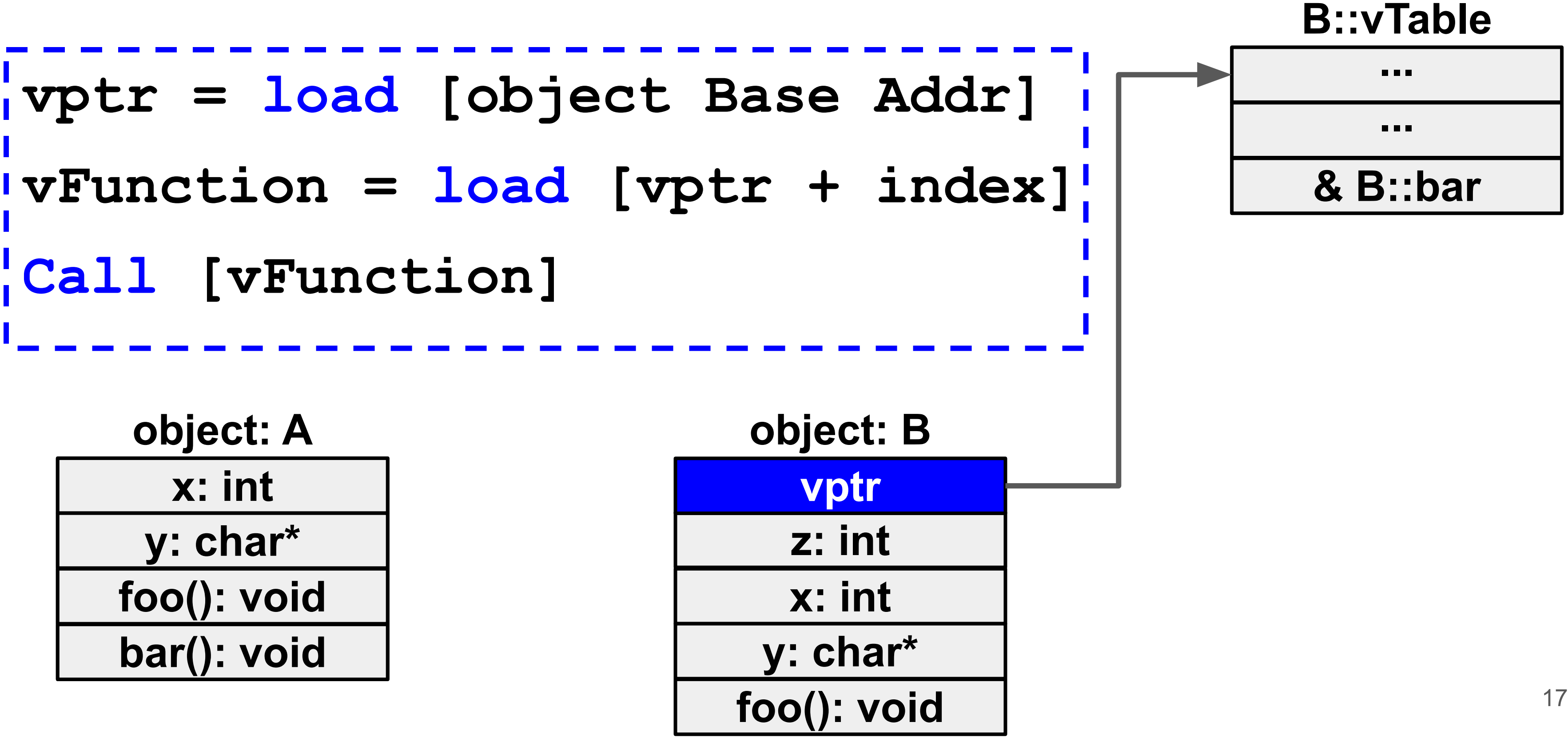
object: B

vptr
z: int
x: int
y: char*
foo(): void

B::vTable

...
...
& B::bar

C++ CONCEPTS: COMPILER



```
vptr = load [object Base Addr]
vFunction = load [vptr + index]
Call [vFunction]
```

B::vTable

...
...
& B::bar

object: A

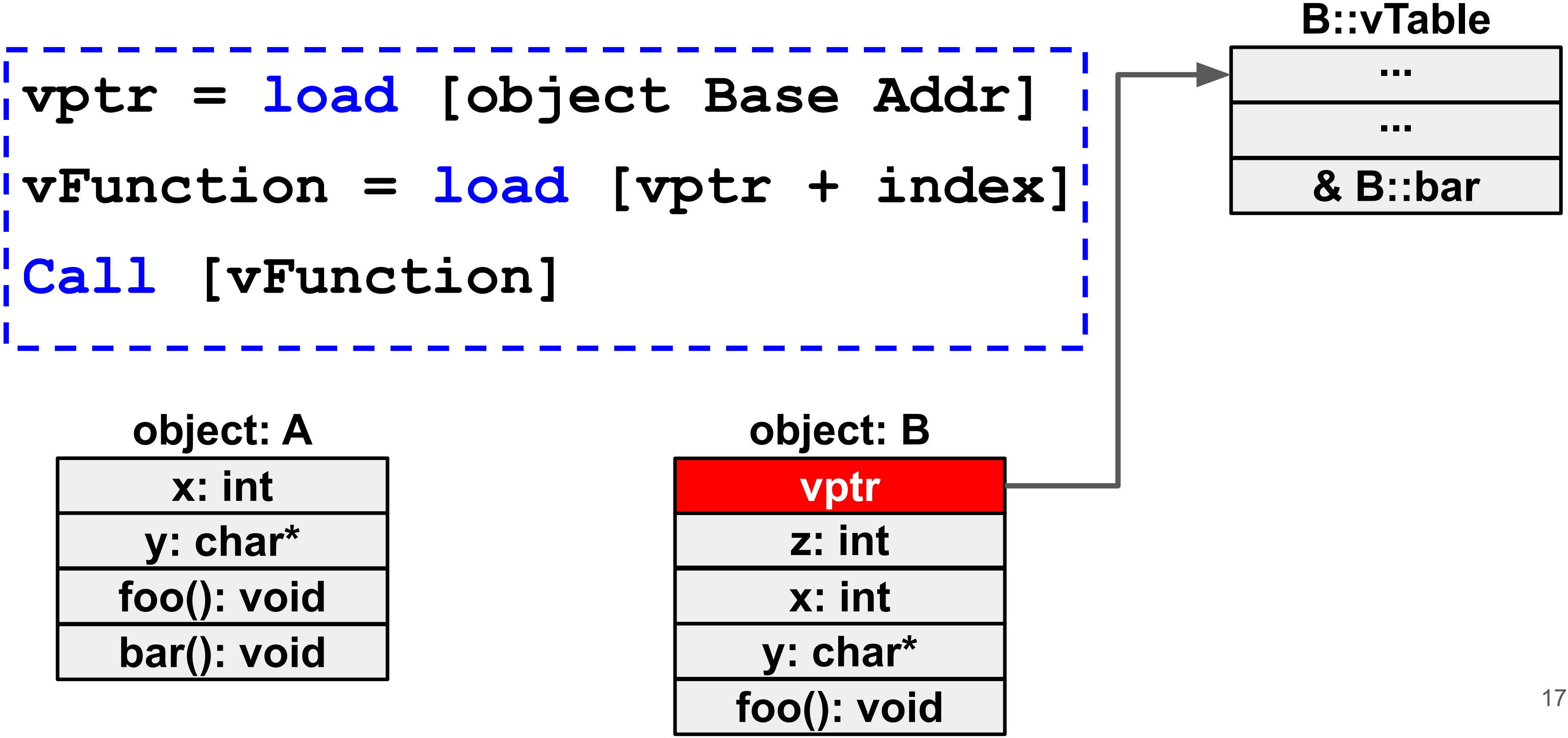
x: int
y: char*
foo(): void
bar(): void

object: B

vptr
z: int
x: int
y: char*
foo(): void

COUNTERFEIT OBJECT ORIENTED PROGRAMMING (COOP)

```
vptr = load [object Base Addr]
vFunction = load [vptr + index]
Call [vFunction]
```



object: A

x: int
y: char*
foo(): void
bar(): void

object: B

vptr
z: int
x: int
y: char*
foo(): void

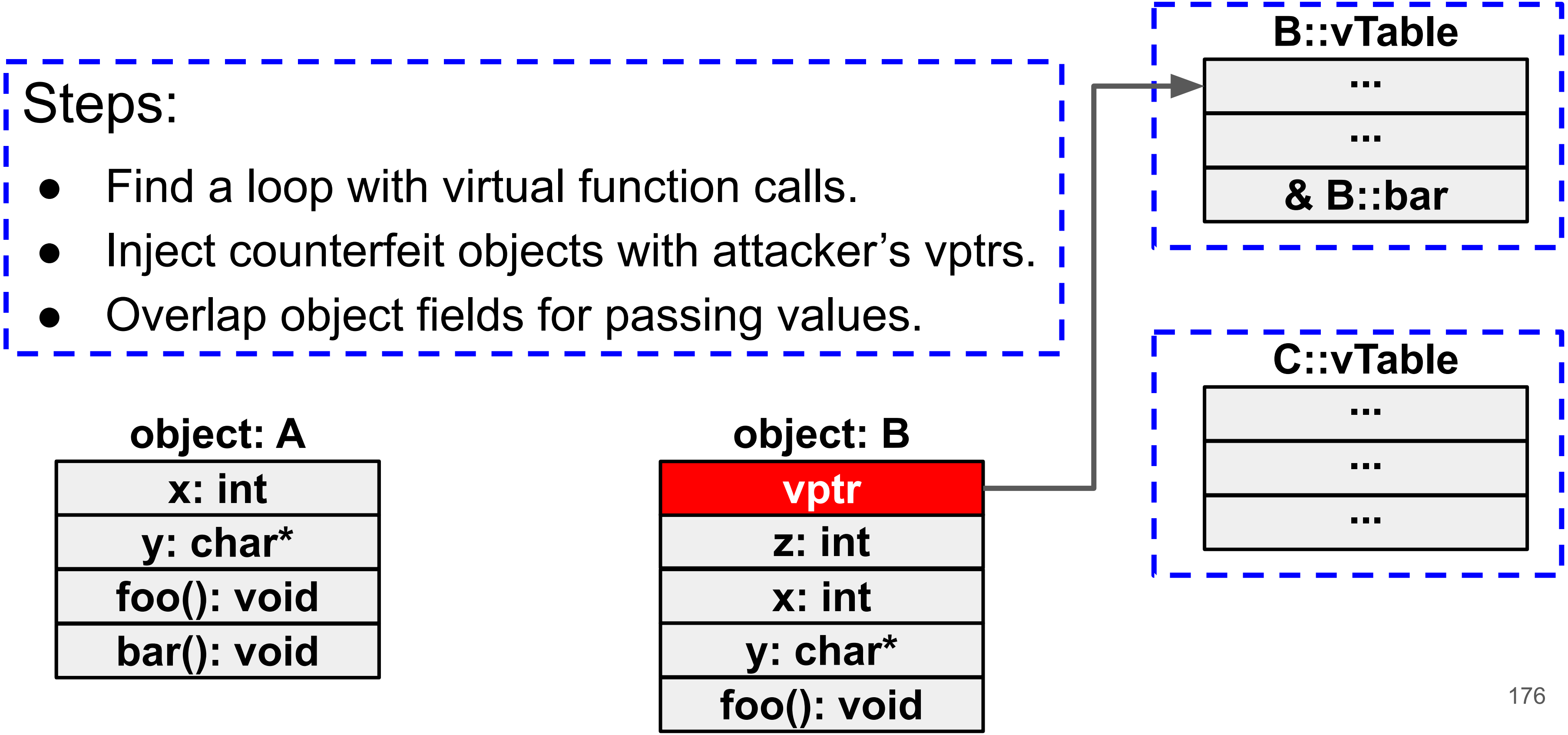
B::vTable

...
...
& B::bar

😈 COUNTERFEIT OBJECT ORIENTED PROGRAMMING (COOP)

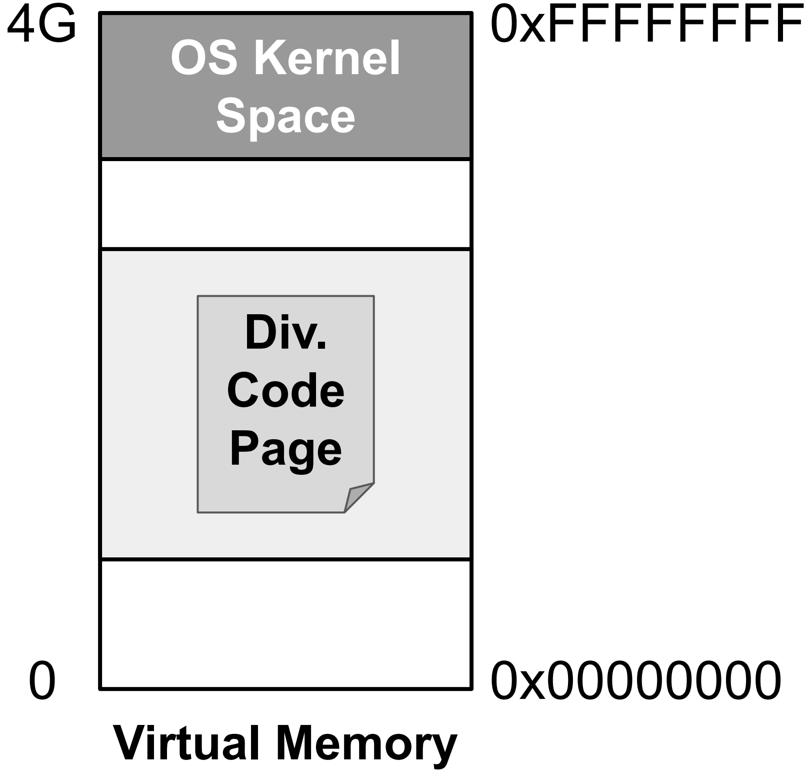
Steps:

- Find a loop with virtual function calls.
- Inject counterfeit objects with attacker’s vptrs.
- Overlap object fields for passing values.



😈 JUST-IN-TIME ROP

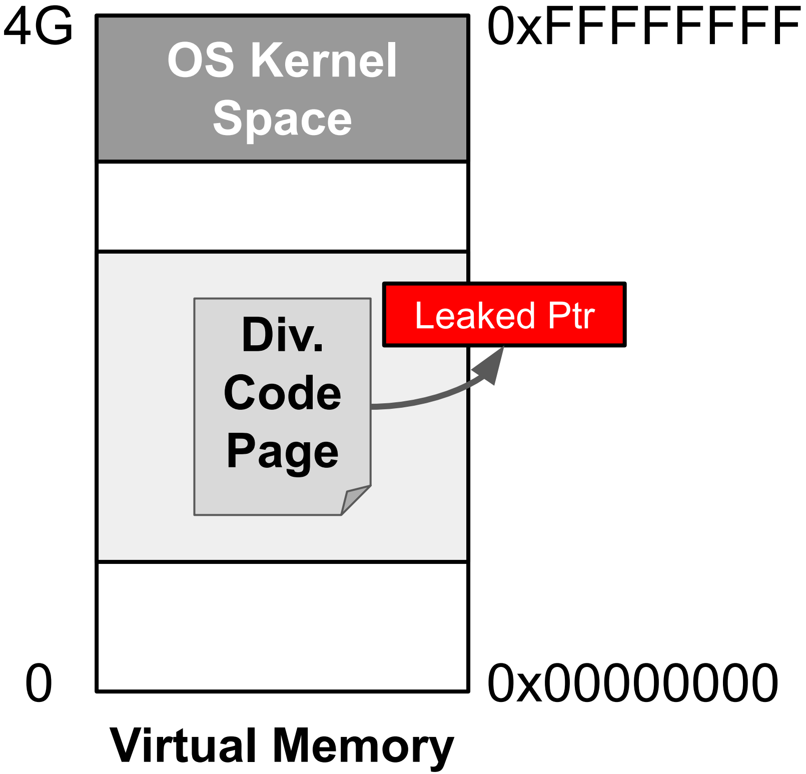
- Main Idea:
  - Repeatedly abuse a memory disclosure.



[8] Snow et. al,Just-in-time code reuse: On the effectiveness of fine-grained ASLR. **[S&P 2013]**

😈 JUST-IN-TIME ROP

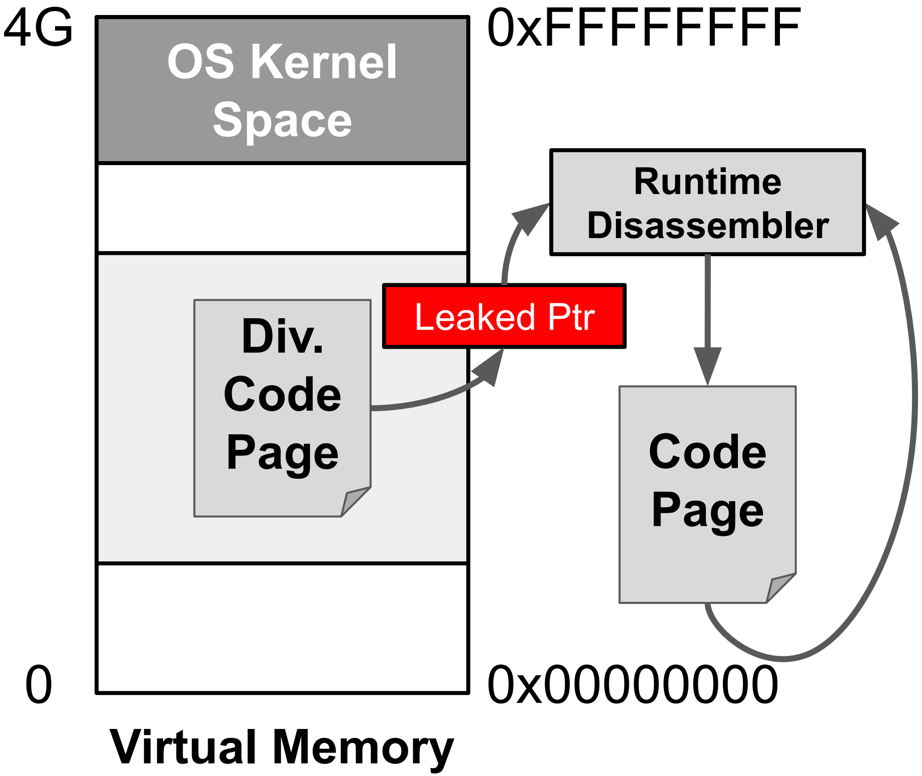
- Main Idea:
  - Repeatedly abuse a memory disclosure.
- Attack Steps:
  - **Leak one code pointer.**



[8] Snow et. al,Just-in-time code reuse: On the effectiveness of fine-grained ASLR. **[S&P 2013]**

😈 JUST-IN-TIME ROP

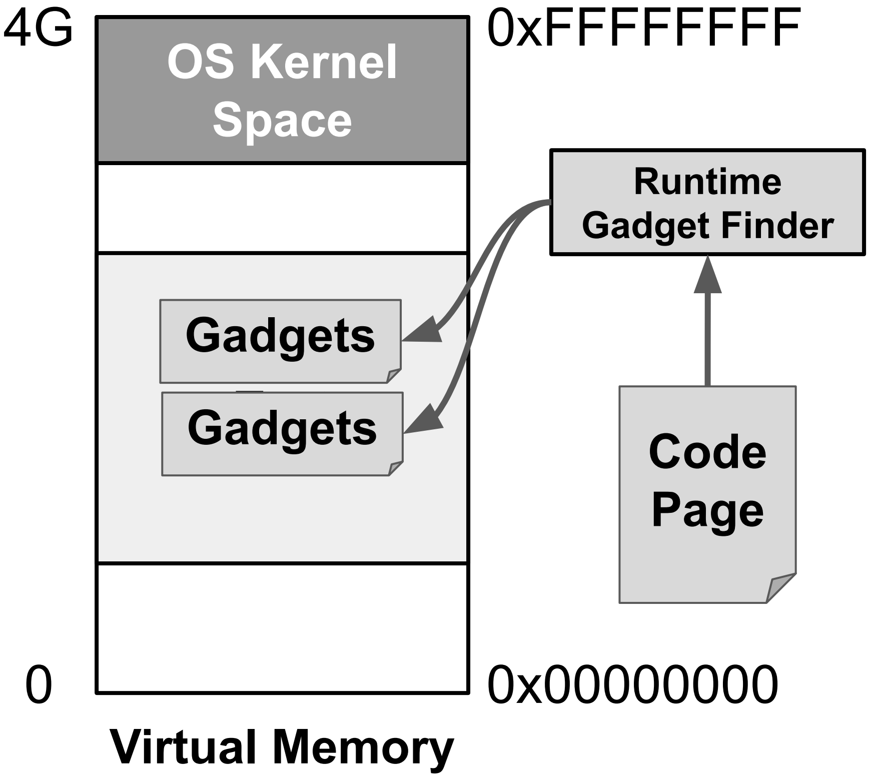
- Main Idea:
  - Repeatedly abuse a memory disclosure.
- Attack Steps:
  - Leak one code pointer.
  - **Scan code pages on the fly.**



[8] Snow et. al,Just-in-time code reuse: On the effectiveness of fine-grained ASLR. **[S&P 2013]**

😈 JUST-IN-TIME ROP

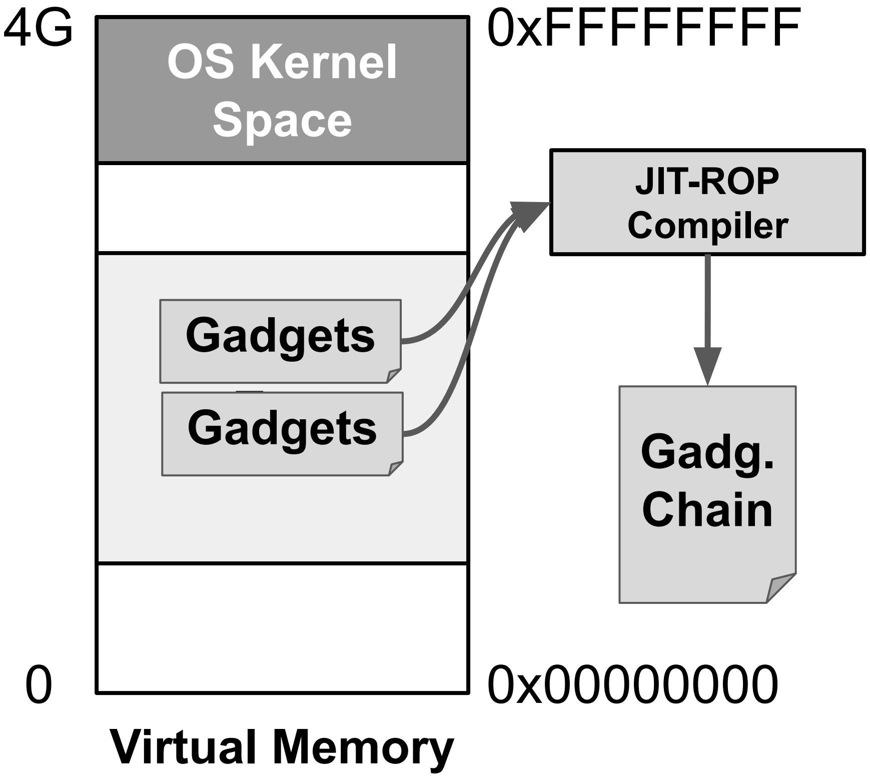
- Main Idea:
  - Repeatedly abuse a memory disclosure.
- Attack Steps:
  - Leak one code pointer.
  - Scan code pages on the fly.
  - **Pinpoint useful gadgets**.



[8] Snow et. al,Just-in-time code reuse: On the effectiveness of fine-grained ASLR. **[S&P 2013]**

😈 JUST-IN-TIME ROP

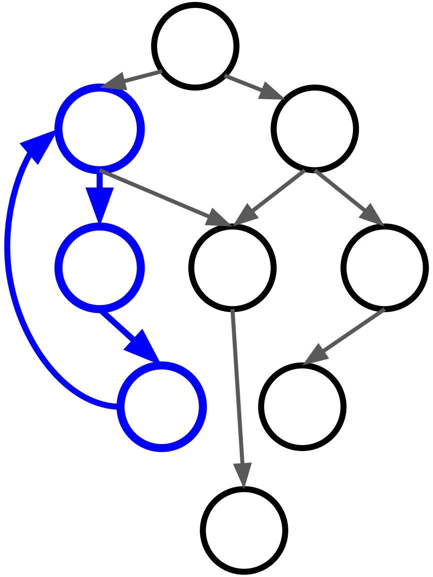
- Main Idea:
  - Repeatedly abuse a memory disclosure.
- Attack Steps:
  - Leak one code pointer.
  - Scan code pages on the fly.
  - Pinpoint useful gadgets.
  - **JIT-compile an ROP gadget chain.**



[8] Snow et. al,Just-in-time code reuse: On the effectiveness of fine-grained ASLR. **[S&P 2013]**

😈 DATA ORIENTED PROGRAMMING (DOP)

- Attack Steps:
    - Trigger a memory safety vulnerability.
    - Manipulate non control data.
    - Use the corrupted data.
- Goal:
    - Never change program CFG.

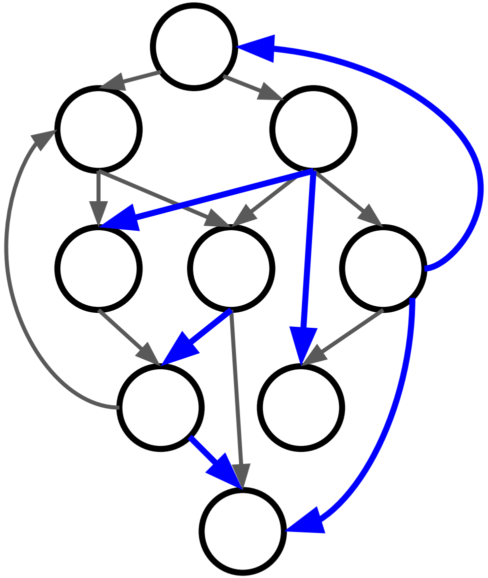


CFG

[10] Cheng et. al., Exploitation techniques and defenses for data-oriented attacks. **[SecDev 2019]**

DATA FLOW INTEGRITY

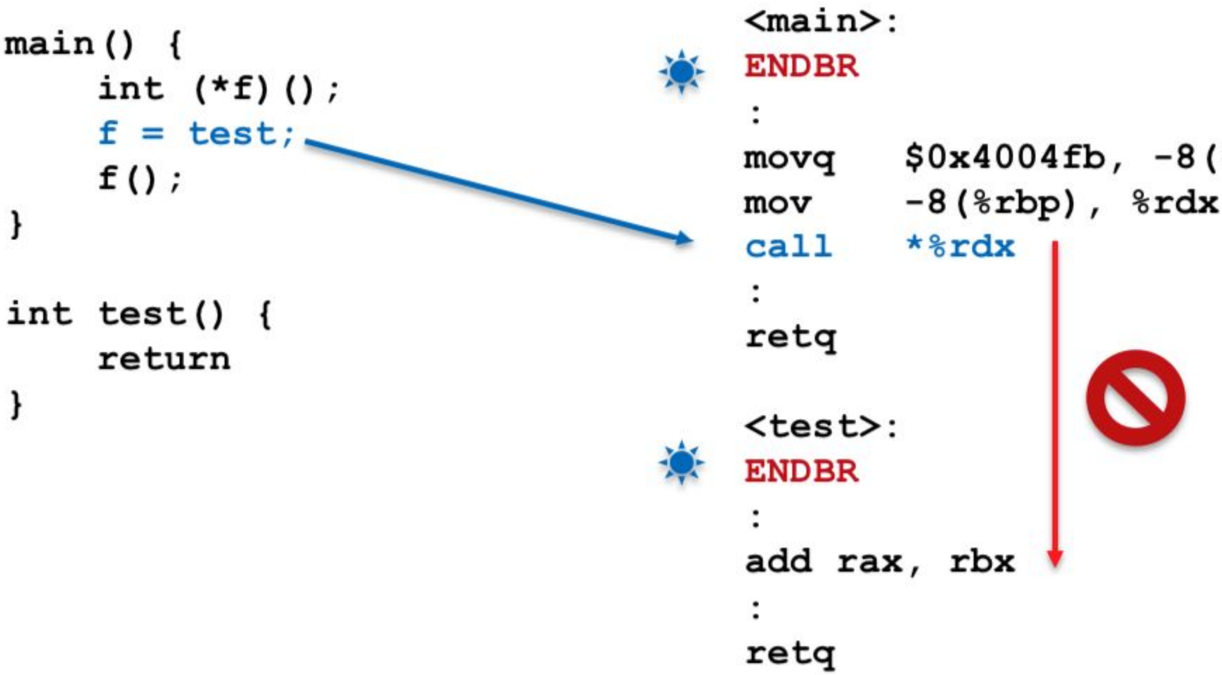
- Main Idea:
  - Construct a compile-time DFG.
    - Load inst. → {IDs of store insts.} with point-to analysis.
  - Enforce it at runtime.
    - Tag every memory word with 2-byte shadow.
    - Write the ID to the Tag upon store.
    - Compare ID vs. set upon load.
- Limitations:
  - Over-approximation.



DFG

[13] Castro et. al., Securing software by enforcing data-flow integrity. **[OSDI 2006]**

INTEL CONTROL FLOW ENF. TECH. (CET)

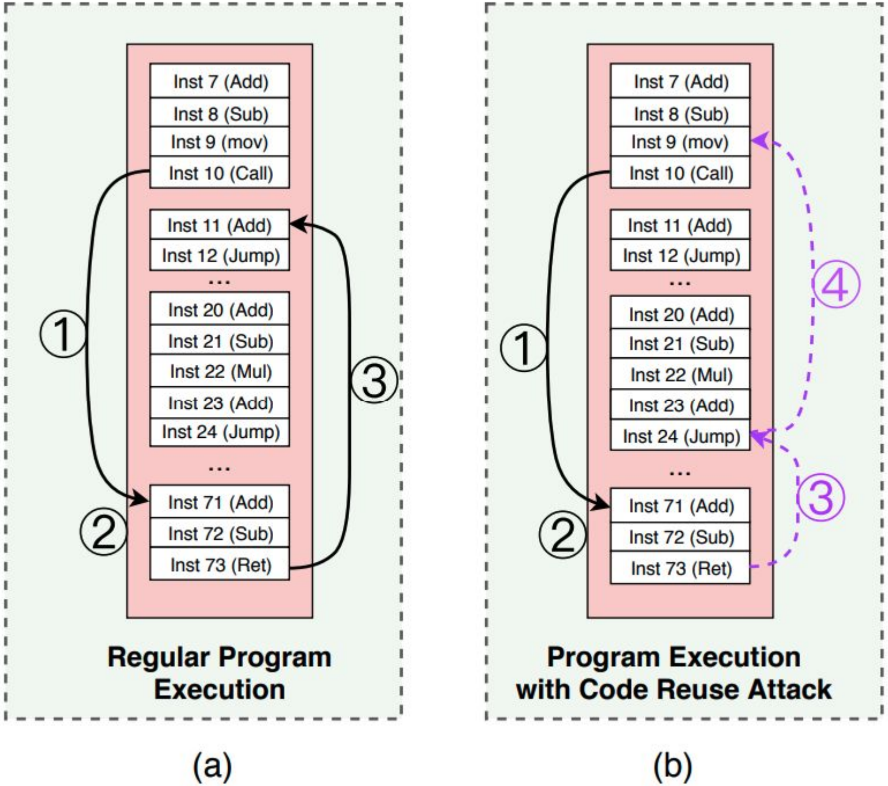


```
main() {
     int (*f)();
     f = test;
     f();
}

int test() {
     return
}
```

```
<main>:
ENDBR
:
movq    $0x4004fb, -8(%rbp)
mov     -8(%rbp), %rdx
call    *%rdx
:
retq

<test>:
ENDBR
:
add rax, rbx
:
retq
```

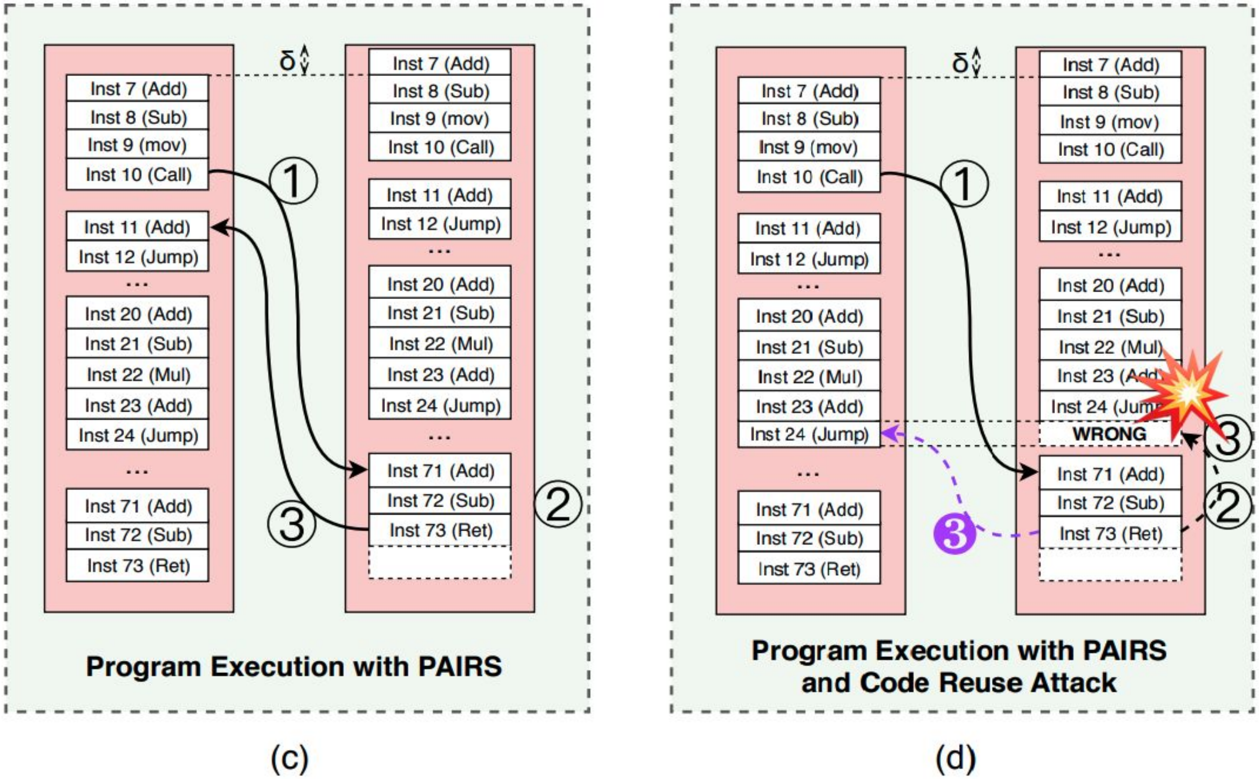
Source:
https://www.linuxplumbersconf.org/event/2/contributions/147/attachments/72/83/CET-LPC-2018.pdf

PAIRS vs. CODE REUSE ATTACKS



[2] Mohamed et. al, PAIRS: Control flow protection using phantom addressed instructions. **[arXiv 2019]**

PAIRS vs. CODE REUSE ATTACKS

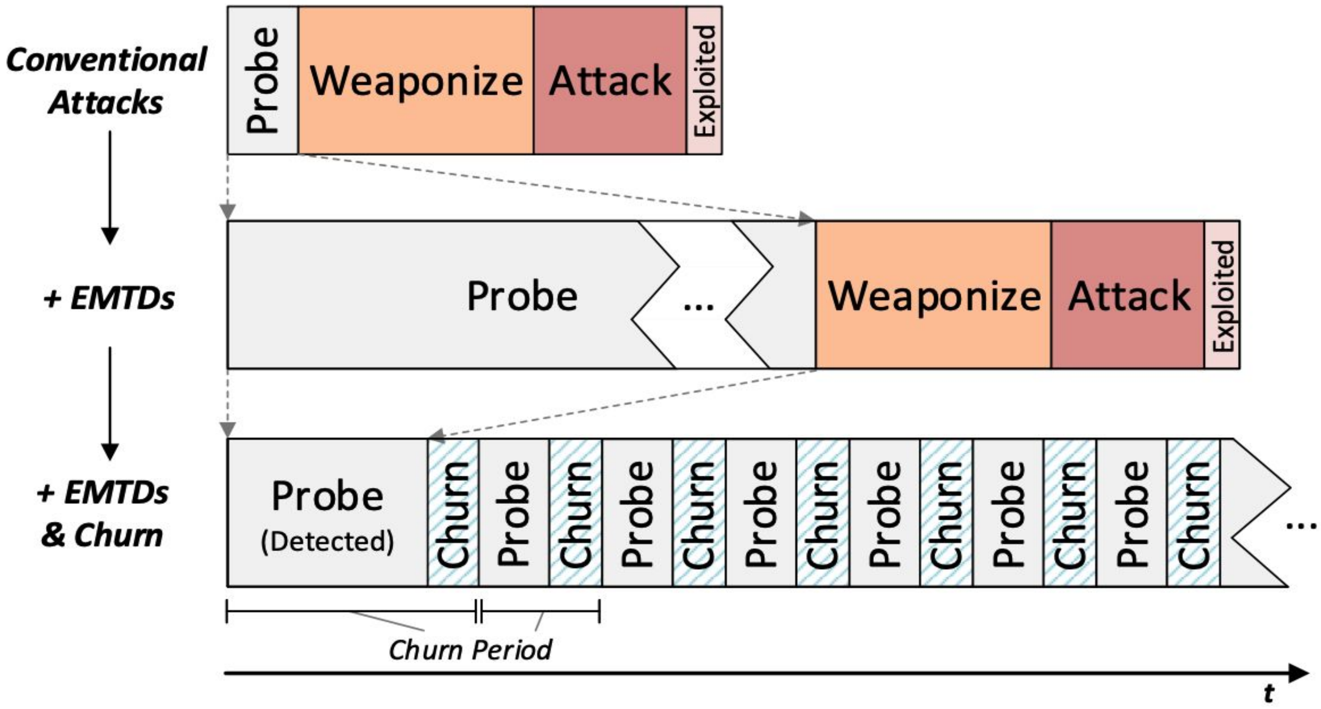


[2] Mohamed et. al, PAIRS: Control flow protection using phantom addressed instructions. **[arXiv 2019]**

DATA SPACE RANDOMIZATION

- Main Idea:
  - Randomize the representation of data stored in memory.
- Approaches:
  - Use encryption with a unique key per variable.
    - DSR.
  - Statically randomize structs layout in a program.
    - GCC struct randomization.
  - Dynamically randomize objects layout in memory.
    - SALADS, SmokeStack, and POLAR.

MOVING TARGET DEFENSE (MORPHEUS)



Gallagher et. al., Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn. **[ASPLOS 2019]**

CALIFORMS: INSERTION POLICIES

```
struct A_opportunistic
{
  char c;
  char tripwire[3];
  int i;
  char buf[64];
  void (*fp)();
}
```

(1) Opportunistic

```
struct A_full {
  char tripwire[2];
  char c;
  char tripwire[1];
  int i;
  char tripwire[3];
  char buf[64];
  char tripwire[2];
  void (*fp)();
  char tripwire[1];
}
```

(2) Full

```
struct A_intelligent {
  char c;
  int i;
  char tripwire[3];
  char buf[64];
  char tripwire[2];
  void (*fp)();
  char tripwire[3];
}
```

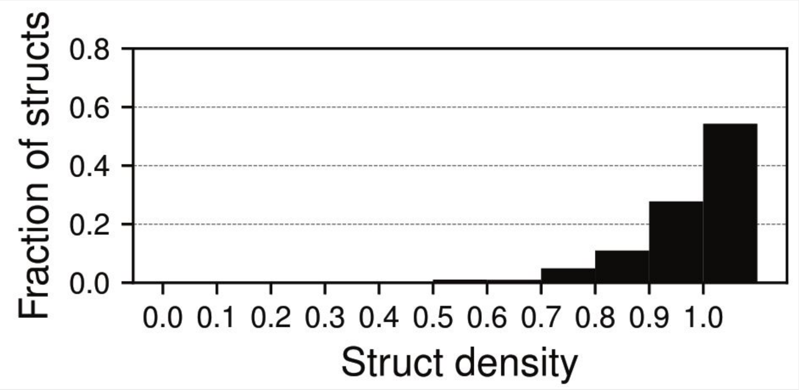
(3) Intelligent

Tripwire Insertion Policies

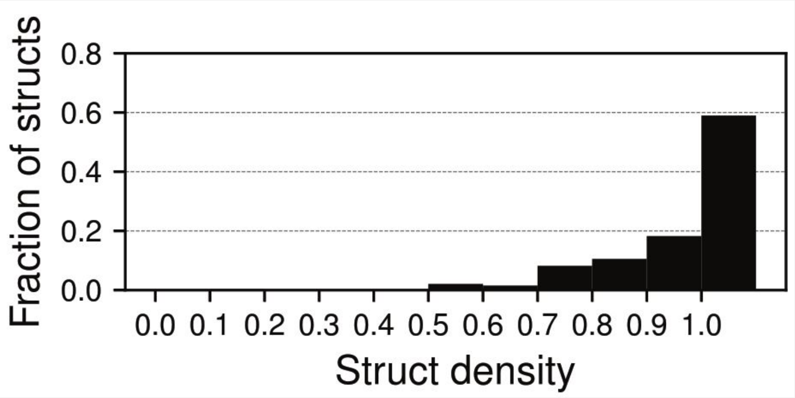
[1] Sasaki et. al., Practical byte-granular memory blacklisting using Califorms. **[MICRO 2019]**

CALIFORMS: DEAD BYTES

Normally Occurring Dead Bytes



SPEC CPU2006 C and C++ Benchmarks

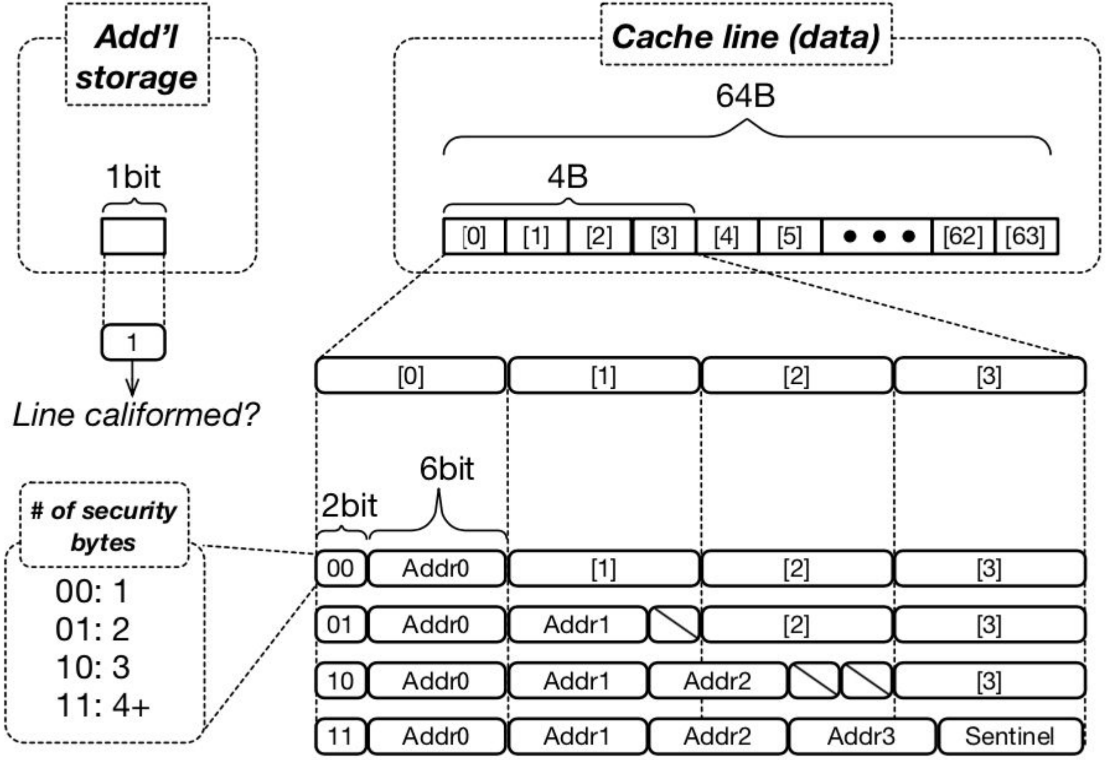


V8 JavaScript Engine

Struct density = $\sum_{\mathtt{i}}^{\mathtt{\#fields}}$`(sizeof(field`$_{\mathtt{i}}$`))/sizeof(struct)`

[1] Sasaki et. al., Practical byte-granular memory blacklisting using Califorms. **[MICRO 2019]**

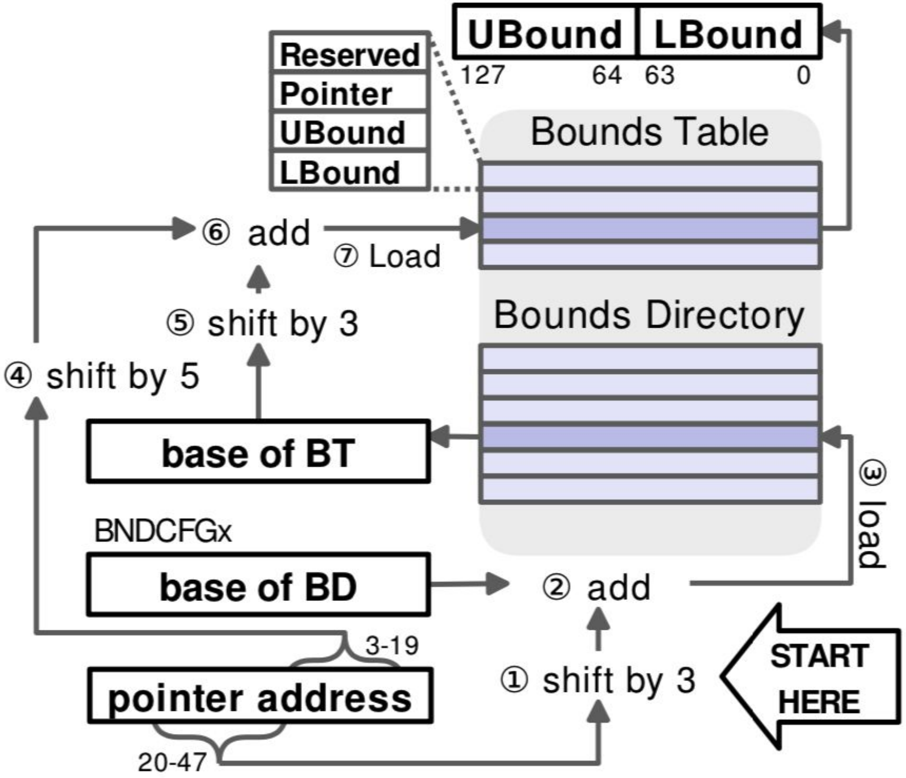
CALIFORMS: CACHELINE FORMAT



[1] Sasaki et. al., Practical byte-granular memory blacklisting using Califorms. **[MICRO 2019]**

INTEL MPX

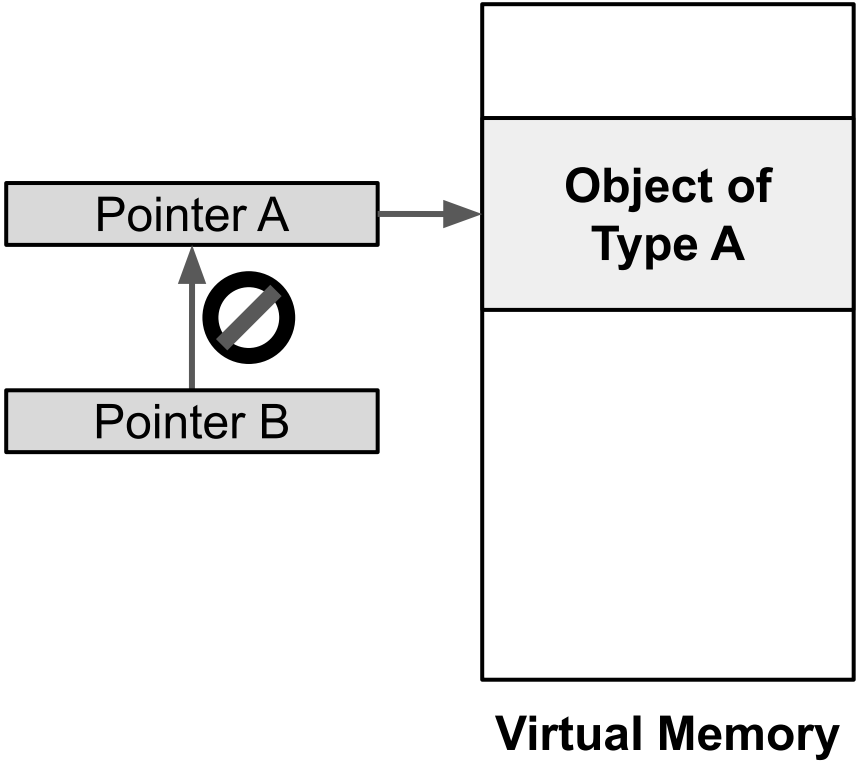
- Main Limitations:
  - High overheads (up to 4x).
  - Lack of multithreading.
  - Incorrect handling of several common C idioms.
  - Poor Interaction with other ISA extensions (like SGX).



[29] Oleksenko et. al., Intel MPX explained **[ACM Surveys 2018]**

TYPE SAFETY

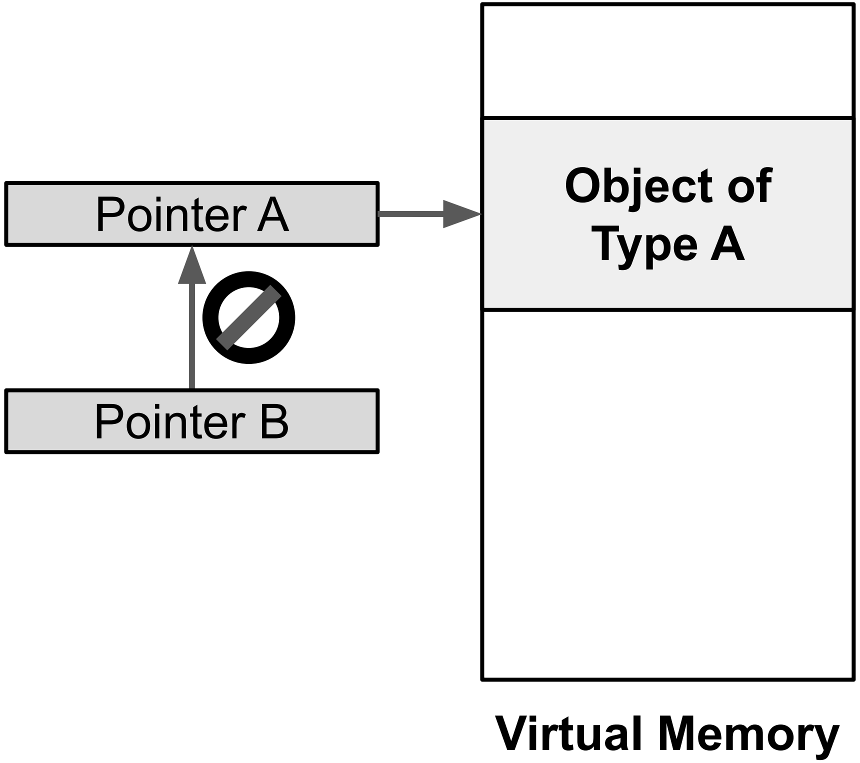
- Main Idea:
  - Add runtime checks to detect incompatible type casting.



[5] Song et. al., SoK: Sanitizing for security. **[S&P 2019]**

TYPE SAFETY

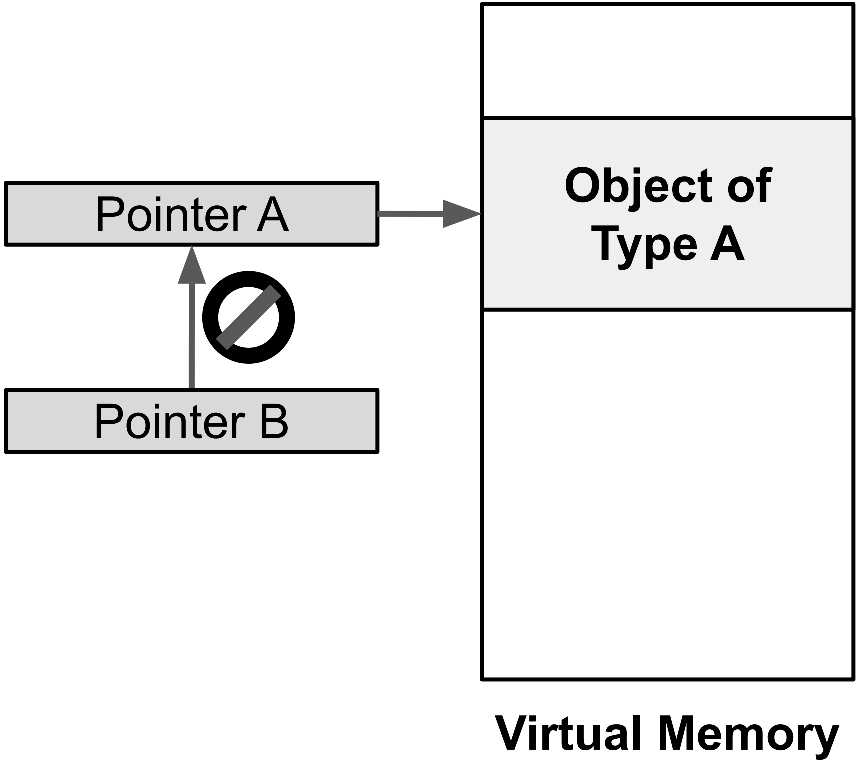
- Main Idea:
  - Add runtime checks to detect incompatible type casting.
- Examples:
  - UBSan and Clang CFI
    - RTTI-based verification.



[5] Song et. al., SoK: Sanitizing for security. **[S&P 2019]**

TYPE SAFETY

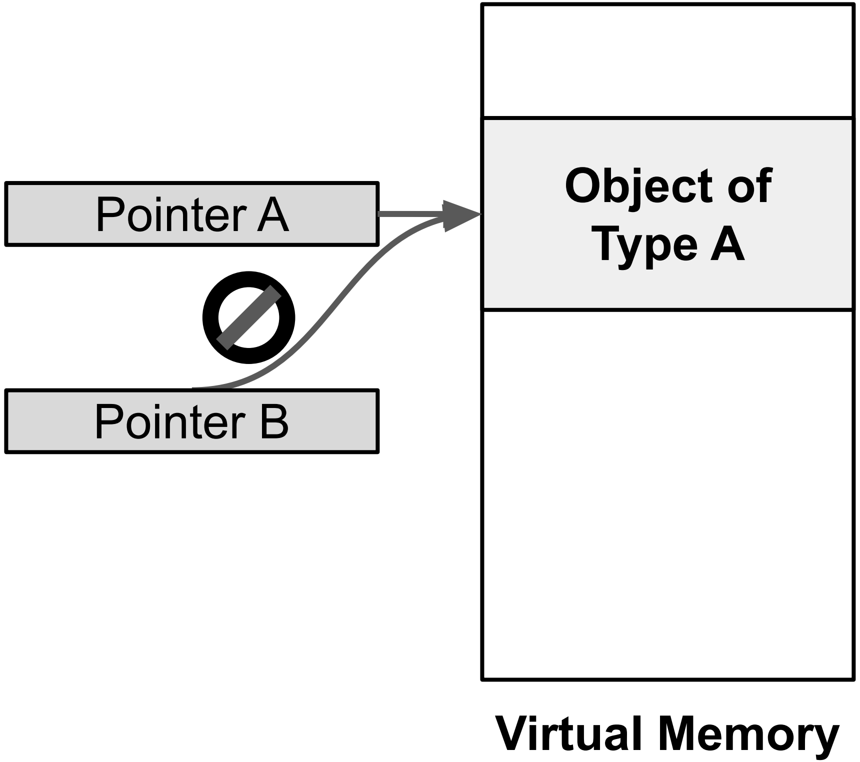
- Main Idea:
  - Add runtime checks to detect incompatible type casting.
- Examples:
  - UBSan and Clang CFI
    - RTTI-based verification.
  - CaVer, TypeSan, and HexType
    - Custom metadata.



[5] Song et. al., SoK: Sanitizing for security. **[S&P 2019]**

TYPE SAFETY

- Main Idea:
  - Add runtime checks to detect incompatible type casting.
- Examples:
  - UBSan and Clang CFI
    - RTTI-based verification.
  - CaVer, TypeSan, and HexType
    - Custom metadata.
  - Clang TySan and EffetiveSan
    - Check pointer dereference.



[5] Song et. al., SoK: Sanitizing for security. **[S&P 2019]**

End of Supplementary Slides