SOURCE-FREE BINARY SOFTWARE SECURITY RETROFITTING

DR. KEVIN HAMLEN

LOUIS A. BEECHERL, JR. DISTINGUISHED PROFESSOR

COMPUTER SCIENCE DEPARTMENT

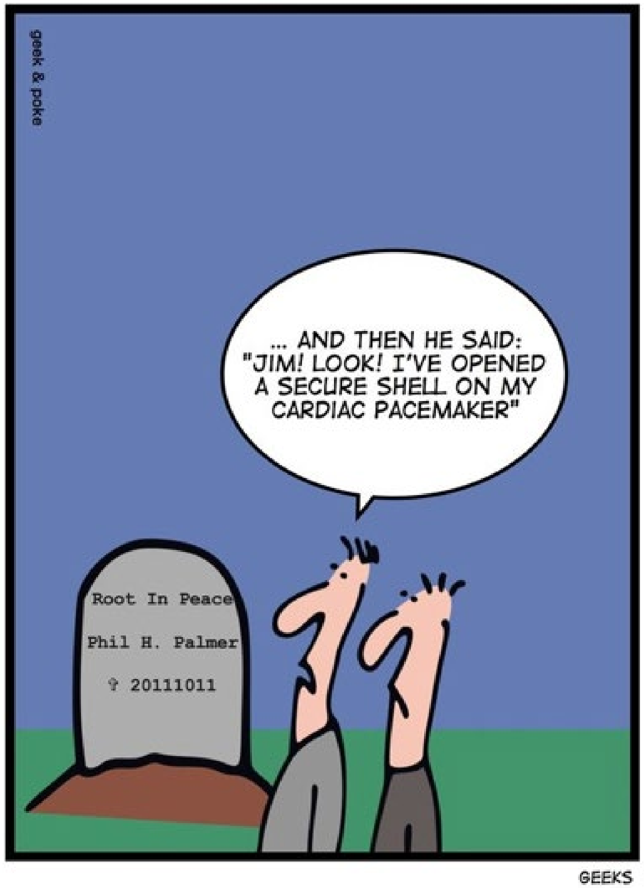
CYBER SECURITY RESEARCH AND EDUCATION INSTITUTE, EXECUTIVE DIRECTOR

THE UNIVERSITY OF TEXAS AT DALLAS

Supported in part by:
NSF Award #1513704 & NSF CAREER Award #1054629,
AFOSR YIP (Career) Award FA9550-08-1-0044,
AFOSR Active Defense Grant FA9550-10-1-0088,
AFOSR Award FA9550-14-1-0173,
ONR Awards N000141410030 & N00014-17-1-2995,
and an NSF I/UCRC Awards from Raytheon & Lockheed Martin

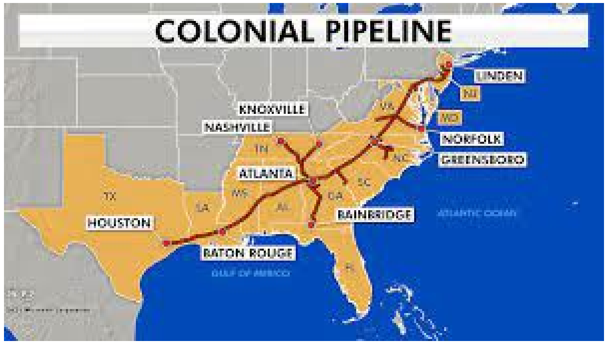
Mission-critical Software Environments

- **Myth:** In mission-critical environments, all software is custom, rigorously tested, and formally verified.
- **Reality:** Most mission-critical environments use commodity software and components extensively.
  - Commercial Off-The-Shelf (COTS)
    - widely available to attackers
  - mostly closed-source
    - independent security audit not feasible
  - supports mainstream OSes (Windows) and architectures (Intel)
  - some effort at secure development, but no formal guarantees



Critical Infrastructure: Critically Insecure

- 2020: Hundreds of US infrastructure networks penetrated by SolarWinds hack
  - **Software exploited:** Microsoft Exchange
  - Supply-line hack infects network monitors at Pentagon, Treasury, Microsoft, Intel, Cisco, …



- 2021: Colonial Oil Pipeline Hack
  - **Software exploited:** Unpatched Windows VPN
  - Leaked password to unused account, no multifactor authentication, no data backups
  - weeks of oil shortages in eastern US, tens of thousands of miles of pipeline checks

- 2010: Stuxnet infiltrates and destroys Iranian nuclear centrifuges
  - **Software exploited:** Siemens Windows apps and PLCs
  - Sets Iranian nuclear program back 3-5 years

(In)famous Linux Vulnerabilities

- Heartbleed
  - OpenSSL vulnerability disclosed April 2014
  - allowed anyone to anonymously grab arbitrary data (e.g., master keys) from internet-facing services
  - affected ~66% of all web servers, email servers, chat servers, VPNs, clients, etc.
  - all versions vulnerable since 2011!
- Shellshock
  - Bash shell vulnerability disclosed September 2014
  - allowed complete compromise - remote code execution
  - all versions vulnerable since 1989(!!)

Are In-house Projects "More Secure"?

- **Idea:** Build all your own custom software in-house from scratch (or contract trusted third-party to build from scratch).
  - expensive, time-consuming
  - error-prone (not built by specialists)
    - 63% of in-house IT projects fail to meet their own specs [CHAOS Report]
  - poor compatibility, hard to maintain
  - very questionable security assurance
    - vulnerable to insider threats, less tested, shaky design, etc.
    - assurance usually based on myth of "security by obscurity"
- Many COTS advantages
  - constantly updated for new threats
  - tested on a mass scale
  - crafted & maintained by specialists
  - cheaper, mass-produced

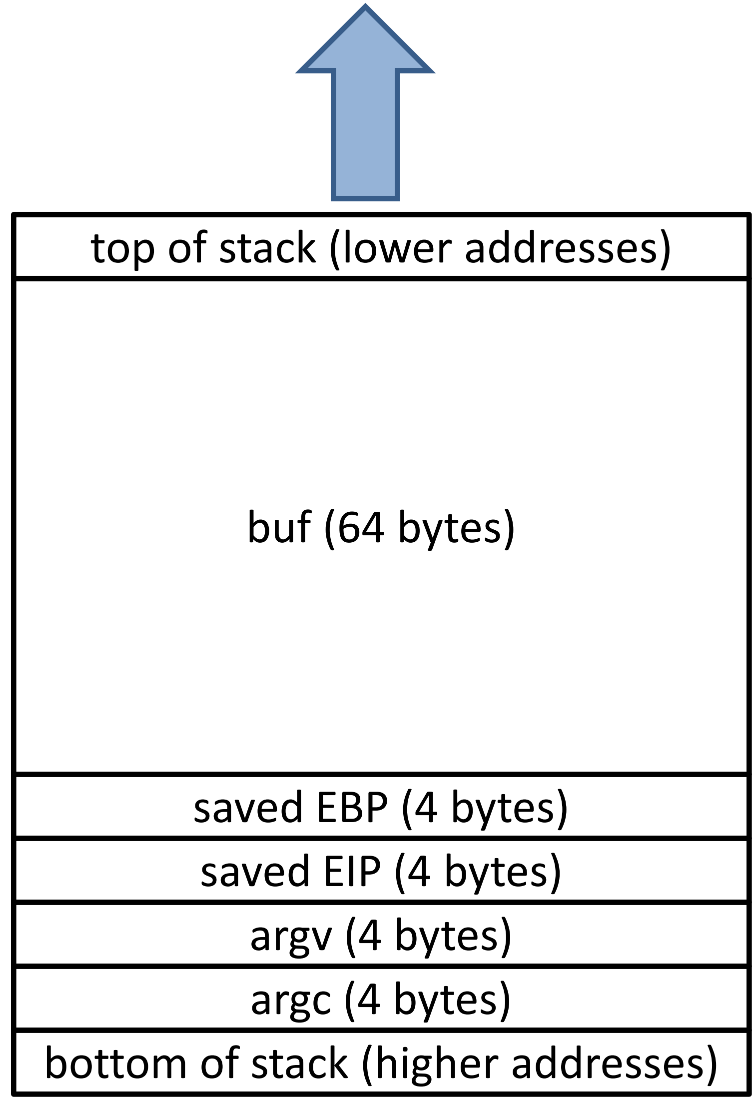
Why is Software so Insecure?

- Huge and constantly evolving
  - Windows XP has 40 million lines of code
  - Microsoft Office had 30 million lines in 2006
  - Debian 5.0 has a staggering 500 million lines!
    - contrast: Space shuttle has only 2.5 million moving parts!
- Often written in unsafe languages
  - C, C++, VC++, Visual Basic, scripting languages, …
- Increasingly sophisticated attacks
  - buffer-overrun
  - direct code-injection
  - return-to-libc
  - return-oriented programming (RoP)
  - implementation disclosure-assisted code-reuse attacks

Code-injection Example

```
8D 45 B8             lea eax,[ebp-48h]
50                   push eax
FF 15 BC 82 2F 01    call <system>
65 72 61 73 65 20    .data “erase ”
2A 2E 2A 20          .data “*.* ”
61 (x24)             .data “aaaaa…”
61 61 61 61          .data “aaaa”
30 FB 1F 00          <addr of buf>
```

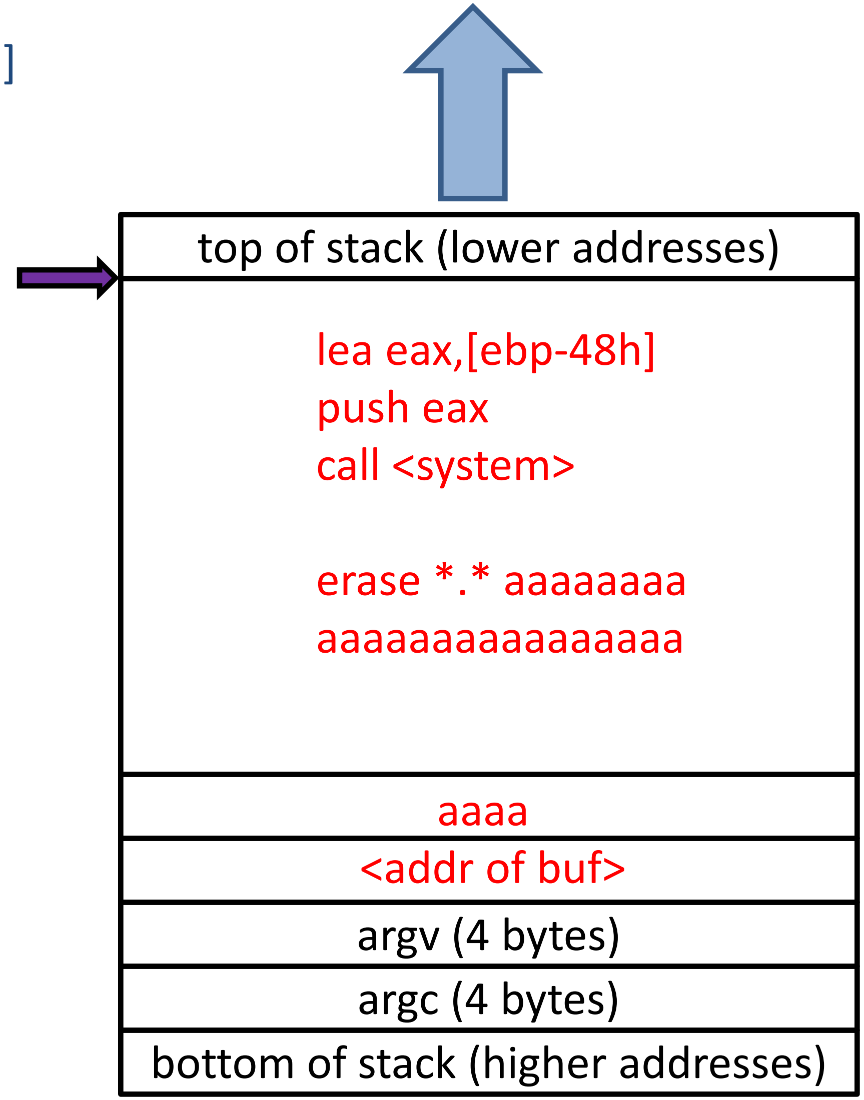
```
void main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return;
}
```



Code-injection Example

```
8D 45 B8             lea eax,[ebp-48h]
50                   push eax
FF 15 BC 82 2F 01    call <system>
65 72 61 73 65 20    .data “erase ”
2A 2E 2A 20          .data “*.* ”
61 (x24)             .data “aaaaa…”
61 61 61 61          .data “aaaa”
30 FB 1F 00          <addr of buf>
```

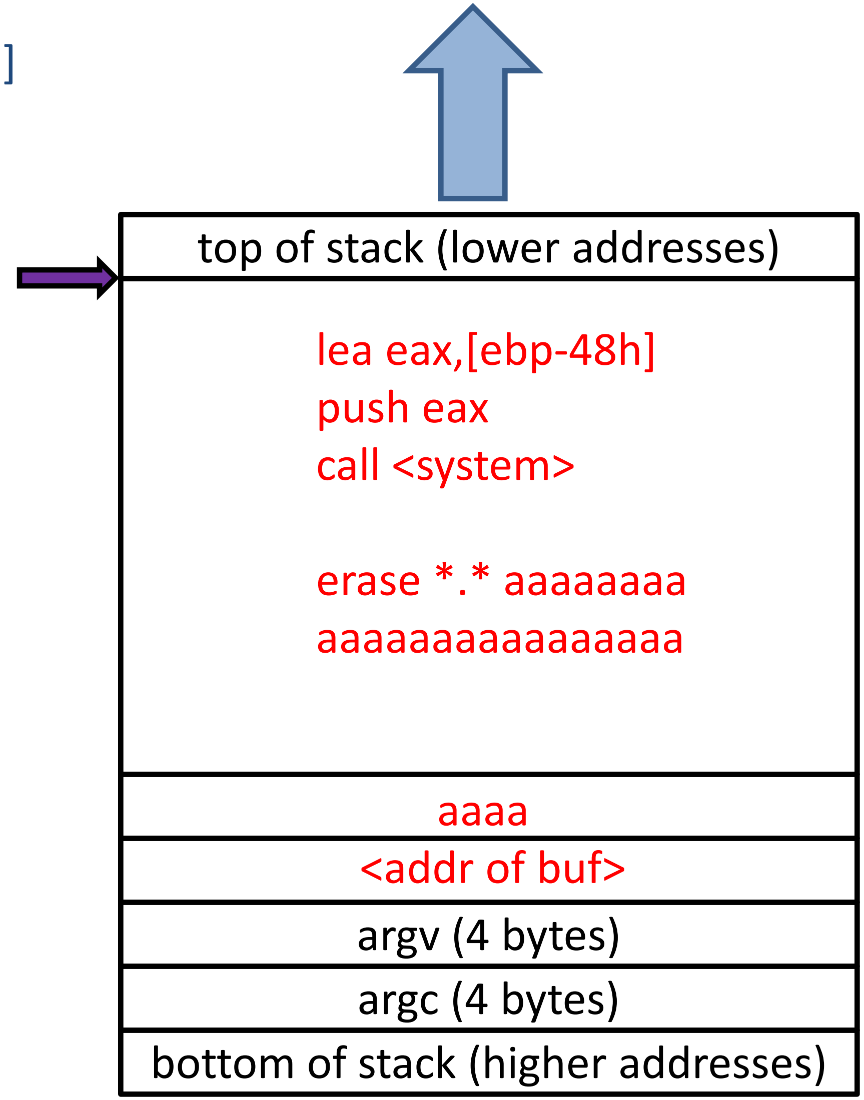
```
void main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return;
}
```



Code-injection Example

```
8D 45 B8             lea eax,[ebp-48h]
50                   push eax
FF 15 BC 82 2F 01    call <system>
65 72 61 73 65 20    .data “erase ”
2A 2E 2A 20          .data “*.* ”
61 (x24)             .data “aaaaa…”
61 61 61 61          .data “aaaa”
30 FB 1F 00          <addr of buf>
```

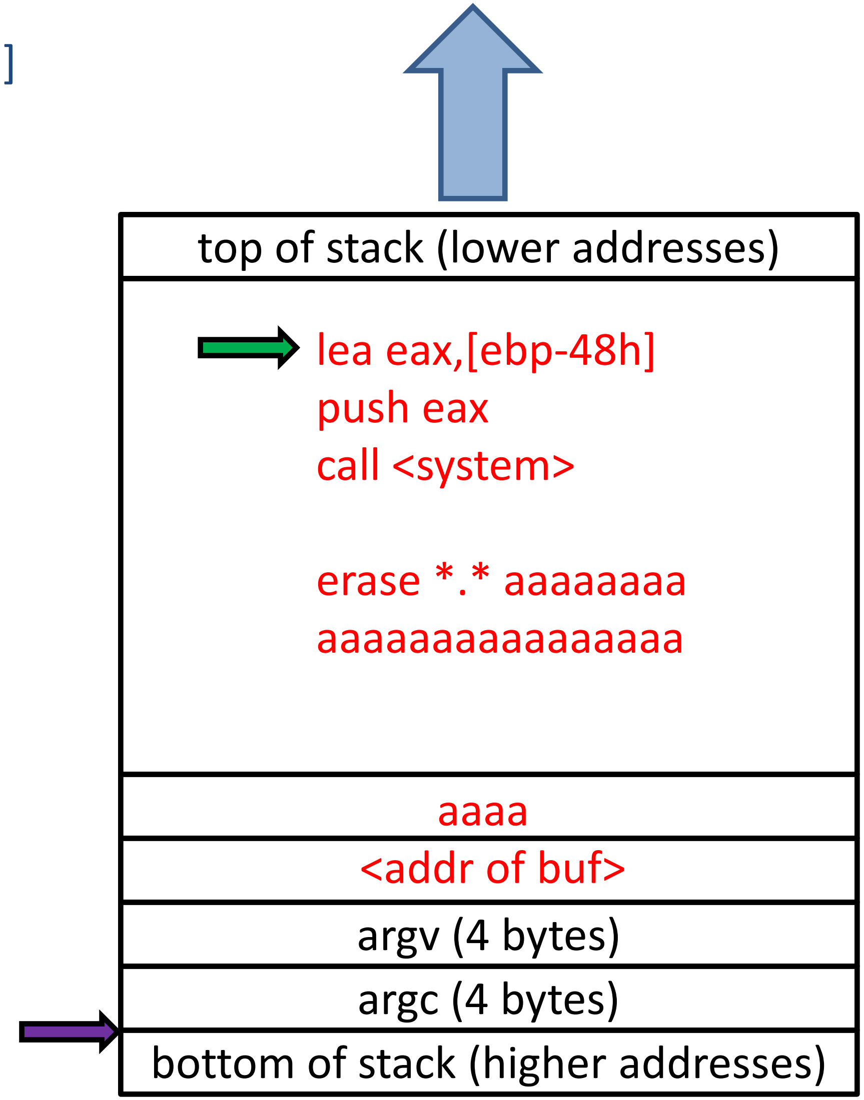
```
void main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return;
}
```



Code-injection Example

8D 45 B8	lea eax,[ebp-48h]
50	push eax
FF 15 BC 82 2F 01	call <system>
65 72 61 73 65 20	.data “erase ”
2A 2E 2A 20	.data “*.* ”
61 (x24)	.data “aaaaa…”
61 61 61 61	.data “aaaa”
30 FB 1F 00	<addr of buf>

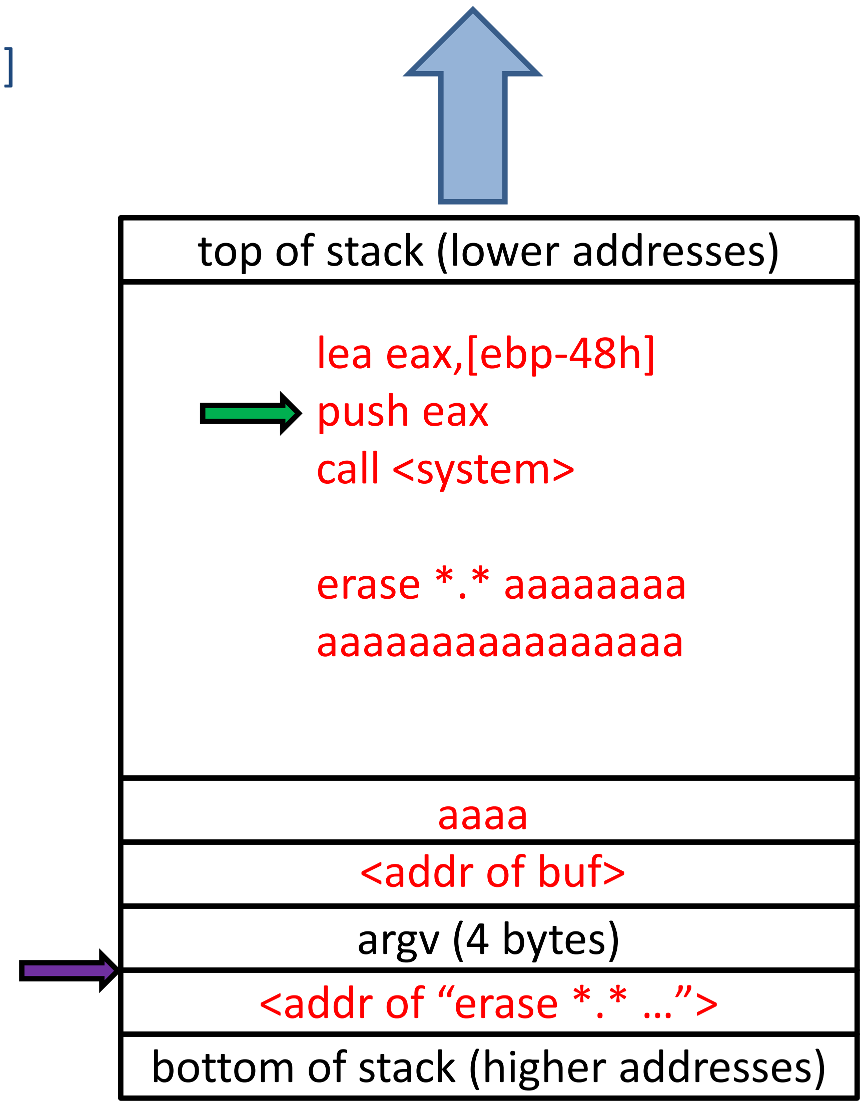
```
void main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return;
}
```



Code-injection Example

8D 45 B8	lea eax,[ebp-48h]
50	push eax
FF 15 BC 82 2F 01	call <system>
65 72 61 73 65 20	.data “erase ”
2A 2E 2A 20	.data “*.* ”
61 (x24)	.data “aaaaa…”
61 61 61 61	.data “aaaa”
30 FB 1F 00	<addr of buf>

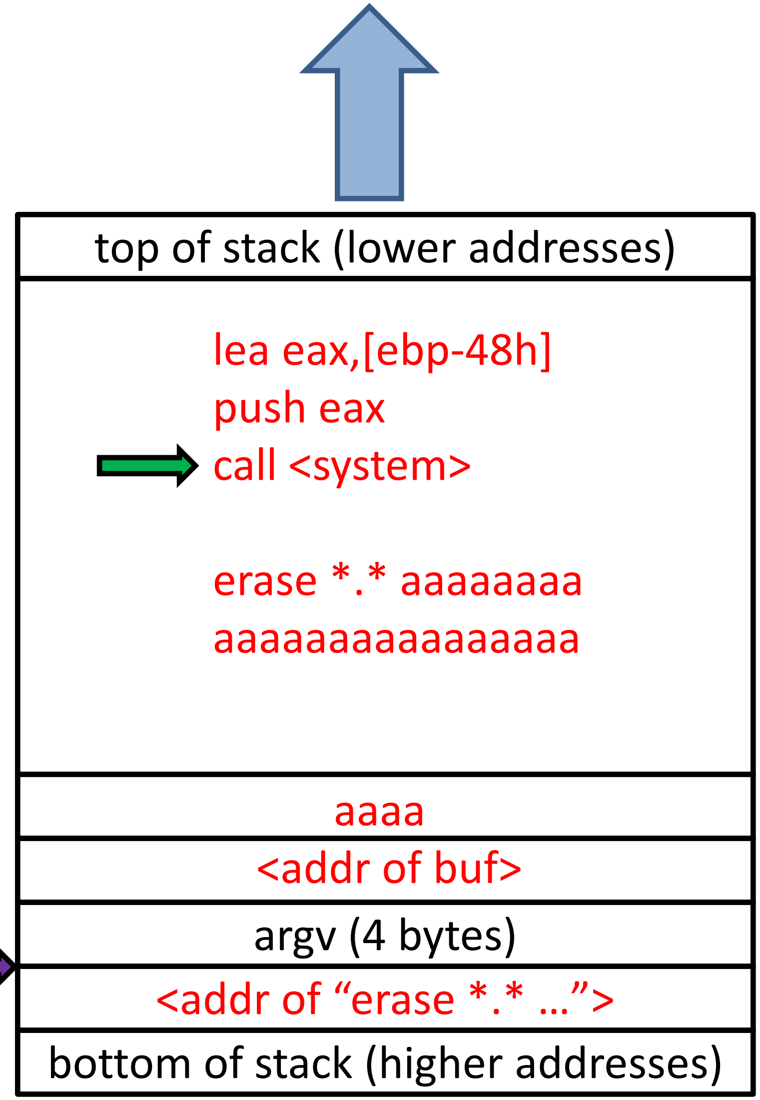
```
void main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return;
}
```



Code-injection Example

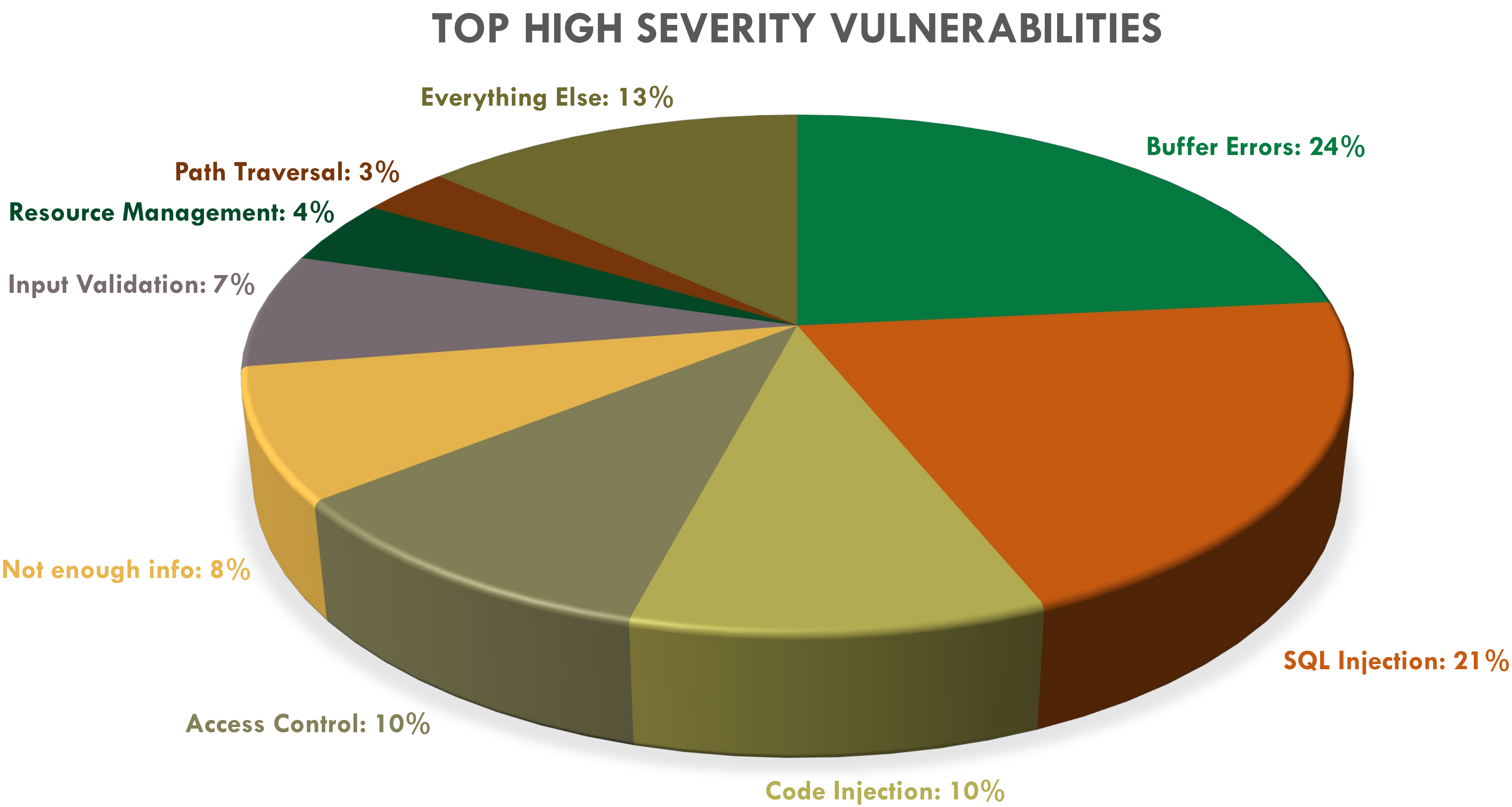
8D 45 B8	lea eax,[ebp-48h]
50	push eax
FF 15 BC 82 2F 01	call <system>
65 72 61 73 65 20	.data “erase ”
2A 2E 2A 20	.data “*.* ”
61 (x24)	.data “aaaaa…”
61 61 61 61	.data “aaaa”
30 FB 1F 00	<addr of buf>

```
void main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return;
}
```



Pernicious Vulnerabilities

[SourceFire Vulnerability Research]



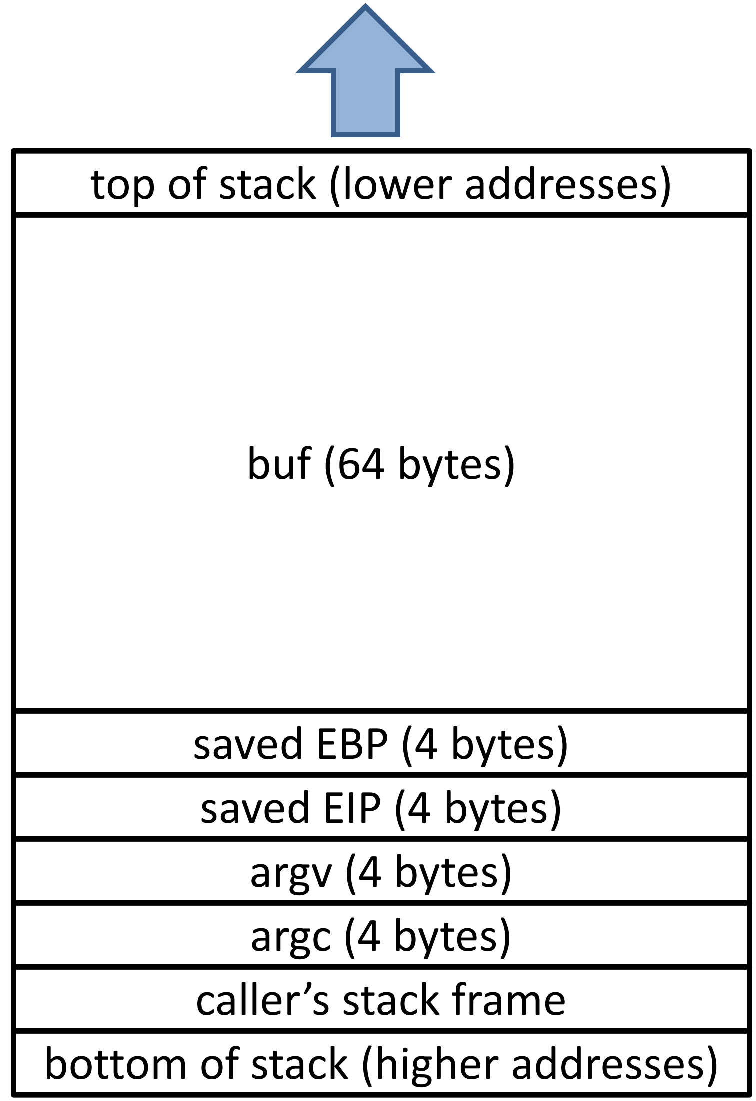
Defense: DEP + ASLR

- Data Execution Prevention (DEP)
  - set stack memory non-executable (hardware-enforced)
- Address Space Layout Randomization (ASLR)
  - randomize locations of libraries on-load
- Counter-attack
  - don’t insert any code onto the stack
  - jump *directly to existing code fragments*
  - called a “code-reuse” attack

ROP Example

61 72 61 73 65 20	.data “erase ”
2A 2E 2A 20	.data “*.* ”
61 (x58)	.data “aaaa…”
BC 82 2F 04	.data <addr1>
61 61 61 61	.data “aaaa”
82 8C 2E 04	.data <addr2>
82 8C 2E 04	.data <addr2>
7F 22 30 04	.data <addr3>

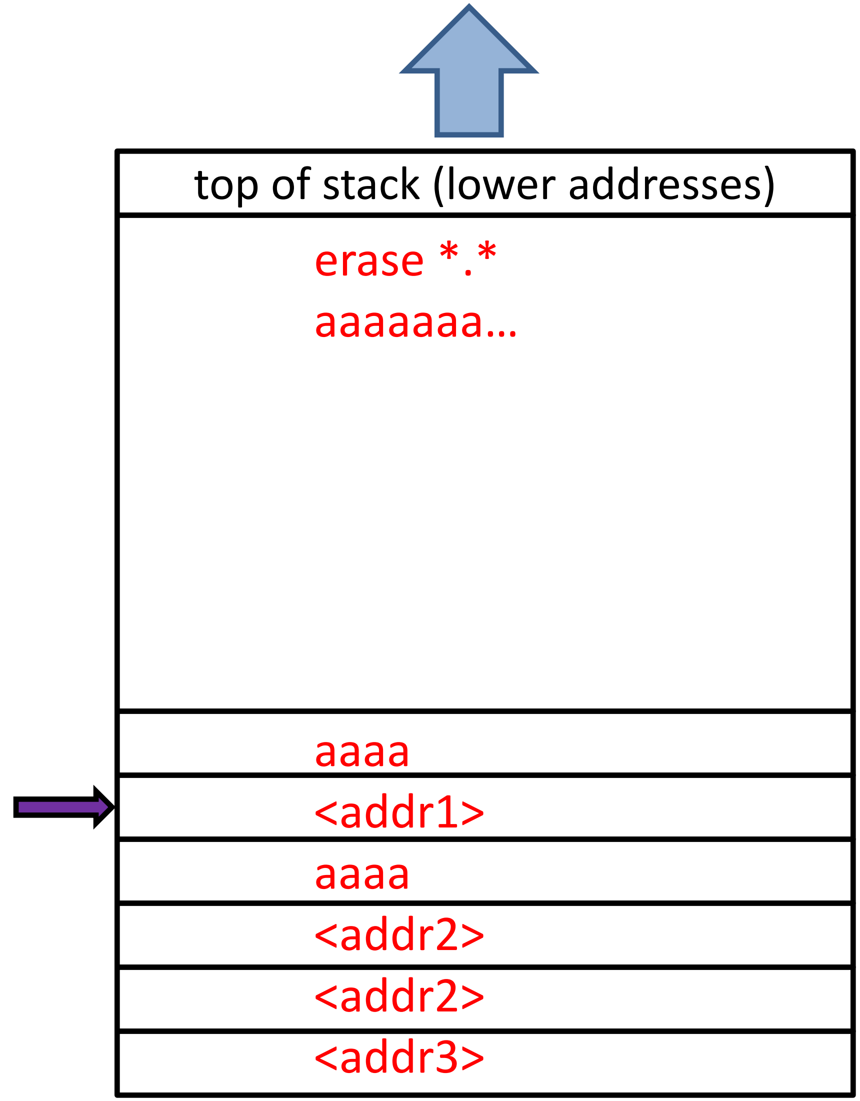
```
void main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return;
}
```



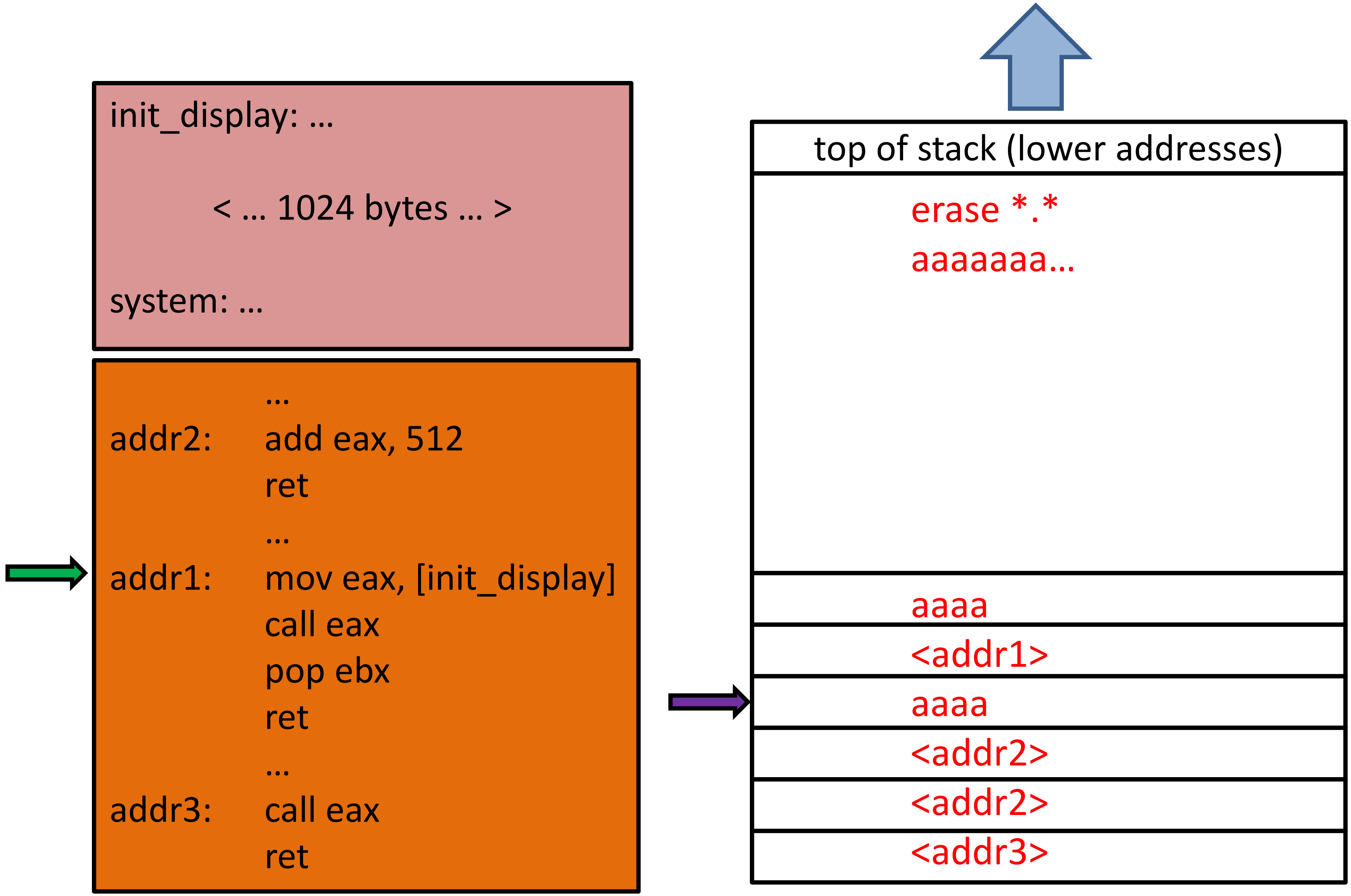
ROP Example

61 72 61 73 65 20	.data “erase ”
2A 2E 2A 20	.data “*.* ”
61 (x58)	.data “aaaa…”
BC 82 2F 04	.data <addr1>
61 61 61 61	.data “aaaa”
82 8C 2E 04	.data <addr2>
82 8C 2E 04	.data <addr2>
7F 22 30 04	.data <addr3>

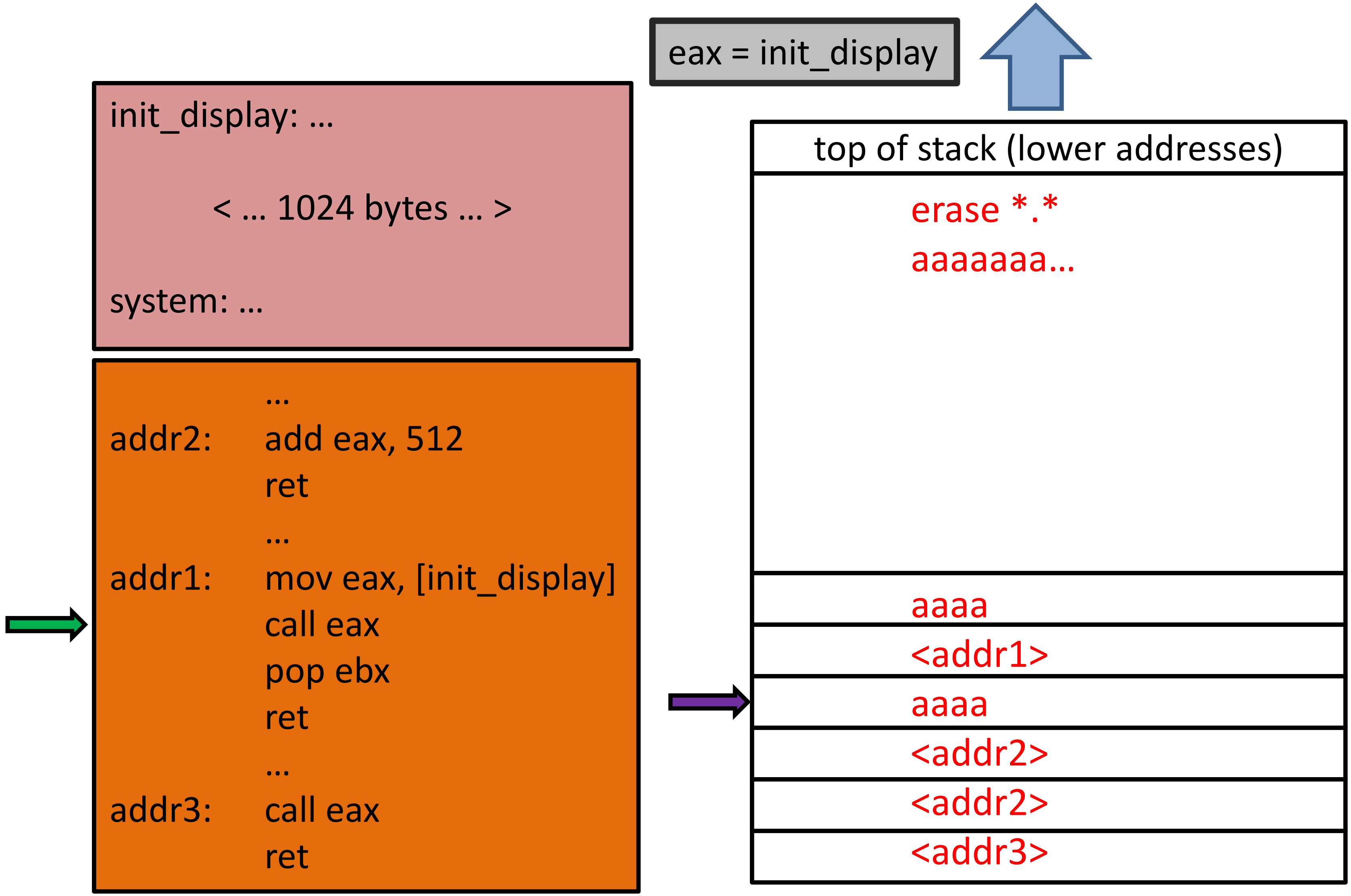
```
void main(int argc, char *argv[])
{
        char buf[64];
        strcpy(buf,argv[1]);
        …
        return;
}
```



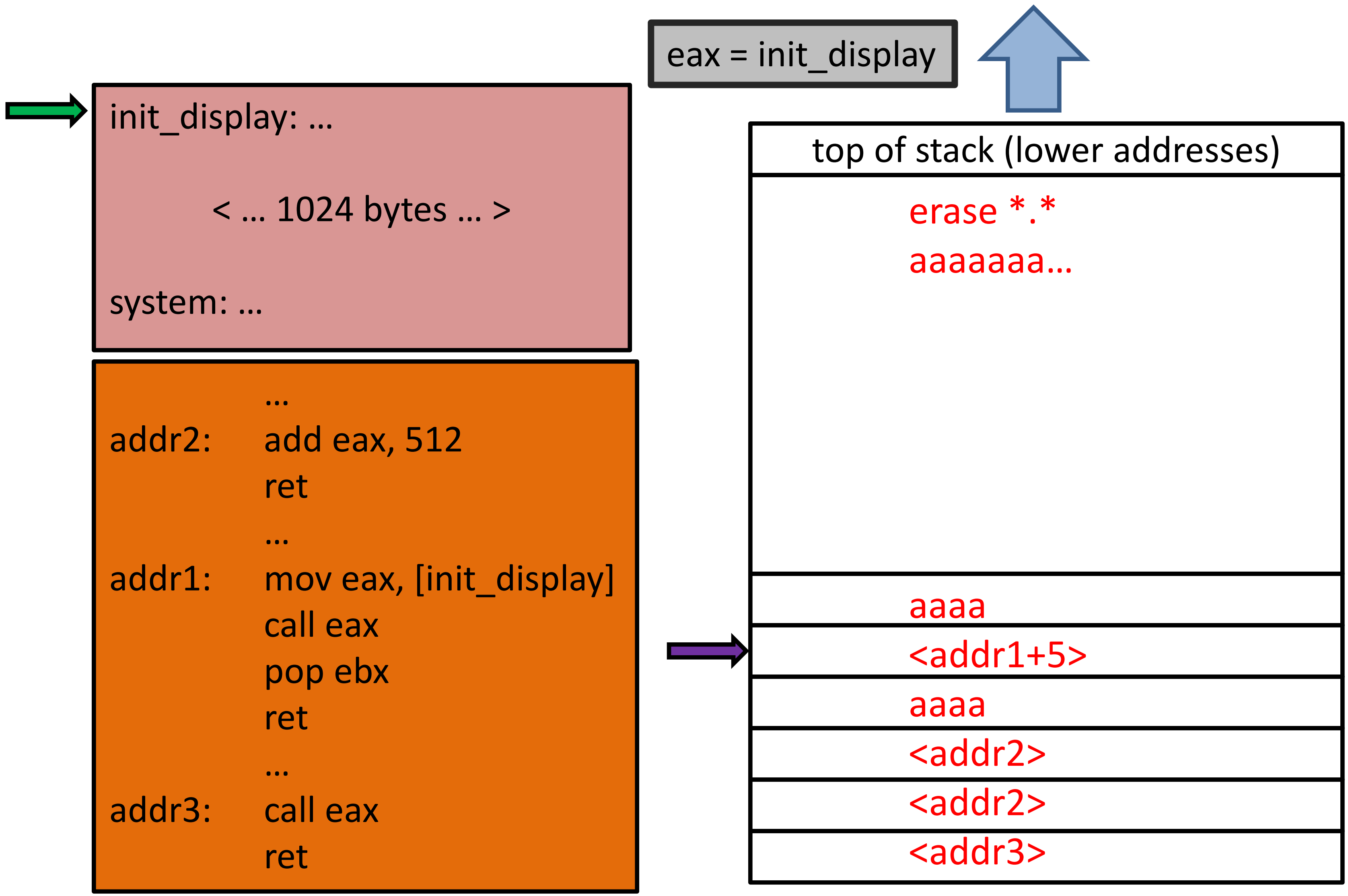
ROP Example


init_display: …
< … 1024 bytes … >
system: …
…
addr2: add eax, 512
ret
…
addr1: mov eax, [init_display]
call eax
pop ebx
ret
…
addr3: call eax
ret
top of stack (lower addresses)
erase *.*
aaaaaaa…
aaaa
<addr1>
aaaa
<addr2>
<addr2>
<addr3>

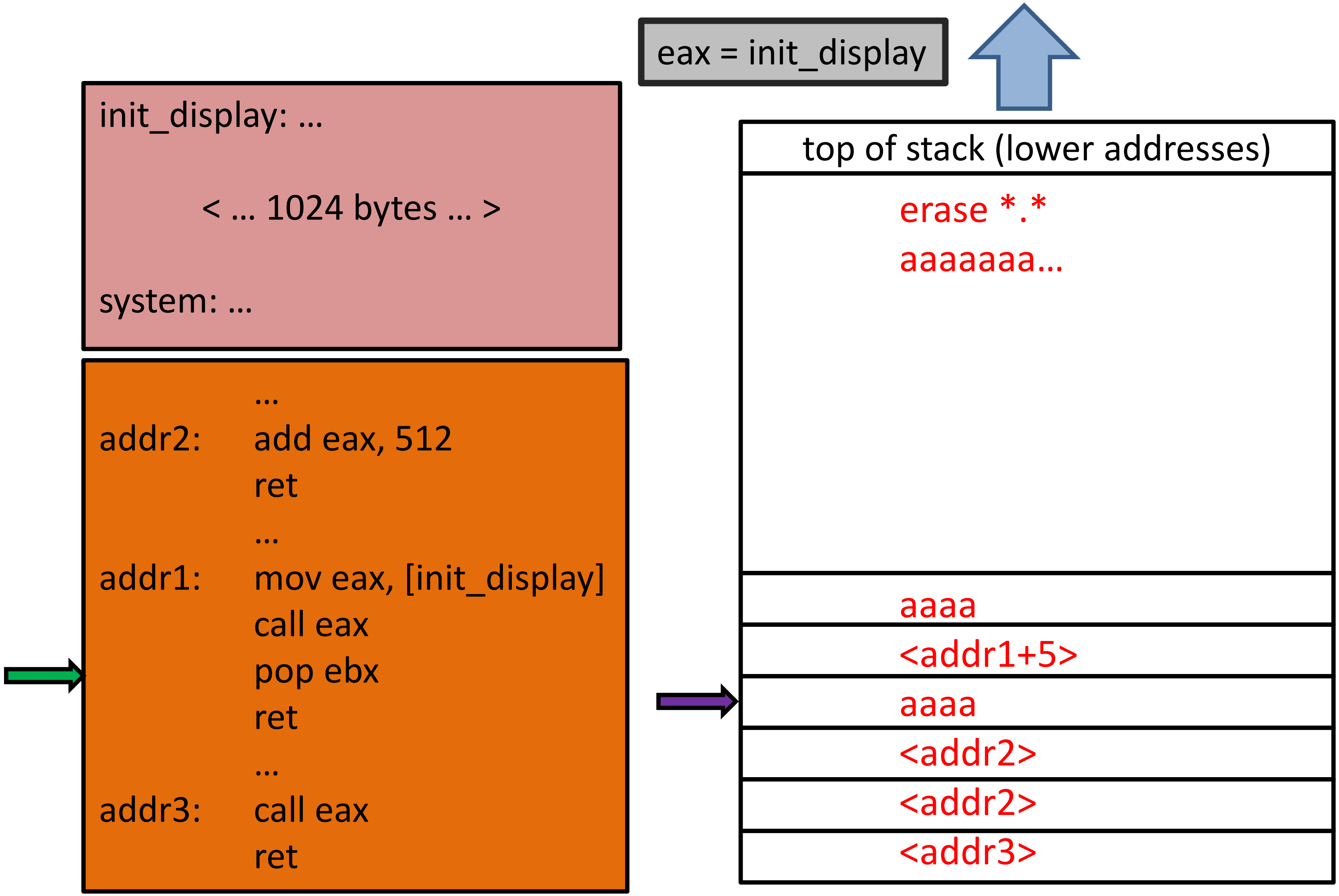
ROP Example


eax = init_display
init_display: …
< … 1024 bytes … >
system: …
…
addr2: add eax, 512
ret
…
addr1: mov eax, [init_display]
call eax
pop ebx
ret
…
addr3: call eax
ret
top of stack (lower addresses)
erase *.*
aaaaaaa…
aaaa
<addr1>
aaaa
<addr2>
<addr2>
<addr3>

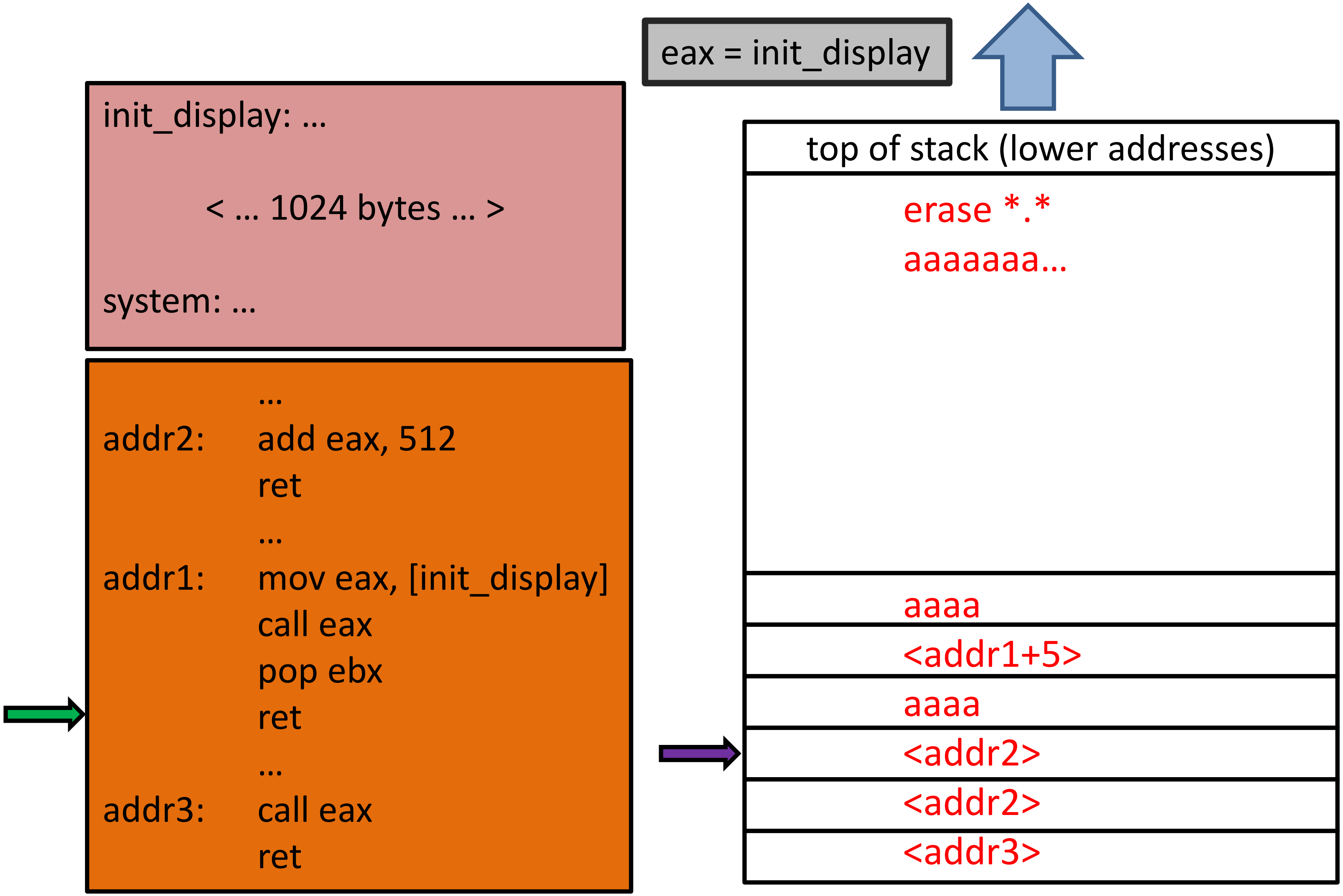
ROP Example


eax = init_display
init_display: …
< … 1024 bytes … >
system: …
…
addr2: add eax, 512
ret
…
addr1: mov eax, [init_display]
call eax
pop ebx
ret
…
addr3: call eax
ret
top of stack (lower addresses)
erase *.*
aaaaaaa…
aaaa
<addr1+5>
aaaa
<addr2>
<addr2>
<addr3>

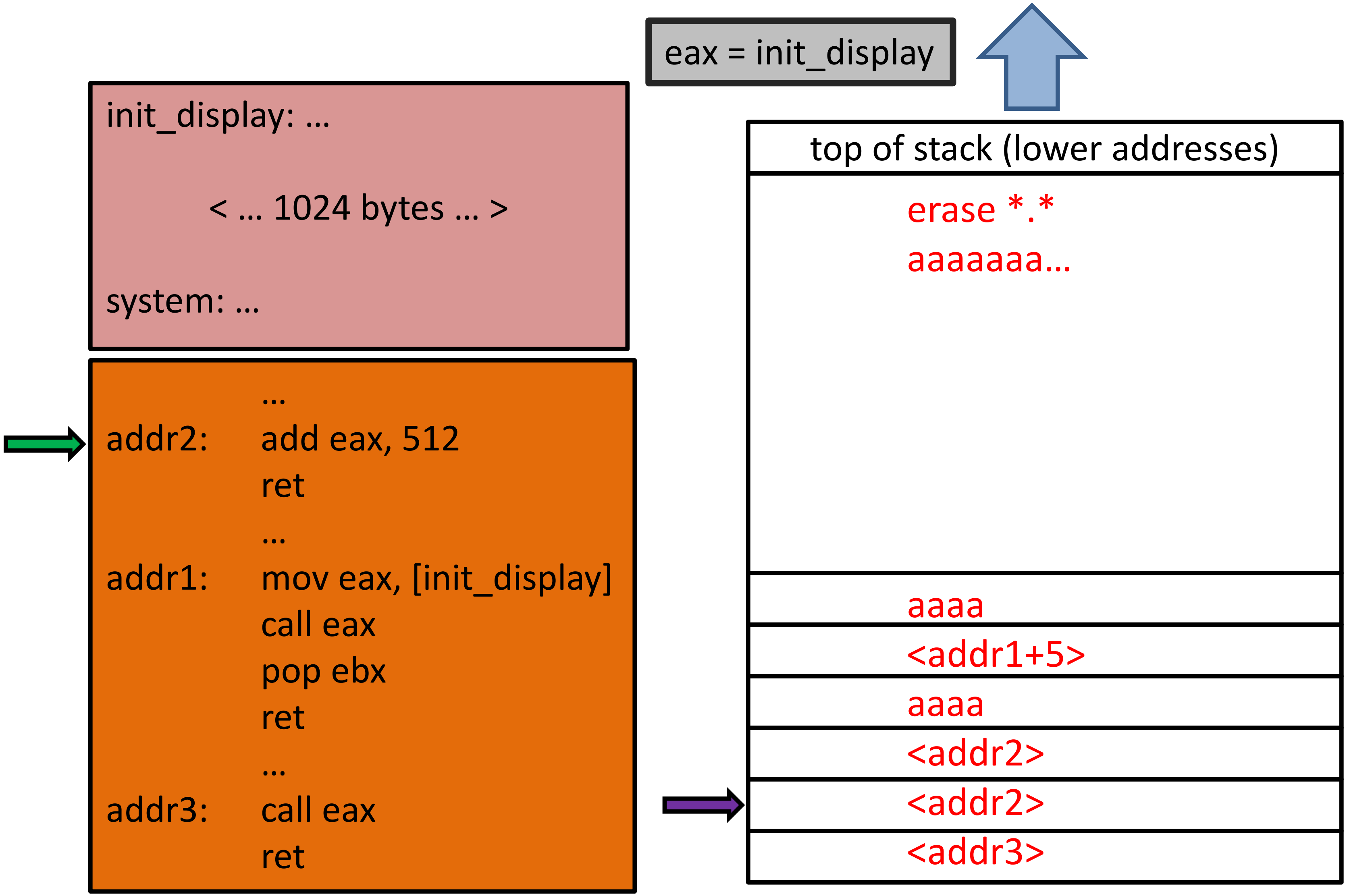
ROP Example


eax = init_display
init_display: …
< … 1024 bytes … >
system: …
…
addr2: add eax, 512
ret
…
addr1: mov eax, [init_display]
call eax
pop ebx
ret
…
addr3: call eax
ret
top of stack (lower addresses)
erase *.*
aaaaaaa…
aaaa
<addr1+5>
aaaa
<addr2>
<addr2>
<addr3>

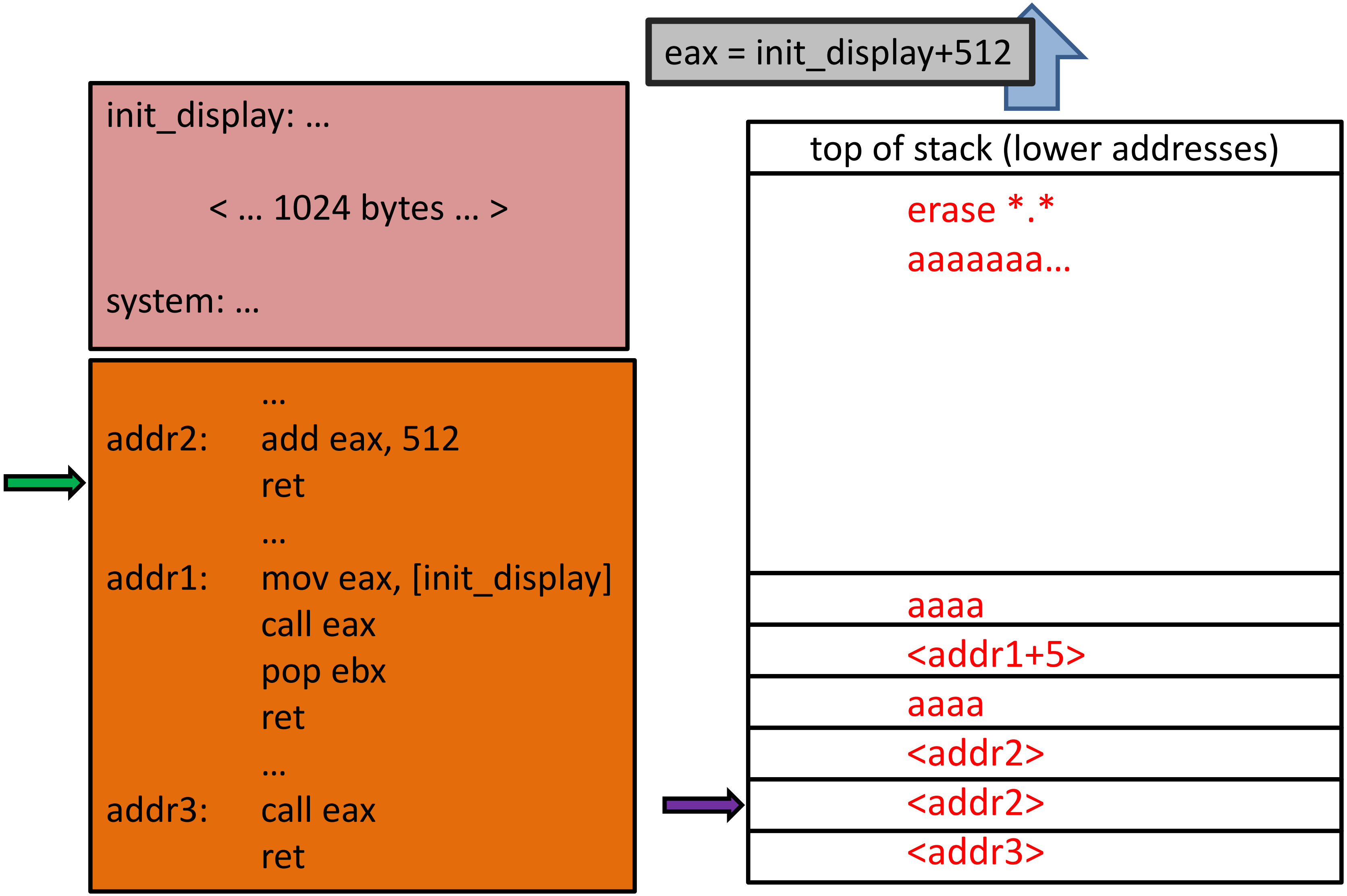
ROP Example


eax = init_display
init_display: …
< … 1024 bytes … >
system: …
…
addr2: add eax, 512
ret
…
addr1: mov eax, [init_display]
call eax
pop ebx
ret
…
addr3: call eax
ret
top of stack (lower addresses)
erase *.*
aaaaaaa…
aaaa
<addr1+5>
aaaa
<addr2>
<addr2>
<addr3>

ROP Example

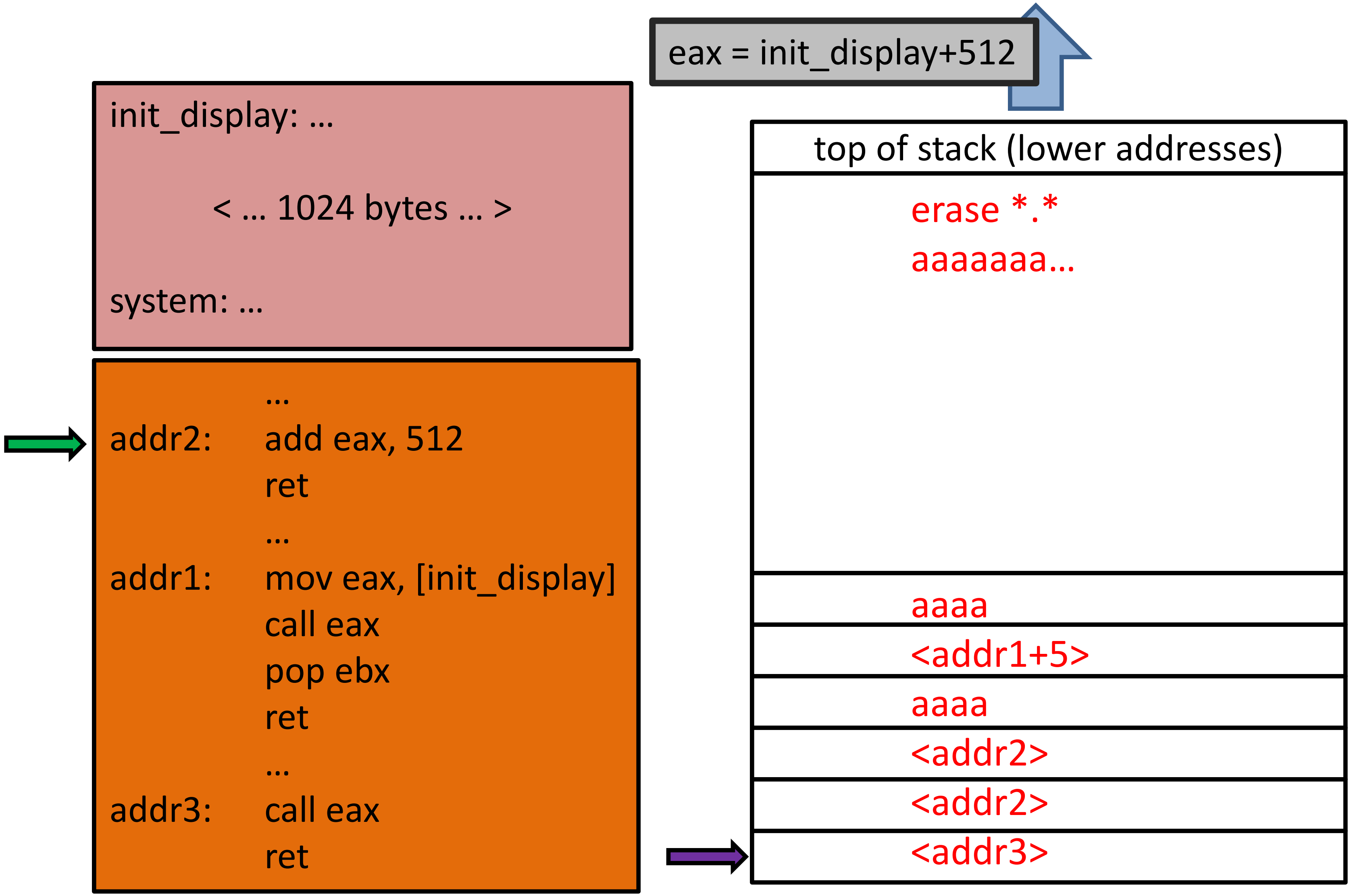

eax = init_display
init_display: …
< … 1024 bytes … >
system: …
…
addr2: add eax, 512
ret
…
addr1: mov eax, [init_display]
call eax
pop ebx
ret
…
addr3: call eax
ret
top of stack (lower addresses)
erase *.*
aaaaaaa…
aaaa
<addr1+5>
aaaa
<addr2>
<addr2>
<addr3>

ROP Example

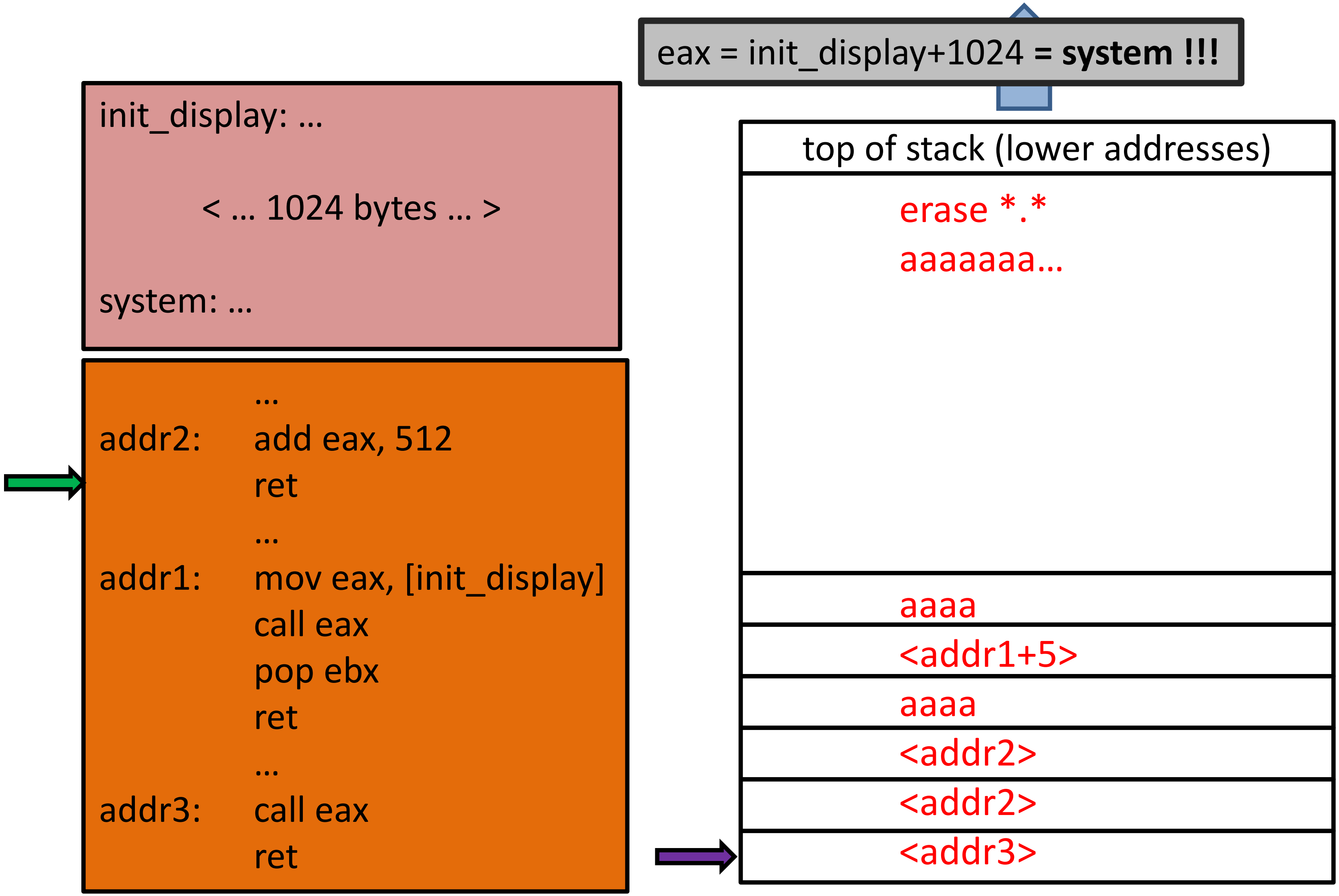


eax = init_display+512
init_display: …
< … 1024 bytes … >
system: …
…
addr2: add eax, 512
ret
…
addr1: mov eax, [init_display]
call eax
pop ebx
ret
…
addr3: call eax
ret
top of stack (lower addresses)
erase *.*
aaaaaaa…
aaaa
<addr1+5>
aaaa
<addr2>
<addr2>
<addr3>

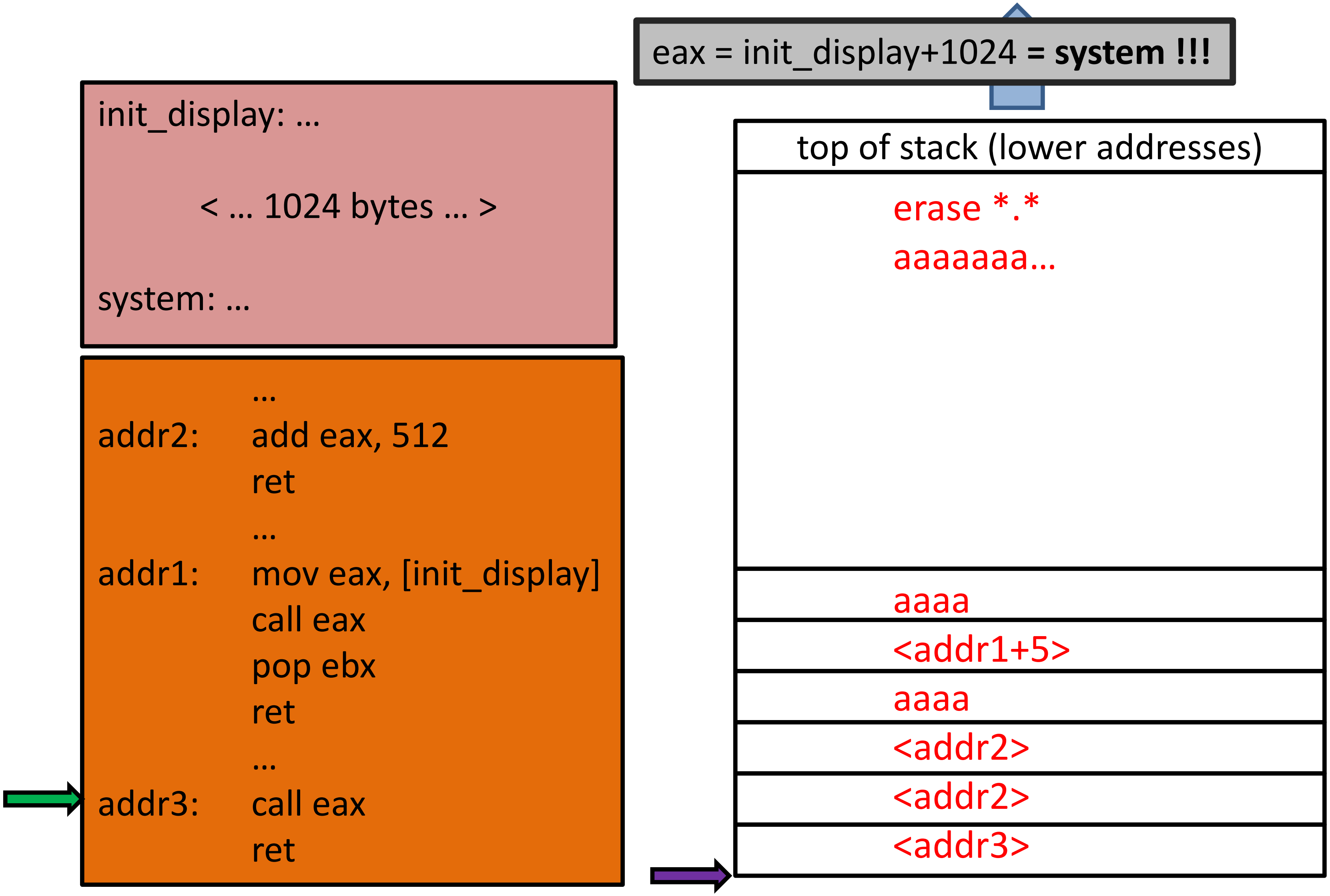
ROP Example


eax = init_display+512
init_display: …
< … 1024 bytes … >
system: …
…
addr2: add eax, 512
ret
…
addr1: mov eax, [init_display]
call eax
pop ebx
ret
…
addr3: call eax
ret
top of stack (lower addresses)
erase *.*
aaaaaaa…
aaaa
<addr1+5>
aaaa
<addr2>
<addr2>
<addr3>

ROP Example

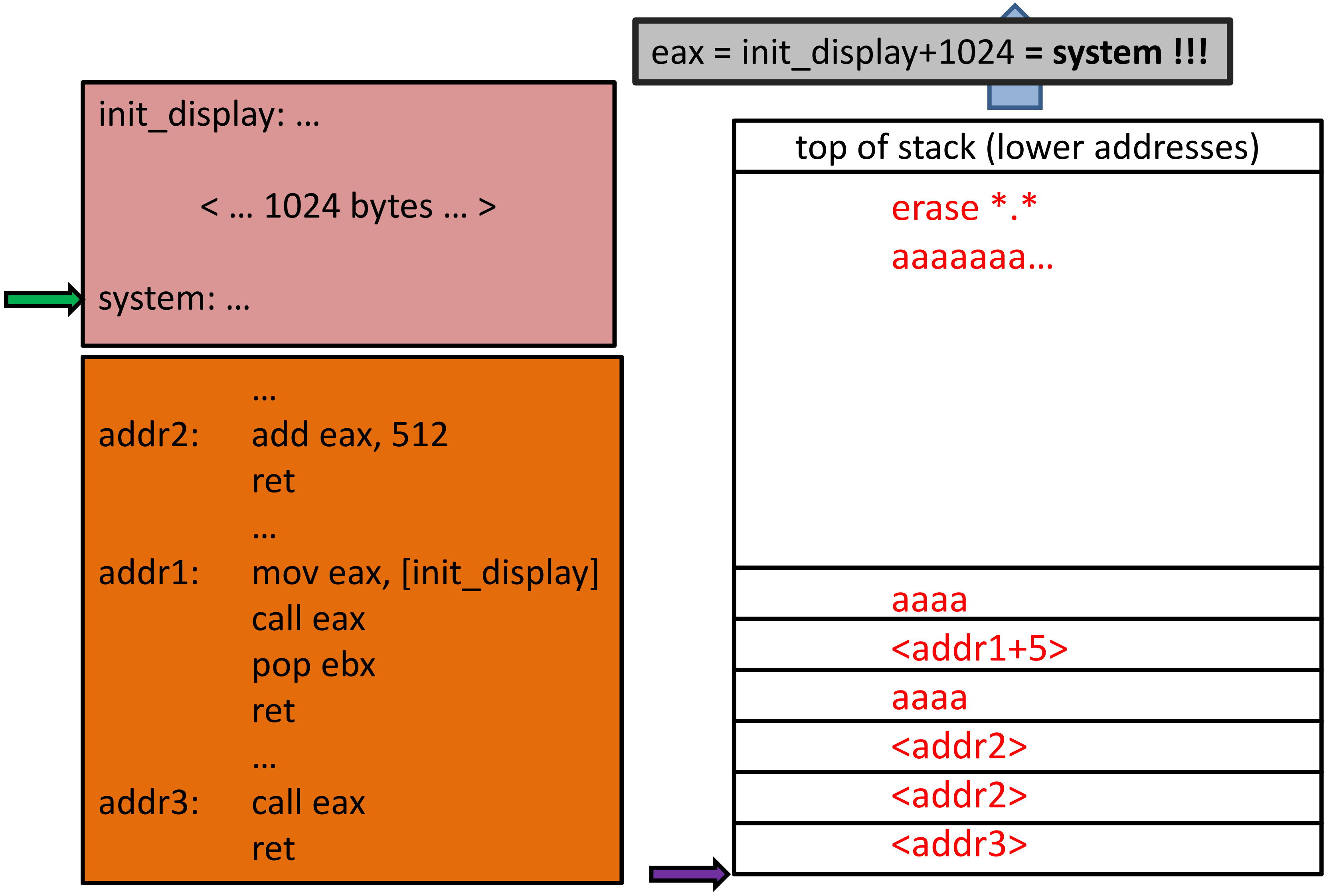

eax = init_display+1024 = **system !!!**
init_display: …
< … 1024 bytes … >
system: …
…
addr2: add eax, 512
ret
…
addr1: mov eax, [init_display]
call eax
pop ebx
ret
…
addr3: call eax
ret
top of stack (lower addresses)
erase *.*
aaaaaaa…
aaaa
<addr1+5>
aaaa
<addr2>
<addr2>
<addr3>

ROP Example



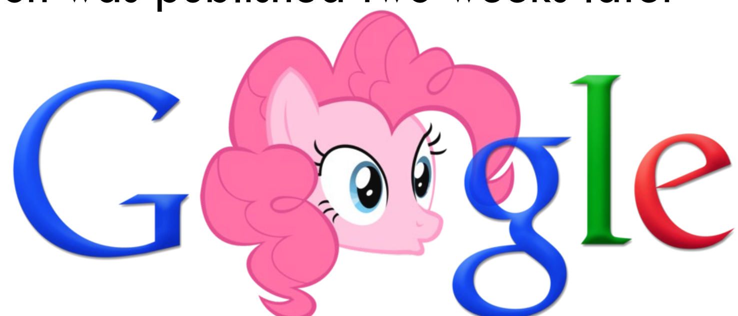
eax = init_display+1024 = **system !!!**
init_display: …
< … 1024 bytes … >
system: …
…
addr2: add eax, 512
ret
…
addr1: mov eax, [init_display]
call eax
pop ebx
ret
…
addr3: call eax
ret
top of stack (lower addresses)
erase *.*
aaaaaaa…
aaaa
<addr1+5>
aaaa
<addr2>
<addr2>
<addr3>

ROP Example

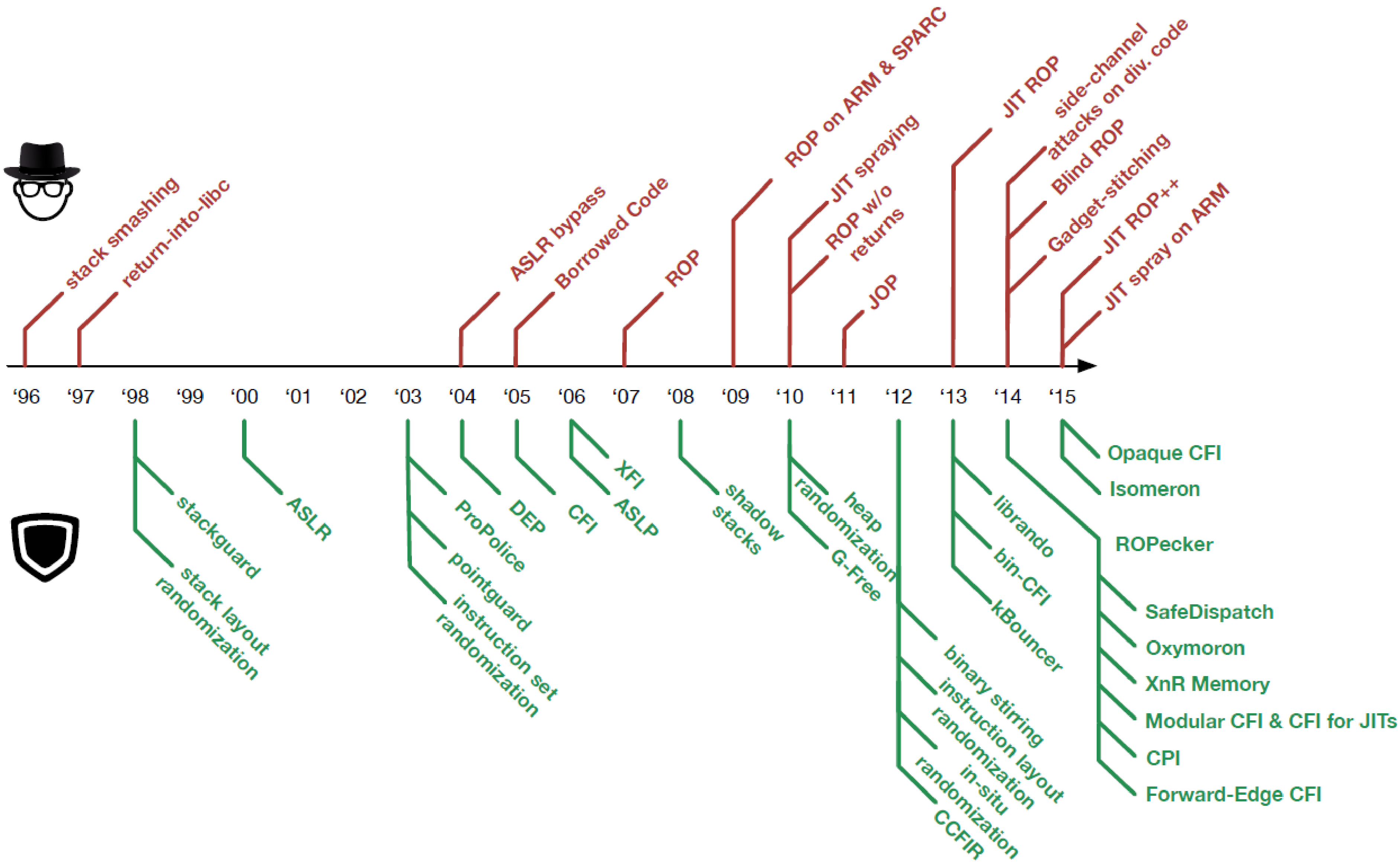

eax = init_display+1024 = **system !!!**
init_display: …
< … 1024 bytes … >
system: …
…
addr2: add eax, 512
ret
…
addr1: mov eax, [init_display]
call eax
pop ebx
ret
…
addr3: call eax
ret
top of stack (lower addresses)
erase *.*
aaaaaaa…
aaaa
<addr1+5>
aaaa
<addr2>
<addr2>
<addr3>

Battling Code-reuse Attacks

- Microsoft's 2012 BlueHat Competition
  - Focused on RoP Mitigation
  - $260,000 total for top three solutions
    - Successful attack against 2nd place solution was published two weeks later
- Google Pwnium Competition
  - Hacker Pinkie Pie paid $60K for Chrome RoP exploit
  - Google fixes the exploit
  - Five months later, Pinkie Pie finds a new RoP exploit in the fixed Chrome, gets paid another $60K
  - Google fixes the 2nd exploit
  - Five months later, Pinkie Pie finds a yet another (partial) exploit, gets paid another $40K

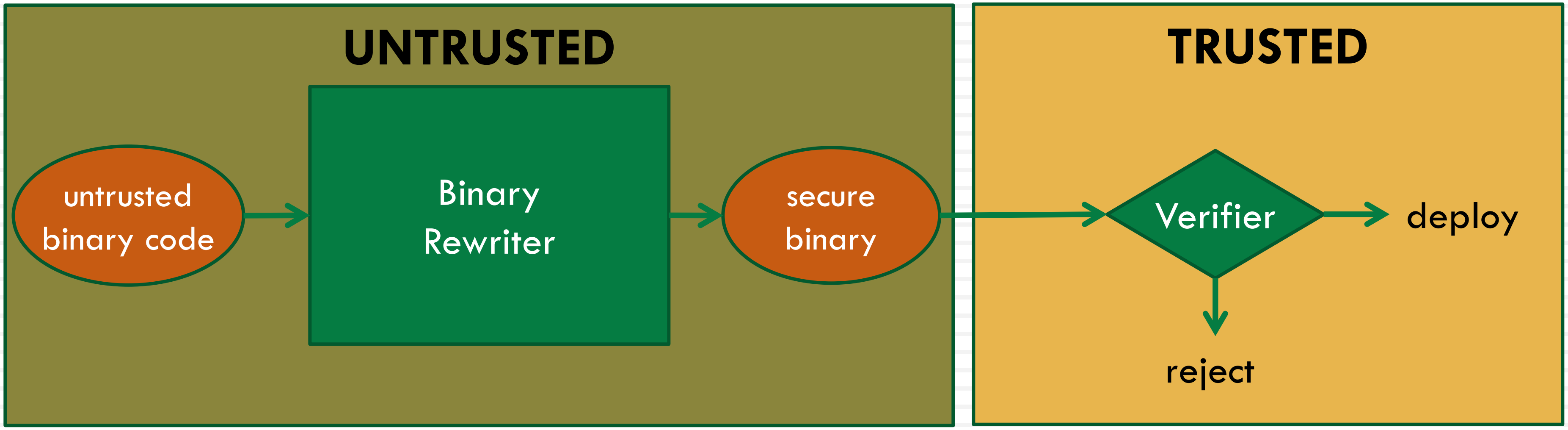


Code-reuse Conflict Timeline



My Research: Security Retrofitting

Secure commodity software AFTER it is compiled and distributed, by automatically modifying it at the binary level.



Advantages

- No need to get code-producer cooperation
- No need to customize the OS/VM
- No custom hardware needed (expensive & slow)
- Not limited to any particular source language or tool chain
- Can enforce consumer-specific policies
- Maintainable across version updates (just re-apply rewriter to newly released version)
- Rewriter remains untrusted, so can outsource that task to an untrusted third party!
  - Local, trusted verifier checks results

Challenges

- Software is in purely binary form
  - no source, no debug info, no disassembly
- Diverse origins
  - various source languages, compilers, tools, …
- Code-producers are uncooperative
  - unwilling to recompile with special compiler
  - unwilling to add/remove features
  - no compliance with any coding standard
- Highly complex binary structure
  - target real-world APIs (e.g., hundreds of thousands of Windows system dll's and drivers)
  - multi-threaded, multi-process
  - event-driven (callbacks), dynamically linked (runtime loading)
  - heavily optimized (binary code & data arbitrarily interleaved)

Three Major Advances

1) Heuristic-free & Machine Learning-based Binary Disassembly
   - automatically recovers high-level program structure from binary software product
   - Superset Disassembly (NDSS'18): recover a *superset* of the control-flow graph
   - Finding the Undecidable Path (PAKDD'14): Optimize CFG via machine learning
2) Native Code Instrumentation
   - method of automatically in-lining extra security checks into untrusted programs
   - Wartell, Mohan, Hamlen, and Lin. *Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code*. CCS 2012.
3) Formal, Automated, Machine-validation
   - automatically PROVES (mathematically) that retrofitted software is immune to certain classes of attacks
   - Wartell, Mohan, Hamlen, and Lin. *Securing Untrusted Code via Compiler-Agnostic Binary Rewriting*. ACSAC 2012.

First Step: Disassembly


```
FF E0 5B 5D C3 0F
88 52 0F 84 EC 8B
```

- Disassemble this hex sequence
  - Turns out x86 disassembly is an undecidable problem!

Valid Disassembly	
FF E0	jmp eax
5B	pop ebx
5D	pop ebp
C3	retn
0F 88 52 0F 84 EC	jcc
8B …	mov

Valid Disassembly	
FF E0	jmp eax
5B	pop ebx
5D	pop ebp
C3	retn
0F	db (1)
88 52 0F 84 EC	mov
8B …	mov

Valid Disassembly	
FF E0	jmp eax
5B	pop ebx
5D	pop ebp
C3	retn
0F 88	db (2)
52	push edx
0F 84 EC 8B …	jcc

Disassembly Intractability

- Even the best reverse-engineering tools cannot reliably disassemble even standard COTS products
- Example: IDA Professional Disassembler (Hex-rays)

Program Name	Disassembly Errors
Microsoft Foundation Class Lib (mfc42.dll)	1216
Media Player (mplayerc.exe)	474
Avant Web Browser (RevelationClient.exe)	36
VMWare (vmware.exe)	183

Innovation: Superset Disassembly

`Byte Sequence: FF E0 5B 5D C3 0F 88 B0 50 FF FF 8B`

● **Disassembled** ✖ **Invalid**

	Hex
●	FF
●	E0
●	5B
●	5D
●	C3
●	0F
●	88
● ✖	B0
	50
✖	FF
	FF
●	8B

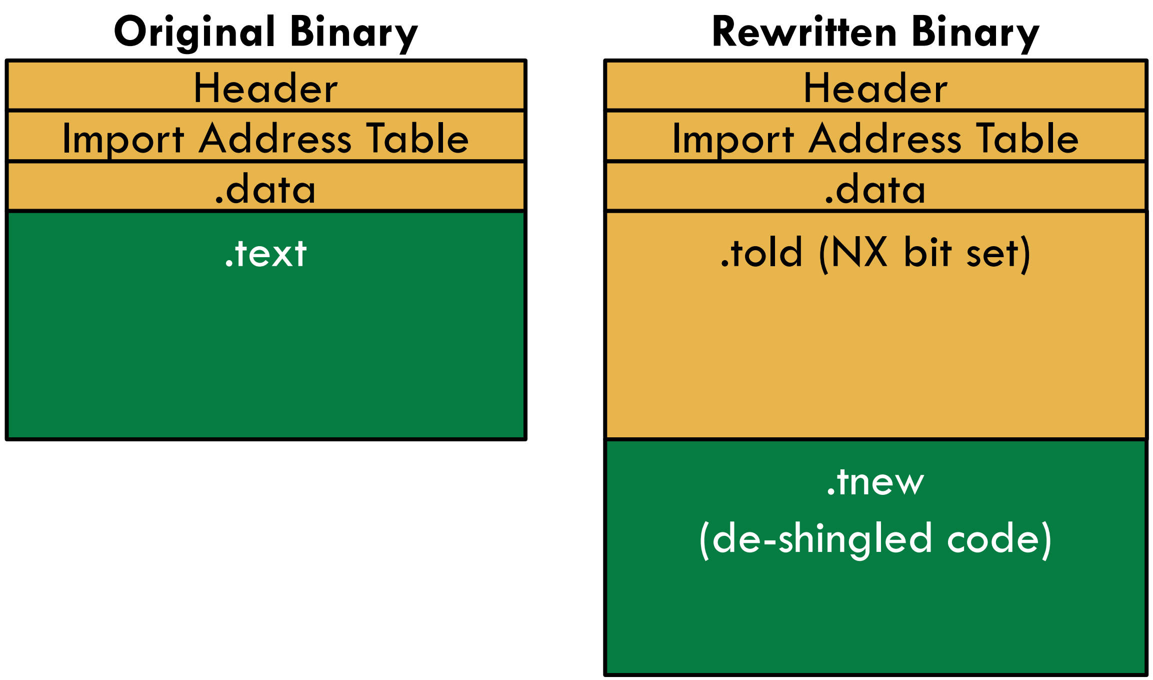
Included Disassembly
jmp eax
pop
L1: pop
retn
jcc
L2: mov
loopne
jmp L1
mov
jmp L2

Problem: Pointers

- We just rearranged everything. Pointers will all point to the wrong places.
  - can't reliably identify pointer data in a sea of unlabeled bytes
- Two kinds of relevant pointers:
  - pointers to static data bytes among the code bytes
  - pointers to code (e.g., method dispatch tables)

Preserving Static Data Pointers

- Put the de-shingled code in a NEW code segment.
  - Set it execute-only (non-writable)
- Leave the original .text section
  - Set it read/write-only (non-execute)



Preserving Code Pointers

- Almost half of all jump instructions in real x86 binaries *compute their destinations at runtime*.
  - Exercise: Why? Examples?
  - …
  - …
  - …
- Must ensure these jumps target *new code locations* instead of old.
  - impossible to statically predict their destinations

Preserving Code Pointers

- Almost half of all jump instructions in real x86 binaries *compute their destinations at runtime*.
  - all method calls (read method dispatch table)
  - all function returns (read stack)
  - almost all API calls (read linker tables)
  - pointer encryption/decryption logic for security
- Must ensure these jumps target *new code locations* instead of old.
  - impossible to statically predict their destinations

Solution: Control-flow Patching

- Create a lookup table that maps old code addresses to new ones at runtime.
- Add instructions that consult the lookup table before any computed jump.

Original	→	Rewritten
jump eax		jump table[eax]

Optimizing

- With these three tricks we can successfully transform (most) real-world COTS binaries even without knowing how they work or what they do!
  - de-shingling disassembly
  - static data preservation
  - control-flow patching
- Limitations
  - runtime code modification conservatively disallowed
  - computing data pointers from code pointers breaks
  - These are <u>compatibility</u> limitations *not security limitations.*
- But it's prohibitively inefficient (increases code size ~700%)
  - need to optimize the approach

Optimization Philosophy

1. If the optimization fails, we might get broken code but *never* unsafe code.
2. The optimizations only need to work for non-malicious, non-vulnerable code fragments.
   - If the code fragment is malicious or vulnerable, we don't want to preserve it!

Optimization #1: De-shingling

- Lots of extra overlapping information
  - Can we prune our disassembly tree?

	Hex	Path 1
●	FF	jmp eax
●	E0	
●	5B	pop
●	5D	L1: pop
●	C3	retn
●	0F	jcc
●	88	
✖	B0	
	50	
✖	FF	
	FF	
●	8B	L2: mov

Machine learning-based Disassembler

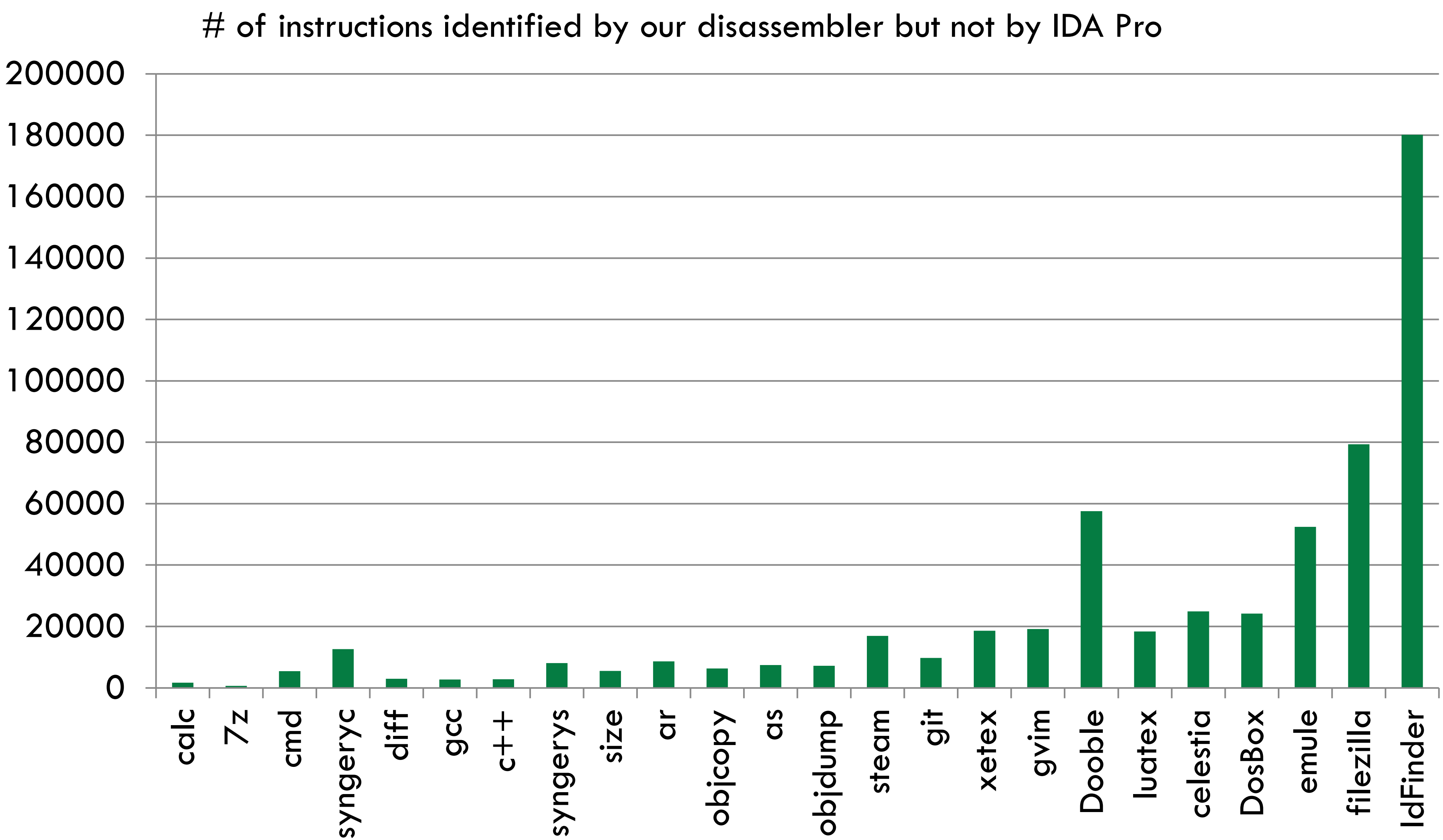
- Insight: Distinguishing real code bytes from data bytes is a "noisy word segmentation problem".
  - Word segmentation: Given a stream of symbols, partition them into words that are contextually sensible. [Teahan, 2000]
  - Noisy word segmentation: Some symbols are noise (data).
- Machine Learning based disassembler
  - based on *k*th-order Markov model
  - Estimate the probability of the sequence B:

$$p(B|M_\alpha) = -\log \prod_{i=1}^{|B|} p(b_i|b_{i-k}^{i-1}, M_\alpha)$$

Wartell, Zhou, Hamlen, Kantarcioglu. "Shingled Graph Disassembly: Finding the Undecidable Path." PAKDD 2014.

Wartell, Zhou, Hamlen, Kantarcioglu, and Thuraisingham. "Differentiating code from data in x86 binaries." *ECML/PKDD* 2011.

Disassembler Stats



PPM Disassembly Stats

	PPM Disassembler		
	False Negative	False Positive	Accuracy
7zFM	0	0	100%
notepad	0	0	100%
DosBox	0	0	100%
WinRAR	0	39	99.982%
mulberry	0	0	100%
scummvm	0	0	100%
emule	0	117	99.988%
Mfc42	0	47	99.987%
mplayerc	0	307	99.963%
revClient	0	71	99.893%
vmware	0	45	99.988%

Optimization #2: Lookup Table Compression

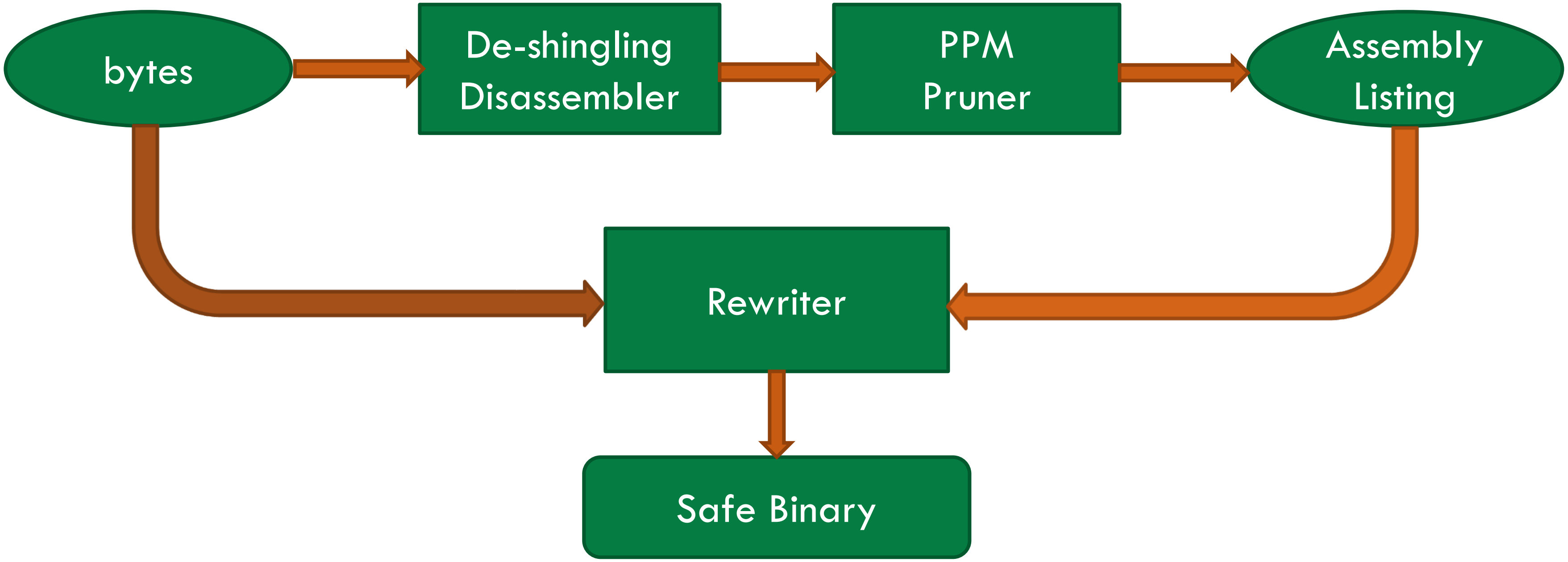
- Idea: Overwrite the old code bytes with the lookup table.
  - PPM disassembler identifies most code bytes
  - Also identifies subset that are possible computed jump destinations.
  - Overwrite those destinations with our lookup table.

Original
call eax

Rewritten
cmp [eax], 0xF4
cmovz eax, [eax+1]
call eax

Applications of our Rewriter

- Three Applications
  - Binary randomization for RoP Defense (STIR)
  - Opaque Control-Flow Integrity (O-CFI)
  - Machine-certified Software Fault Isolation (Reins)



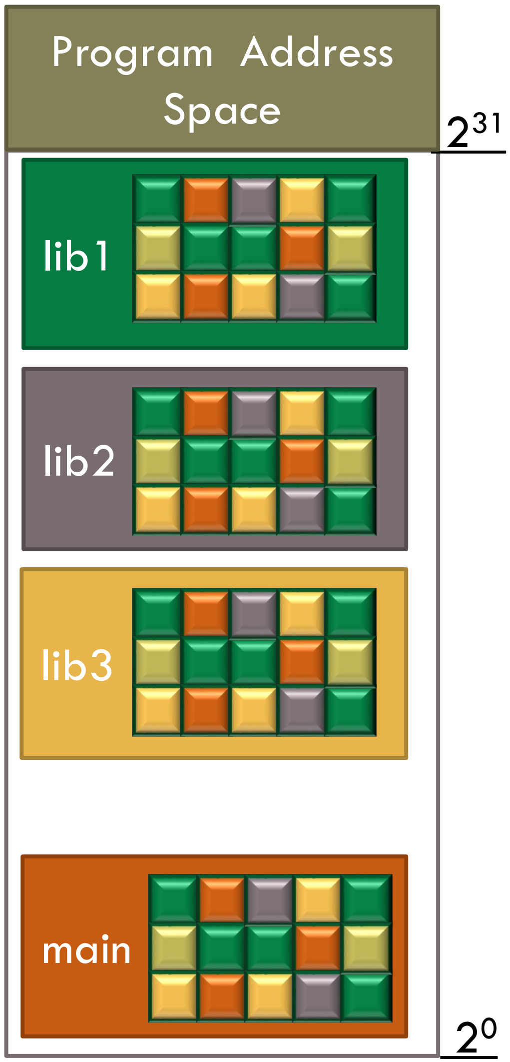
RoP Defense Strategy

- RoP is one example of a broad class of attacks that require attackers to know or predict the location of binary features

<u>Defense Goal</u>

Frustrate such attacks by randomizing the feature space

STIR – Self-Transforming Instruction Relocation
O-CFI – Opaque Control-Flow Integrity

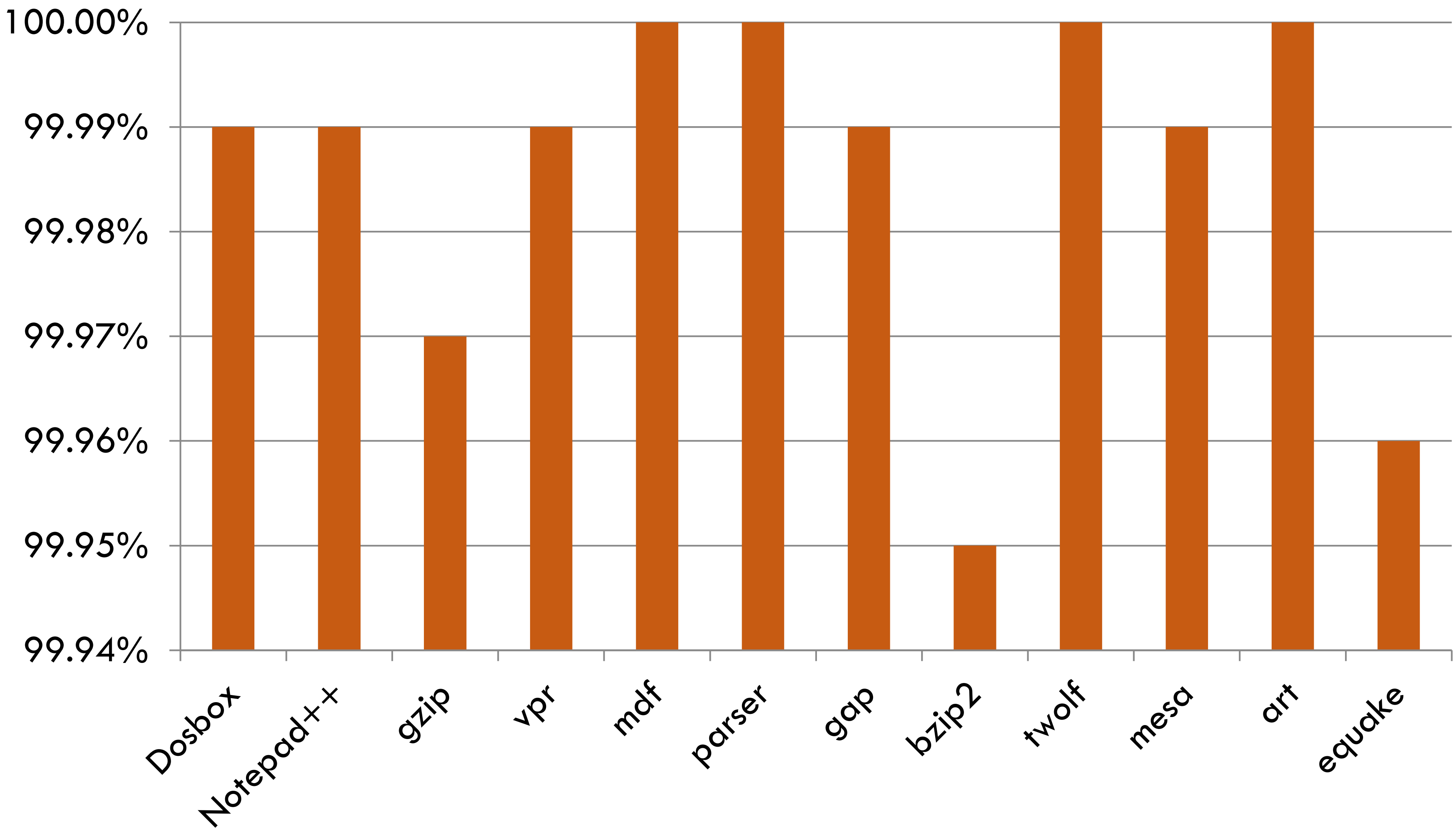


- Randomly reorder the program's internal layout every time the program loads
  - Attacker cannot reliably locate code addresses for code-reuse attacks
  - Astronomically low chance of attack success
  - Exact attack probability is *mathematically computable* as an entropy calculation

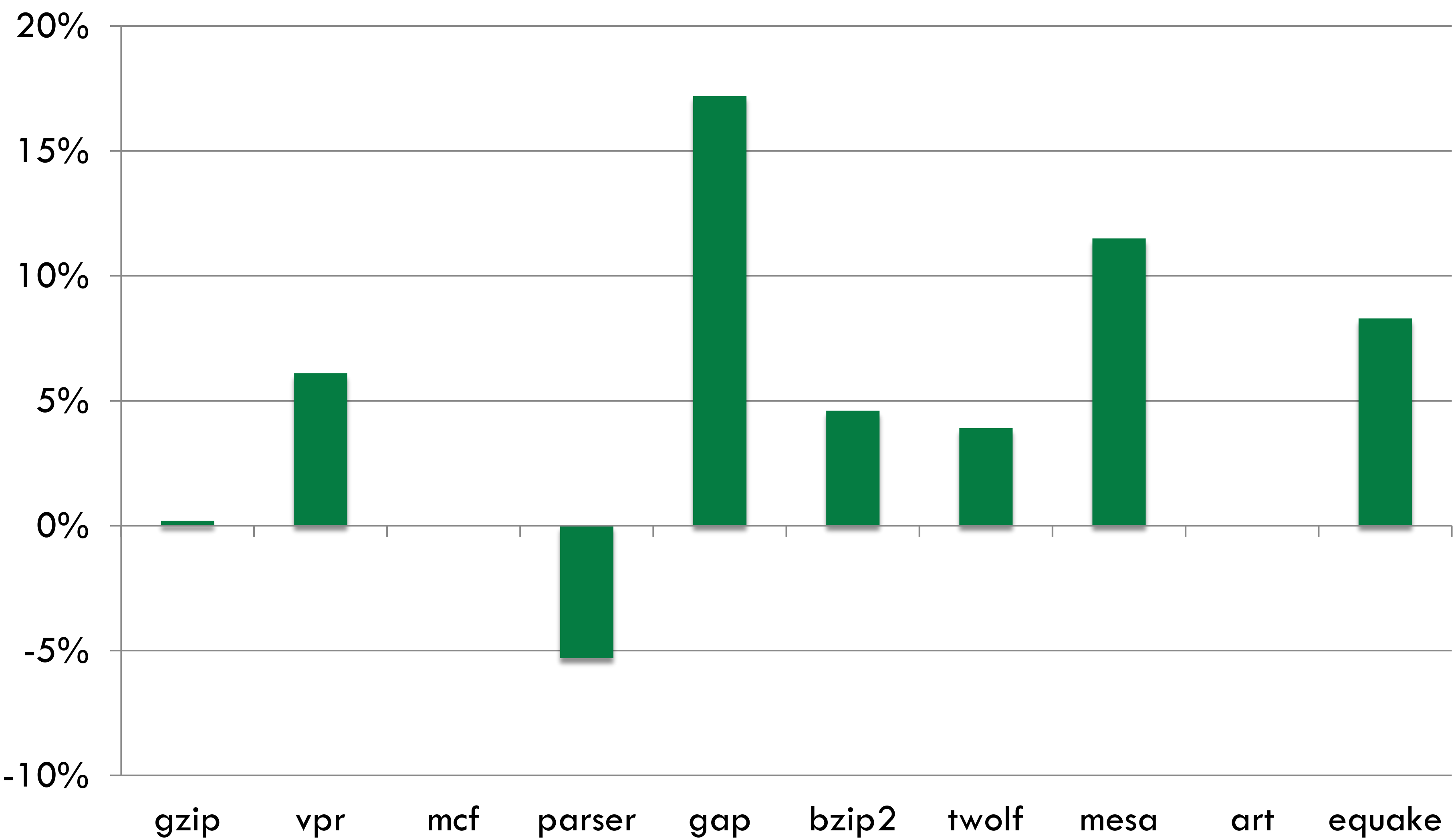
STIR/O-CFI Implementation

- Supports Windows PE and Linux ELF files
- Tested on SPEC2000 benchmarks and the entire coreutils chain for Linux
- 1.5% program runtime efficiency overhead on average
- Wartell, Mohan, Hamlen, and Lin. "Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code." *Proc. ACM Computer and Communications Security* (CCS), 2012.
  - Won 2nd place in the NYU-Poly AT&T Best Applied Security Paper of the Year competition
- Mohan, Larsen, Brunthaler, Hamlen, Franz. "Opaque Control-Flow Integrity." *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2015.
  - Conceals code reachability info to defeat even advanced attackers who can inspect portions of the randomized program memory image!

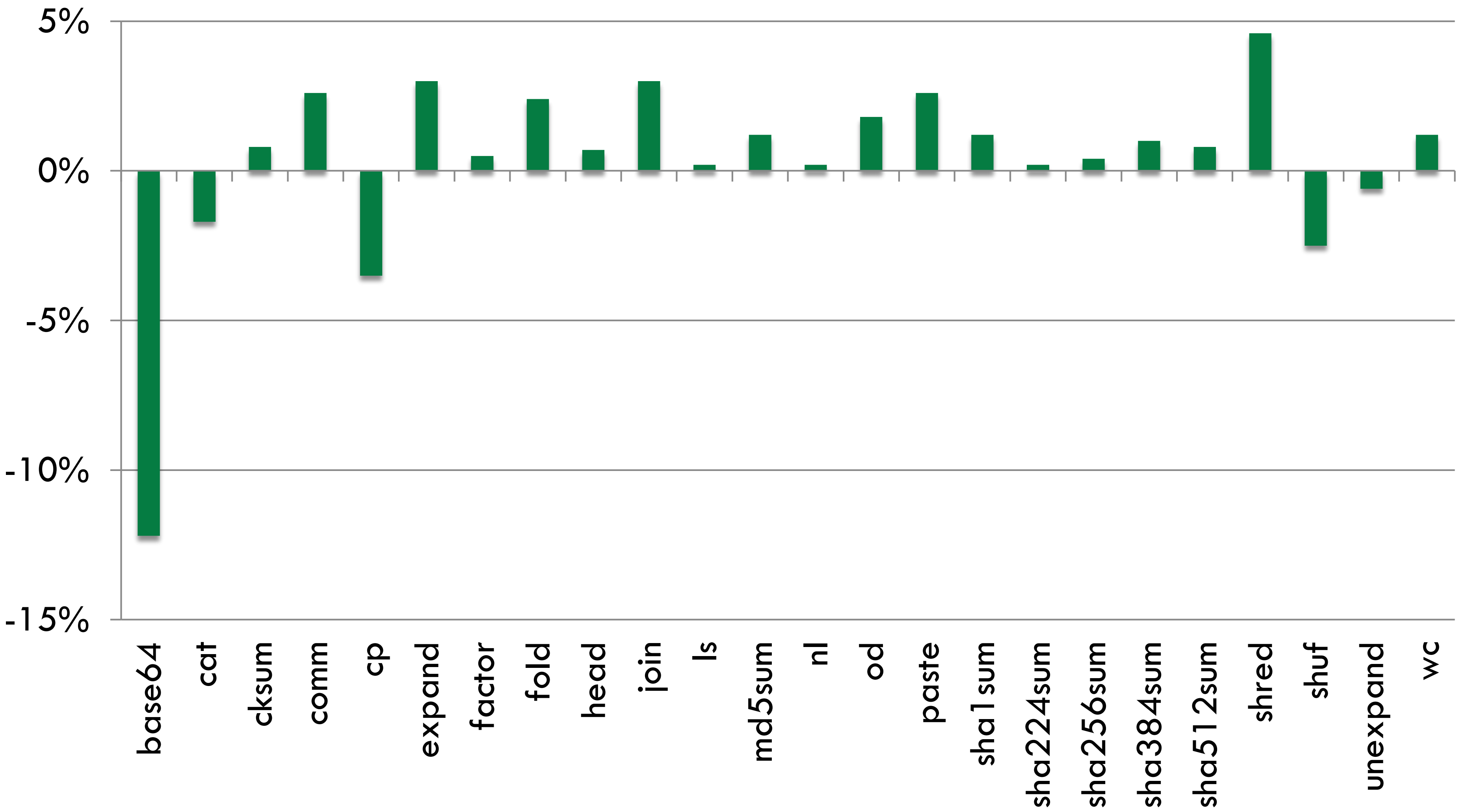
Gadget Reduction



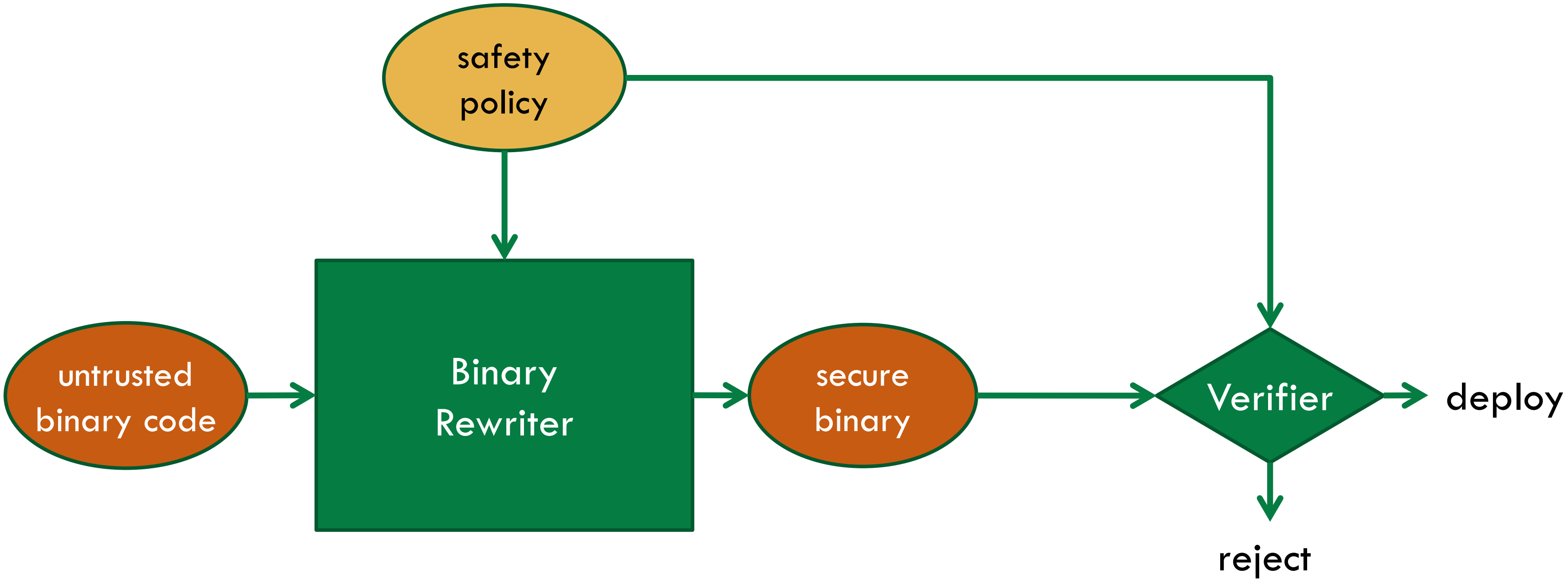
Windows STIR Runtime Overhead



Linux STIR Runtime Overhead



Custom Safety Policy Enforcement with Machine-provable Assurance



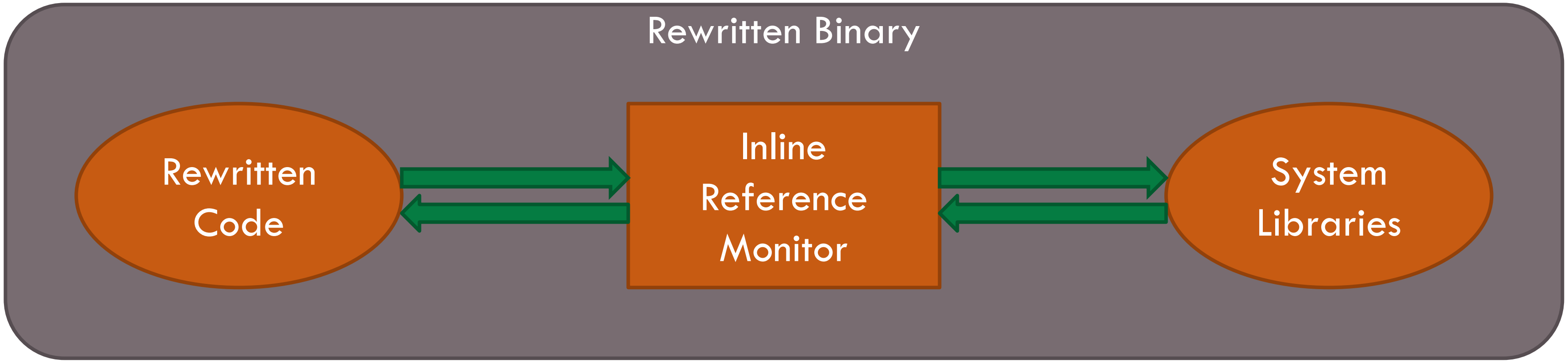
An API Policy


```
function conn = ws2_32::connect(
  SOCKET, struct sockaddr_in *, int) -> int;
function cfile = kernel32::CreateFileW(
  LPCWSTR, DWORD, DWORD, LPSECURITY_ATTRIBUTES,
  DWORD, DWORD, HANDLE) -> HANDLE WINAPI;

event e1 = conn(_, {sin_port=25}, _) -> 0;
event e2 = cfile("*.exe", _, _, _, _, _, _) -> _;

policy = e1* + e2*;
```

Policy: Applications may not both open email connections *and* create files whose names end in ".exe".

Reference Monitor In-lining

- In-line security checks as rewriting progresses
  - checks uncircumventable due to control-flow and memory safety
  - ensures *complete mediation*



REINS - Rewriting and In-lining System

- Prototype targets full Windows XP/7/8 OS
  - significantly harder than Linux
- 2.4% average runtime overhead
- 15% average process size increase
- Tested on SPEC2000, malware, and large GUI binaries
  - Eureka email client and DOSBox, much larger than any previous implementation had accomplished
- Wartell, Mohan, Hamlen, and Lin. Securing Untrusted Code via Compiler-Agnostic Binary Rewriting. *Proc. 28th Annual Computer Security Applications Conference*, 2012.
  - won Best Student Paper at ACSAC

Control-Flow Safety

- Used PittSFIeld approach [McCamant & Morrisett, 2006]
  - Break binaries into chunks
    - chunk – fixed length (16 byte) basic blocks
  - Only one extra guard instruction necessary
  - Mask instruction only affects violating flows

Original
`call eax`

Rewritten
`cmp [eax], 0xF4`
`cmovz eax, [eax+1]`
`and eax, 0x0FFFFFF0`
`call eax`

Jump Table w/ Masking

Original Instruction: eax = 0x411A40

.text:0040CC9B	FF DO	call eax

Original Possible Target:

.text:00411A40	5B	pop ebp

Rewritten Instructions: eax = 0x411A40 / eax = 0x534AB0

.tnew:0052A1C0	80 38 F4	cmp byte ptr [eax], F4h
.tnew:0052A1C3	0F 44 40 01	cmovz eax, [eax+1]
.tnew:0052A1C7		and eax, 0x0FFFFFF0
.tnew:0052A1CE	FF D0	call eax

Rewritten Jump Table:

.told:00411A40	F4 B9 4A 53 00	F4 dw 0x534AB0

Rewritten Target:

.tnew:00534AB0	5B	pop ebp

Next Two Lectures

- Wednesday: Some of our most recent work for Navy and DARPA
  - automated binary software *attack surface reduction* using technologies underlying STIR
- Monday: The sciences behind it all…
  - Theory of In-lined Reference Monitors (IRMs)
  - Computability theory and Enforceability theory

Selected References

1. Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. **Opaque Control-Flow Integrity.** In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS)*, February 2015.
2. Frederico Araujo, Kevin W. Hamlen, Sebastian Bierdermann, and Stefan Katzenbeisser. **From Patches to Honey-Patches: Lightweight Attacker Misdirection, Deception, and Disinformation.** In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, pp. 942-953, November 2014.
3. Richard Wartell, Yan Zhou, Kevin W. Hamlen, and Murat Kantarcioglu. **Shingled Graph Disassembly: Finding the Undecidable Path.** In *Proceedings of the 18th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, pp. 273-285, May 2014.
4. Safwan Mahmud Khan, Kevin W. Hamlen, and Murat Kantarcioglu. **Silve Lining: Enforcing Secure Information Flow at the Cloud Edge.** In *Proceedings of the 2nd IEEE Conference on Cloud Engineering (IC2E)*, pp. 37-46, March 2014.
5. Kevin W. Hamlen. **Stealthy Software: Next-generation Cyber-attacks and Defenses, Invited paper.** In *Proceedings of the 11th IEEE Intelligence and Informatics Conference (ISI)*, pp. 109-112, June 2013.
6. Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. **Securing Untrusted Code via Compiler-Agnostic Binary Rewriting.** In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, pp. 299-308, December 2012.
7. Richard Wartell, Vishath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. **Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code.** In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pp. 157-168, October 2012.
8. Vishwath Mohan and Kevin W. Hamlen. **Frankenstein: Stitching Malware from Benign Binaries.** In *Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT)*, pp. 77-84, August 2012.

Selected References

9. Kevin W. Hamlen, Micah M. Jones, and Meera Sridhar. **Aspect-oriented Runtime Monitor Certification**. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 126-140, March-April 2012.

10. Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. **Differentiating Code from Data in x86 Binaries**. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, Vol. 3, pp. 522-536, September 2011.

11. Micah Jones and Kevin W. Hamlen. **A Service-oriented Approach to Mobile Code Security**. In *Proceedings of the 8th International Conference on Mobile Web Information Systems (MobiWIS)*, pp. 531-538, September 2011.

12. Meera Sridhar and Kevin W. Hamlen. **Flexible In-lined Reference Monitor Certification: Challenges and Future Directions**. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages meets Program Verification (PLPV)*, pp. 55-60, January 2011.

13. Micah Jones and Kevin W. Hamlen. **Disambiguating Aspect-oriented Security Policies**. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD)*, pp. 193-204, March 2010.

14. Aditi Patwardhan, Kevin W. Hamlen, and Kendra Cooper. **Towards Security-aware Visualization for Analyzing In-lined Reference Monitors**. In *Proceedings of the International Workshop on Visual Languages and Computing (VLC)*, pp. 257-260, October 2010.

15. Meera Sridhar and Kevin W. Hamlen. **ActionScript In-lined Reference Monitoring in Prolog**. In *Proceedings of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL)*, pp. 149-151, January 2010.

16. Meera Sridhar and Kevin W. Hamlen. **Model-checking In-lined Reference Monitors**. In *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pp. 312-327, January 2010.