Julian Kirsch

Malicious Bits and How to Fight Them

PhD Thesis Defense

Malicious Bits

STRUCTURE OF THIS PRESENTATION

Malicious Software

Hides Behaviour

Attacks System

Obfuscated Malware

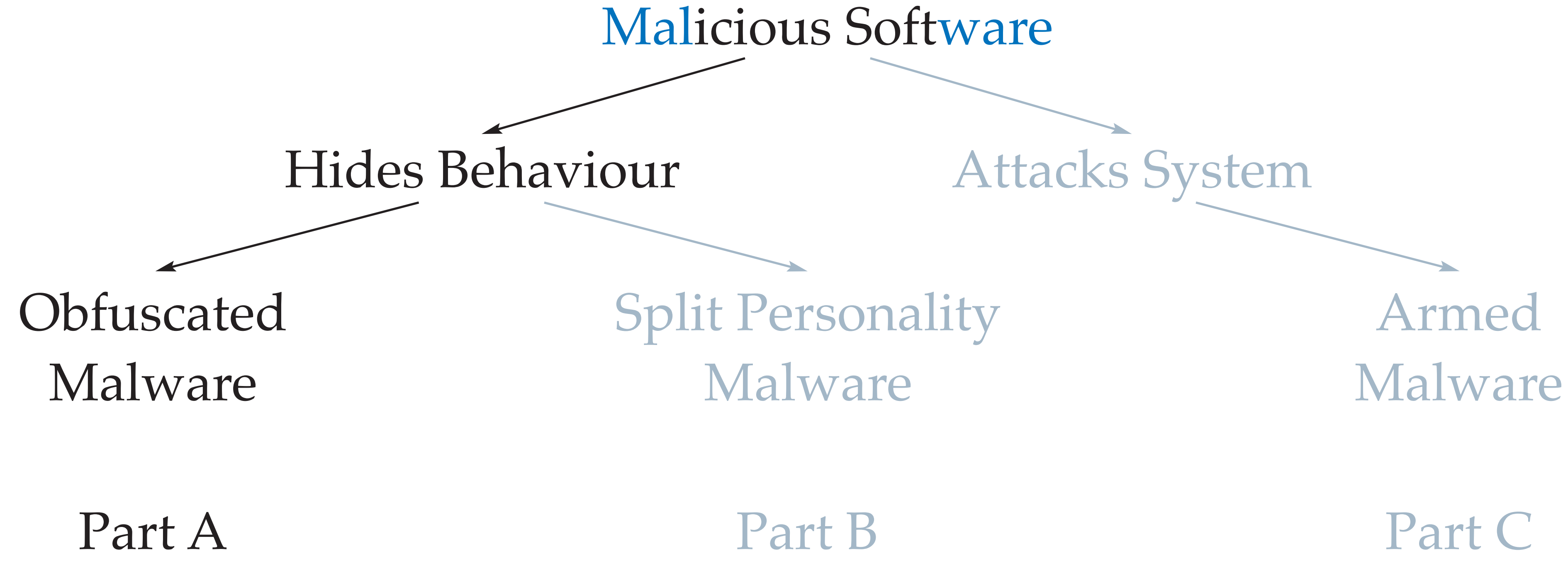
Split Personality Malware

Armed Malware

Part A

Part B

Part C


Malicious Software
Hides Behaviour
Attacks System
Obfuscated Malware
Split Personality Malware
Armed Malware
Part A
Part B
Part C

Obfuscated Malware

„The action of making something obscure, unclear, or unintelligible.“

—Oxford English Dictionary

- Malware: Hindering Static Binary Analysis

Original		Obfuscated
$x \leftarrow 13$ $y \leftarrow 37$ $z \leftarrow x + y$ $z = ?$	Obfuscation → ← De-Obfuscation	$x \leftarrow 13$ $y \leftarrow 37$ $z \leftarrow 2 \cdot (x \vee \neg y) - (\neg x \wedge y) - (x \wedge \neg y)$ $z = ?$

Introduction & Motivation

CONTROL FLOW LINEARIZATION

Obfuscation Technique	Description	Analysis
Mixed Boolean Arithmetic Expressions	[3, 8]	[6, 7]
Bogus Control Flow	[1]	[9]
Control Flow Flattening	[1]	[10, 11]
Control Flow Linearization	[4, 5]	?
Self-Modifying Code	[14]	[13]
Opaque Predicates	[1]	[9]
Interpreted Execution	[2]	[12]

Introduction & Motivation

CONTROL FLOW LINEARIZATION

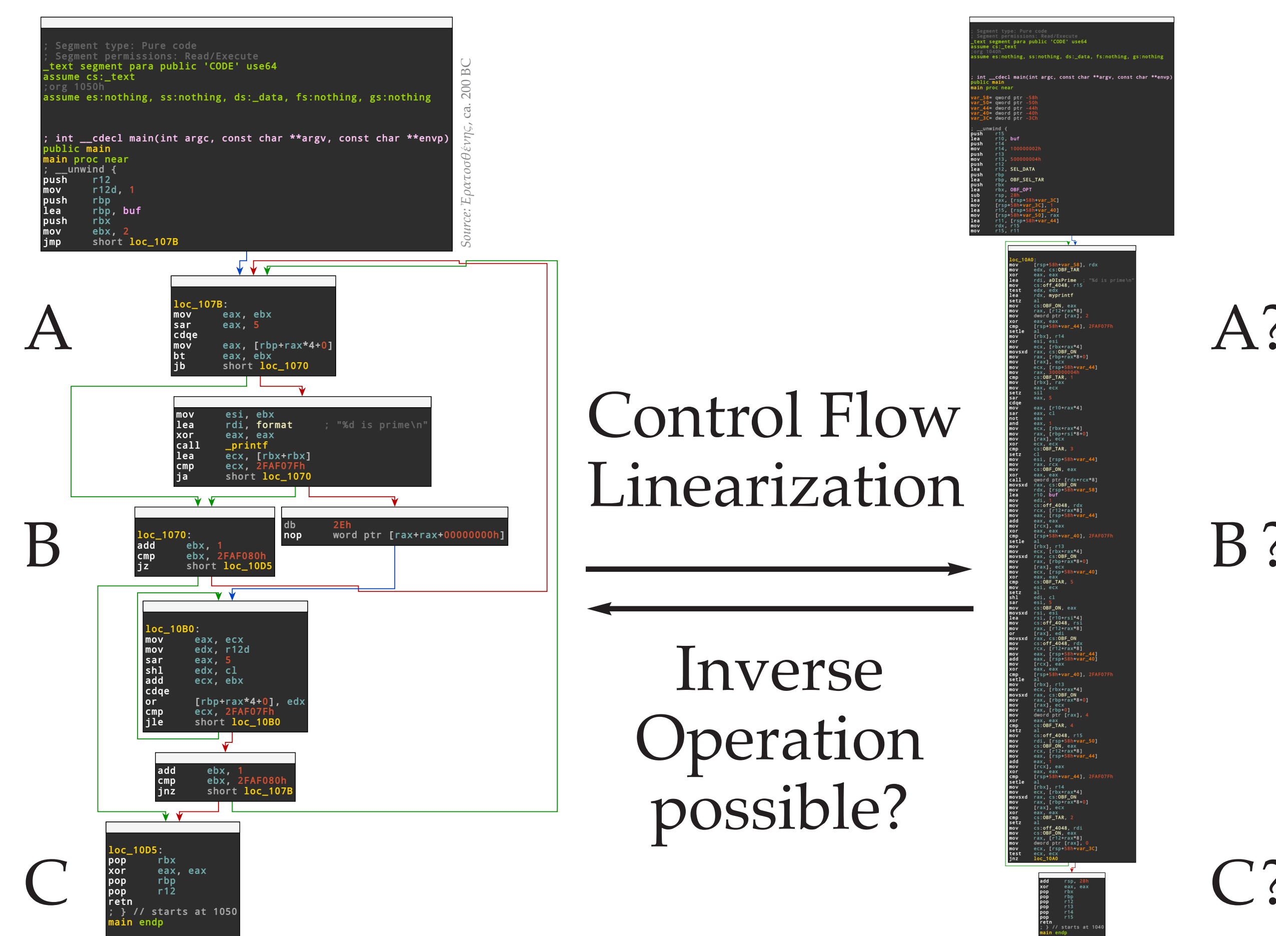
Research Question I

How does Control Flow Linearization impact analysis difficulty, and how can the original control flow graph be reconstructed from linearized machine code?

Obfuscating Transformation

CONTROL FLOW LINEARIZATION

- Control flow is made *implicit*.
- This makes it difficult to …
  - … establish *happens-before* relationships.
  - … enumerate paths through the program.
- First public implementation: *MOVfuscator* [5]



Obfuscating Transformation

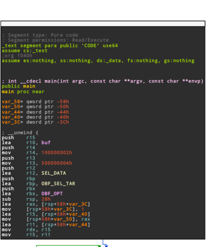
CONTROL FLOW LINEARIZATION

- Structure:
  - Prologue: Stack + Memory Initialization
  - Body: Loop until exit condition is true
  - Epilogue: Stack Teardown
- Central Observations:
  - *All* instructions are executed unconditionally.
  - Memory writes must be *discarded* depending on internal state.

➡ Every variable has a real and a scratch version.

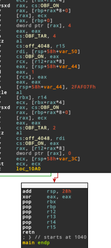
➡ State variable controls what version to target.

Prologue



Body

Epilogue



Obfuscating Transformation

CONTROL FLOW LINEARIZATION

```
1  #include <stdlib.h>
2
3  #define OFF_REAL    1
4  #define OFF_SCRATCH 0
5
6  void nop(void) { return; }
7
8  int main(int argc, char **argv) {
9      unsigned int state[2] = { 0, 0 }; unsigned i[2] = { 0, 13 }; unsigned j[2] = { 0, 37 };
10     int (*exit_ptr[2])(int code) = { nop, exit };
11
12     while (1) {
13                 i[state[OFF_REAL] == 0] += 1;
14                 j[state[OFF_REAL] == 0] -= 1;
15             state[state[OFF_REAL] == 0] = j[state[OFF_REAL] == 0] == 8;
16          exit_ptr[state[OFF_REAL] == 1](i[state[OFF_REAL] == 1]);
17     }
18 }
```

Deobfuscation Approach

CONTROL FLOW LINEARIZATION

```
13             i[state[OFF_REAL] == 0] += 1;
14             j[state[OFF_REAL] == 0] -= 1;
15         state[state[OFF_REAL] == 0] = j[state[OFF_REAL] == 0] == 8;
16      exit_ptr[state[OFF_REAL] == 1](i[state[OFF_REAL] == 1]);
```

Control Flow Graph Recovery:

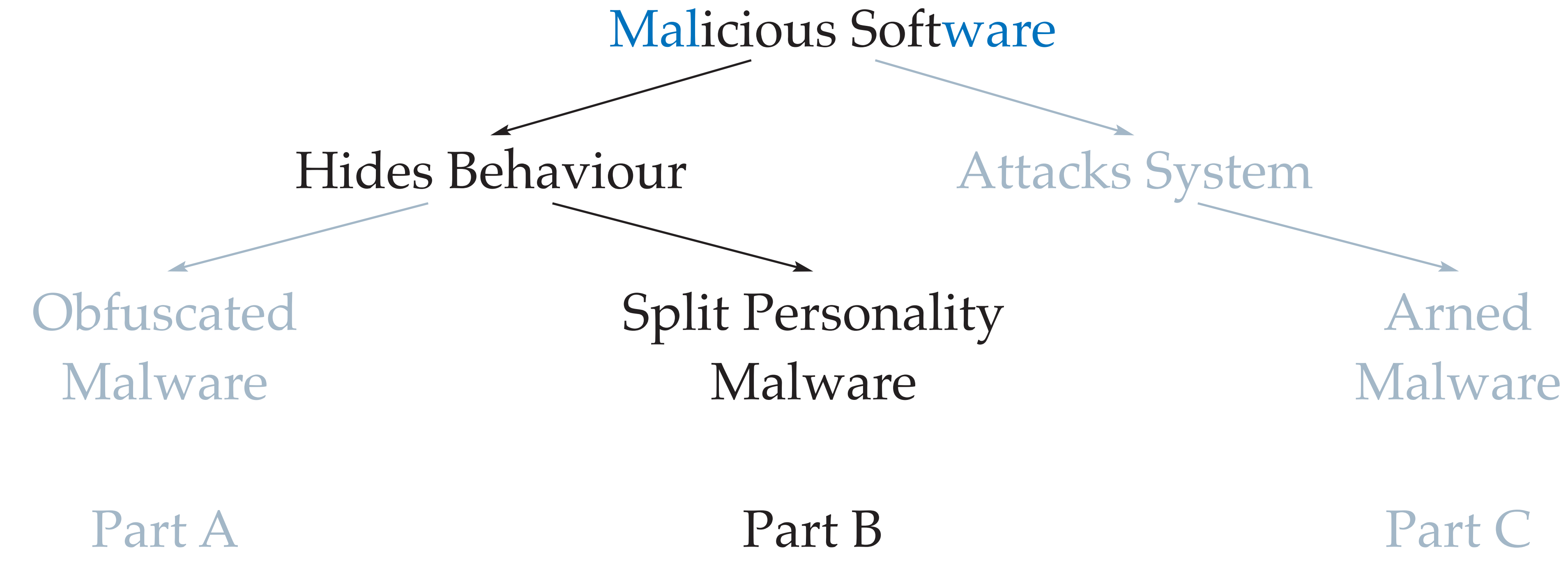
- State variable — Heuristic: most-accessed memory location
- Basic Blocks — Solve constraints on state variable
- (Un)conditional Jumps — Value Set Analysis of assignment to state variable

Evaluation

CONTROL FLOW LINEARIZATION

- Deobfuscation evaluation target: *MOVfuscator*
- Applying angr symbolic execution engine to reference binary [19]:

	Original	**Movfuscated**	**Demovfuscated**
Number of Basic Blocks Executed	37	99,999	87
Analysis Time (s)	5.1	*timeout*	17.9
Explored Paths	2	1	3
Executable Size (bytes)	5400	5,962,776	5,962,776


Malicious Software
Hides Behaviour
Attacks System
Obfuscated Malware
Split Personality Malware
Arned Malware
Part A
Part B
Part C

Split Personality Malware

„[What is] colloquially known as split personality disorder, is a mental disorder characterized by the maintenance of at least two distinct and relatively enduring personality states.“
—*Diagnostic and Statistical Manual of Mental Disorders*

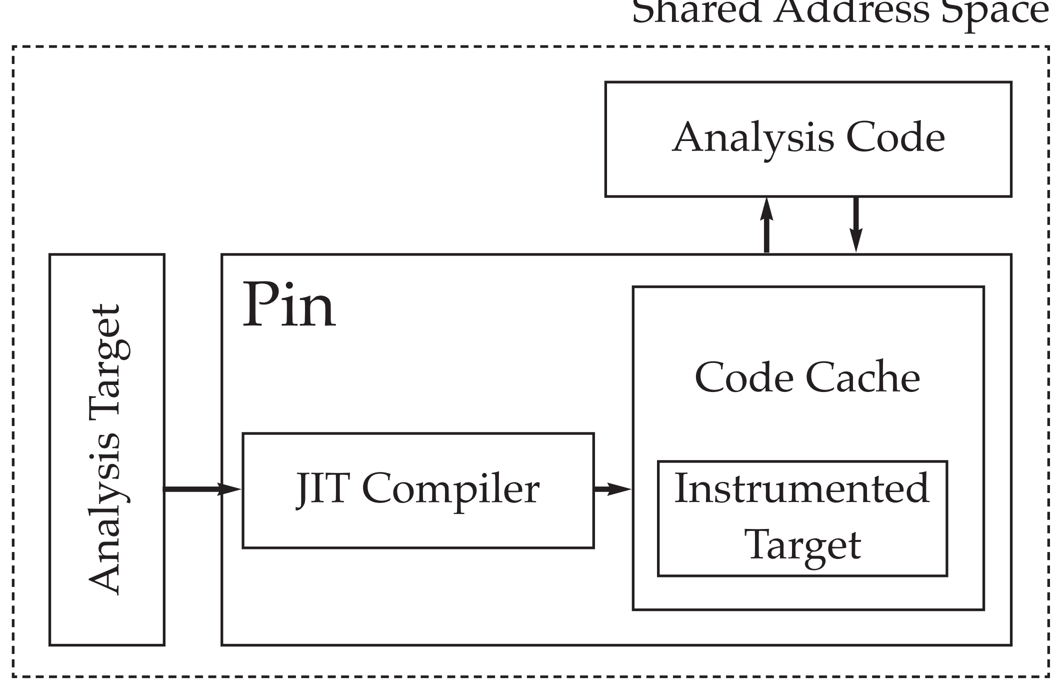
- Malware: Hindering Dynamic Binary Analysis

```
if (debugger_detected()) {
    do_something_unsuspicious();
} else {
    encrypt_files();
}
```

Introduction & Motivation

DYNAMIC BINARY INSTRUMENTATION

- Add code to binary application at specific points during execution. [16]
- Components:
  - Analysis Target
  - Analysis Code
  - DBI Framework
- Security requirements to guarantee reliable dynamic analysis [17, 20]:
  - S1 Interposition
  - S2 Inspection
  - S3 Isolation
  - S4 Transparency



Source: [16]

Introduction & Motivation

DYNAMIC BINARY INSTRUMENTATION

Research Question II

What guarantees on transparency, isolation, interposition, and inspection are provided by current dynamic binary instrumentation tools?

Breaking Transparency

DYNAMIC BINARY INSTRUMENTATION

- Idea: Modern x86-64 CPUs are *complex*.
  How well does Pin handle *corner cases*?

Corner Case	Description	S1 (INTERPOSITION)	S2 (INSPECTION)	S3 (ISOLATION)	S4 (TRANSPARENCY)
`syscall` instruction	Does not update to `rcx` register	?	?	?	✗
`rdfsbase` instruction	Returns Pin TLS instead of guest TLS	?	?	?	✗
`fxsave` instruction [18]	Does not mask original `rip`	?	?	?	✗
Self-Modifying Code	De-synchronizes code cache	?	?	?	✗
No-execute Bit	Ignored (!)	?	?	?	✗

Breaking Isolation

DYNAMIC BINARY INSTRUMENTATION

- Determine code cache location using (1)
- Overwrite cached code and transfer control using (2) + (3)

	Corner Case	Description	INTERPOSITION S1	INSPECTION S2	ISOLATION S3	STEALTHINESS S4
	`syscall` instruction	Does not update to `rcx` register	?	?	?	✗
	`rdfsbase` instruction	Returns Pin TLS instead of guest TLS	?	?	?	✗
(1)	`fxsave` instruction [18]	Does not mask original `rip`	?	?	?	✗
(2)	Self-Modifying Code	De-synchronizes code cache	?	?	?	✗
(3)	No-execute Bit	Ignored (!)	?	?	?	✗

Breaking Isolation

DYNAMIC BINARY INSTRUMENTATION

```
1  get_real_rip:                         ; (1) fxsave method to obtain address
2      fldz                              ;     of fldz instruction in code cache
3      fxsave [rax]
4      mov     rax, [rax+8]
5      ret
6  break_isolation:
7      call        get_real_rip          ;     rax = code cache location
8      lea         rdi, [rel escaped]    ;     rdi = address of escaped@.text
9      mov word   [rax],     0xb848      ; (2) write into cache
10     mov qword  [rax+2],   rdi         ;     movabs rax, <address of escaped>
11     mov word   [rax+10],  0xe0ff      ;     jmp rax
12     call        get_real_rip          ;     call modified get_real_rip
13 escaped:
14     nop
15
```

Breaking Isolation

DYNAMIC BINARY INSTRUMENTATION

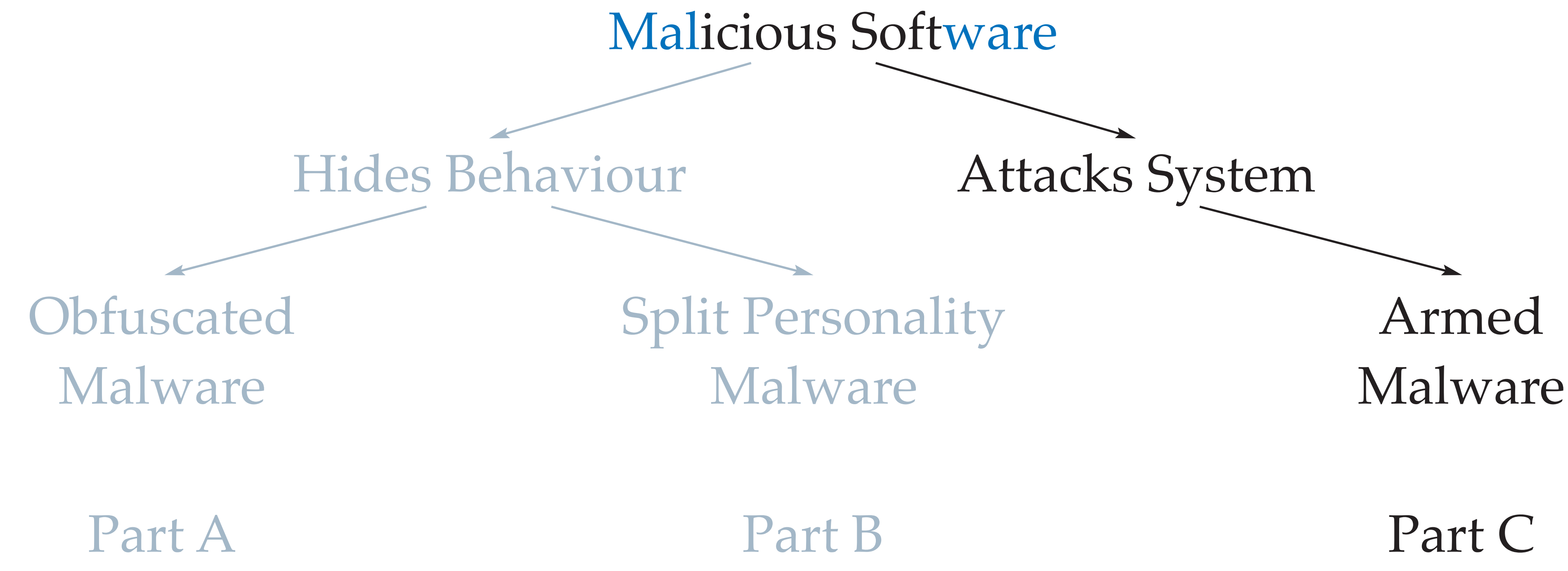
```
1  get_real_rip:                         ; (3) modified function in code
2      movabs    rax, &escaped           ;     cache transfers control to
3      jmp rax                           ;     escaped@.text
4
5
6  break_isolation:
7      call      get_real_rip            ;     rax = code cache location
8      lea       rdi, [rel escaped]      ;     rdi = address of escaped@.text
9      mov word  [rax],      0xb848      ; (2) write into cache
10     mov qword [rax+2],    rdi         ;     movabs rax, <address of escaped>
11     mov word  [rax+10], 0xe0ff        ;     jmp rax
12     call      get_real_rip            ;     call modified get_real_rip
13 escaped:
14     nop                               ; (3) this code executes outside the
15                                       ;     instrumentation vm
```

Breaking Interposition & Inspection

DYNAMIC BINARY INSTRUMENTATION

- Shared address space implies that breaking isolation also breaks inspection and interposition.

Corner Case	Description	S1 (INTERPOSITION)	S2 (INSPECTION)	S3 (ISOLATION)	S4 (STEALTHINESS)
`syscall` instruction	Does not update to `rcx` register	✅	✅	✅	❌
`rdfsbase` instruction	Returns Pin TLS instead of guest TLS	✅	✅	✅	❌
`fxsave` instruction [18]	Does not mask original `rip`	❌	❌	❌	❌
Self-Modifying Code	De-synchronizes code cache	❌	❌	❌	❌
No-execute Bit	Ignored (!)	❌	❌	❌	❌


Malicious Software
Hides Behaviour
Attacks System
Obfuscated Malware
Split Personality Malware
Armed Malware
Part A
Part B
Part C

Introduction & Motivation

ARMED MALWARE

Exploit Mitigation Mechanism	Software Project	Year
Stack Protector	gcc	1997
Address Space Layout Randomization	Linux (PaX)	2001
Write xor Execute	OpenBSD	2003
Relocations Read-Only	gcc	2009
SafeStack	clang	2014
Control Flow Guard	Windows 8.1	2015
Control Flow Integrity	gcc	2018

Introduction & Motivation

ARMED MALWARE

Research Question III

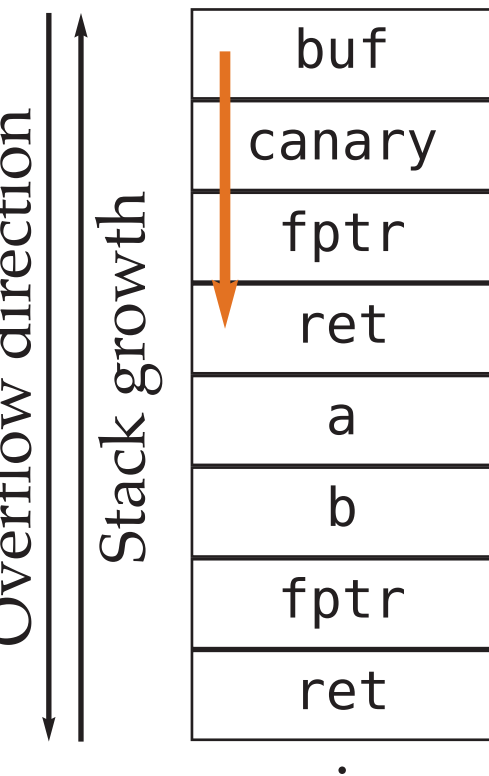
What security guarantees are offered by current versions of the longest-standing exploit mitigations in presence of memory corruption vulnerabilities?

Stack Protector Basic Functionality

SMASHING THE STACK PROTECTOR FOR FUN AND PROFIT

```c
void g(void) {
    uint8_t buf[16];
    /* function body of g */
    return;
}

void f(void) {
    uint64_t a;
    uint64_t b;
    /* function body of f */
    g();
    return;
}
```



1. Place canary on entry

2. Check canary on exit
3. Terminate program if corrupted canary is detected

Ideal Stack Protector Properties

SMASHING THE STACK PROTECTOR FOR FUN AND PROFIT

- P1 Re-randomization of `canary` value
  - per-process, per-thread, per-function
- P2 `reference` value stored in read / only memory far away from architectural stack
- P3 Immediate program termination on detected `canary` value corruption

→ Measure properties across different combinations of hardware and operating systems

reference

Overflow direction

Stack growth

⋮

buf

canary

fptr

ret

a

b

fptr

ret

⋮

State of the Stack Protector

SMASHING THE STACK PROTECTOR FOR FUN AND PROFIT

- P1 Re-randomization:
  - Windows: per-function canary
  - All others: per-process canary randomization, constant across `fork()`
- P2 Storage location of reference value:
  - ✔ Safe implementations (not reachable / not writable)
  - 🔶 Weak implementations (reachable, but not from stack)
  - ❌ Vulnerable implementations (reachable from stack)
- P3 Immediate termination:
  - Linux + glibc: Read attacker controlled values from memory
  - All others terminate the program safely

State of the Stack Protector

SMASHING THE STACK PROTECTOR FOR FUN AND PROFIT

Operating System	CPU Arch.	C Library	Stack		TLS		Global		Dyn.	
			M	S	M	S	M	S	M	S
Android 7.0	ARMv7	Bionic	✓	✓	✓	✓	✓	✓	✓	✓
Android 7.0	x86-64	Bionic	✓	✗	✓	✓	✓	✓	✓	✓
macOS 10.12.1	x86-64	libSystem	✓	✓	✓	✓	✓	✓	✓	✓
FreeBSD 11.00	x86-64	libc.so.7	✓	✓	✓	✓	🔶	🔶	✓	✓
OpenBSD 6.0	x86-64	libc.so.88.0	✓	✓	–	–	✓	✓	✓	✓
Windows 10	x86	msvcr1400	✓	✓	✓	✓	🔶	🔶	✓	✓
Windows 10	x86-64	msvcr1400	✓	✓	✓	✓	🔶	🔶	✓	✓
Windows 7	x86	msvcr1400	✓	✓	✓	✓	🔶	🔶	✓	✓
Windows 7	x86-64	msvcr1400	✓	✓	✓	✓	🔶	🔶	✓	✓

State of the Stack Protector

SMASHING THE STACK PROTECTOR FOR FUN AND PROFIT

			Stack		TLS		Global		Dyn.	
Operating System	**CPU Arch.**	**C Library**	**M**	**S**	**M**	**S**	**M**	**S**	**M**	**S**
Arch Linux	x86-64	libc-2.26.so	✔	❌	🔶	🔶	✔	✔	✔	✔
Debian Jessie	x86	libc-2.19.so	✔	❌	🔶	🔶	✔	✔	✔	✔
Debian Jessie	ARMv7	libc-2.19.so	✔	✔	✔	✔	🔶	🔶	✔	✔
Debian Jessie	PowerPC	libc-2.19.so	✔	❌	🔶	🔶	✔	✔	✔	✔
Debian Jessie	s390x	libc-2.19.so	✔	❌	🔶	🔶	✔	✔	✔	✔
Debian Stretch	x86-64	diet 0.33	❌	❌	🔶	🔶	✔	✔	✔	🔶
Debian Stretch	x86-64	musl 1.1.16	✔	❌	🔶	🔶	🔶	✔	🔶	✔
Ubuntu 14.04 LTS	x86-64	eglibc 2.15	✔	❌	🔶	🔶	✔	✔	✔	✔

Address Space Layout Randomization

DYNAMIC LOADER ORIENTED PROGRAMMING ON LINUX

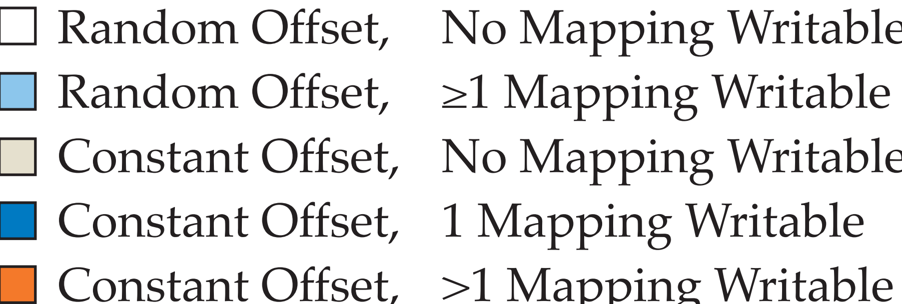
- Address Space Layout Randomization (ASLR) makes *absolute* position of objects in memory unknown.
- Idea: Measure *relative* positions of objects in memory *among each other*.

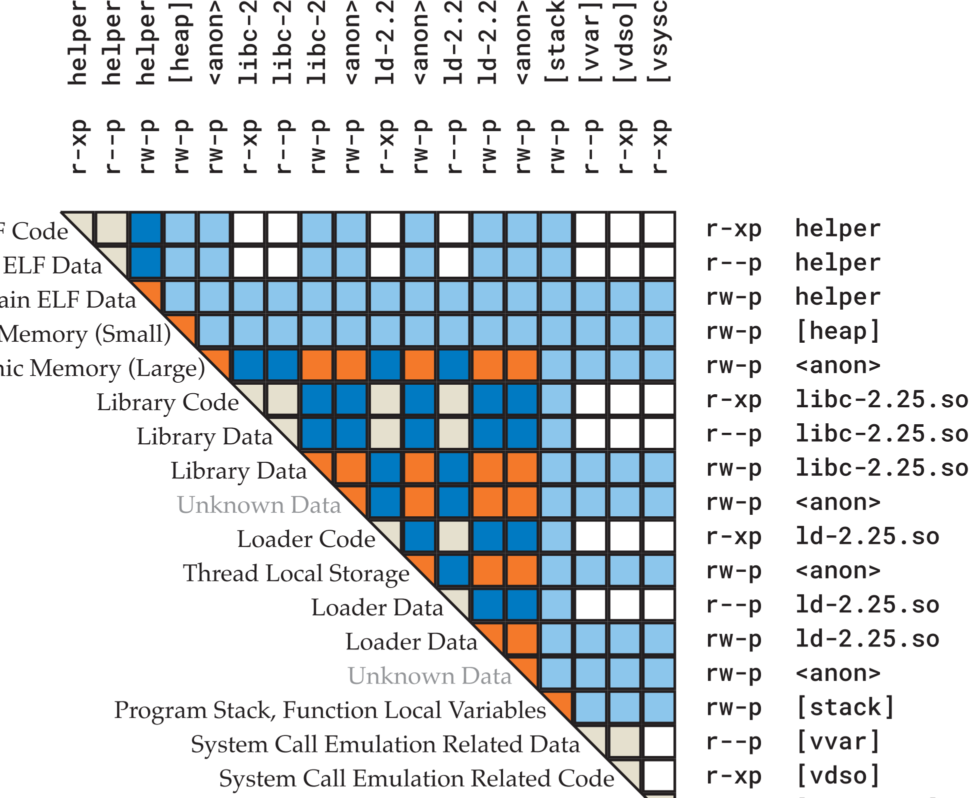
```
$ cat /proc/self/maps | head -c12
62c02ae5a000
$ cat /proc/self/maps | head -c12
5b57a714d000
$ cat /proc/self/maps | head -c12
55c5bcf9d000
$ cat /proc/self/maps | head -c12
64760e613000
$ cat /proc/self/maps | head -c12
5b49c20df000
$ cat /proc/self/maps | head -c12
5c96974c8000
$ cat /proc/self/maps | head -c12
5dda53031000
$ cat /proc/self/maps | head -c12
64d2e1f7f000
```

Address Space Layout Randomization

DYNAMIC LOADER ORIENTED PROGRAMMING ON LINUX

- Relative distancing of memory mappings with full ASLR on Linux
- Which mappings contain writable function pointers that get dispatched reliably?





Identifying Attack Targets

DYNAMIC LOADER ORIENTED PROGRAMMING ON LINUX

- Record instruction trace of most basic interaction of user binary with standard runtime: Process Termination
- Perform taint analysis to determine extended *is-writable-*property on dispatched code pointers:
  - writable code pointers (direct)
  - operations applying writable operands to code pointers (indirect)

The Wiedergänger Attack

DYNAMIC LOADER ORIENTED PROGRAMMING ON LINUX

- In presence of unbound array access vulnerability:

```
1  int main(int argc, char **argv) {
2      /* Exemplary initialization */
3      uint8_t *array = malloc(0x200000);
4      size_t idx = 0, val = 0;
5
6      while (scanf("%zu %zu", &idx, &val) == 2) {
7          array[idx] = val;
8      }
9
10     return 0;
11 }
```

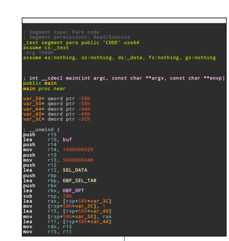
→ Full ASLR bypass via corrupted structures in dynamic loader on Linux

Control Flow Linearization

CONCLUSION

- CFL incurs significant performance overhead.
- CFL effectively breaks symbolic execution.
- Control flow can be recovered by our algorithm.
- Implementation of the algorithm targeting *MOVfuscator.*

➡ Apply CFL selectively to algorithmic core only.



Dynamic Binary Instrumentation

CONCLUSION

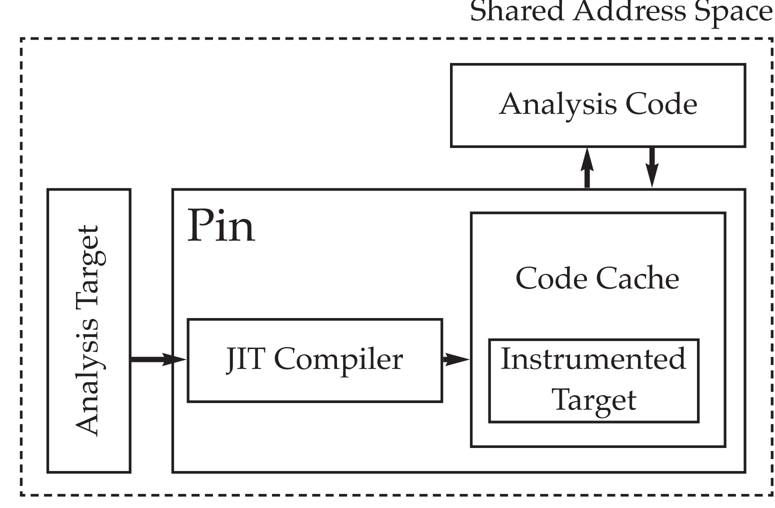
- Pin fails to provide transparency, isolation, inspection, interposition guarantees

➡ Pin is unsuitable for analysis of untrusted code

- We escalated a DoS bug in *wget* to full code execution when instrumented

➡ Instrumented code is less secure against exploitation

➡ Pin is unsuitable to enhance binary production code



Exploit Mitigations on Linux

CONCLUSION

- Weak stack canary implementations on Linux:
  - No re-randomization of canary value when forking a new process, starting a thread or entering a function
  - ➡ Stack Canaries in forking software can *easily* be bypassed
  - Reference value is placed in writable memory next to stack
  - ➡ Stack Canaries in threading software can *trivially* be bypassed
- ASLR on Linux places memory segments at fixed relative distances
- Control structures of dynamic loader can be corrupted

➡ Full ASLR bypass possible for unbound array access vulnerabilities

- More formal methods for low-level computer security topics
- Strong(er?) transparency of dynamic binary analysis
- Introduction of hardened implementations of ASLR & stack protector for Linux+glibc

Questions?

ありがとうございます。

Shared Address Space

Analysis Code

Analysis Target

Pin

Code Cache

JIT Compiler

Instrumented Target

reference

Overflow direction

Stack growth

⋮
buf
canary
fptr
ret
a
b
fptr
ret
⋮

Main ELF Code	r-xp helper
Main ELF Data	r--p helper
Main ELF Data	rw-p helper
Dynamic Memory (Small)	rw-p [heap]
Dynamic Memory (Large)	rw-p <anon>
Library Code	r-xp libc-2.25.so
Library Data	r--p libc-2.25.so
Library Data	rw-p libc-2.25.so
Unknown Data	rw-p <anon>
Loader Code	r-xp ld-2.25.so
Thread Local Storage	rw-p <anon>
Loader Data	r--p ld-2.25.so
Loader Data	rw-p ld-2.25.so
Unknown Data	rw-p <anon>
Program Stack, Function Local Variables	rw-p [stack]
System Call Emulation Related Data	r--p [vvar]
System Call Emulation Related Code	r-xp [vdso]
System Call Emulation Related Code	r-xp [vsyscall]

References

FOR FULL REFERENCE LIST, SEE THESIS

- [1] Pascal Junod et al. • *Obfuscator-LLVM — Software Protection for the Masses*
- [2] Sudeep Ghosh et al. • *Matryoshka: Strengthening Software Protection via Nested Virtual Machines*
- [3] Yongxin Zhou et al. • *Information Hiding in Software with Mixed Boolean-Arithmetic Transforms*
- [4] Stephen Dolan • *Mov is Turing-Complete*
- [5] Christopher Domas • *The MOVfuscator*

References

FOR FULL REFERENCE LIST, SEE THESIS

- [6] Adrien Guinet et al. • *Arybo: Manipulation, Canonicalization and Identification of Mixed Boolean-Arithmetic Symbolic Expressions*
- [7] Binbin Liu et al. • *MBA-Blast: Onveiling and Simplifying Mixed Boolean-Arithmetic Obfuscation*
- [8] Henry Warren • *Hacker's Delight*
- [9] Yan Shoshitaishvili et al. • *Firmalice – Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware*

References

FOR FULL REFERENCE LIST, SEE THESIS

- [10] ESET Research Laboratories • *Stadeo – Stantinko Botnet Analysis Tools*
- [11] Sophos Research Laboratories • *Attacking Emotet's Control Flow Flattening*
- [12] Rolph Rolles • *Deobfuscating VMProtect*
- [13] Babak Yadegari et al. • *A Generic Approach to Automatic Deobfuscation of Executable Code*
- [14] Peter Nordin et al. • *Evolving Turing-Complete Programs for a Register Machine with Self-modifying Code*

References

FOR FULL REFERENCE LIST, SEE THESIS

- [15] Davide Balzarotti et al. • *Efficient Detection of Split Personalities in Malware*
- [16] Chi-Keung Luk et al. • *Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation*
- [17] Tal Garfinkel et al. • *A Virtual Machine Introspection Based Architecture for Intrusion Detection*
- [18] Francisco Falcón et al. • *Dynamic Binary Instrumentation Frameworks: I know you're there spying on me*

References

FOR FULL REFERENCE LIST, SEE THESIS

[19] Yan Shoshitaishvili et al. • *Firmalice – Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware*

[20] Tamas Lengyel et al. • *Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System*

Backup Slides

Dynamic Analysis Approaches

DYNAMIC BINARY INSTRUMENTATION

- Security requirements to guarantee reliable analysis [17, 20]:
  - S1 Interposition
  - S2 Inspection
  - S3 Isolation
  - S4 Transparency

Dynamic Analysis Interface	Detection
Operating System	OS Artefacts [15]
System / Process Emulator	CPU semantics discrepancies [15]
Virtual Machine	Timing Discrepancies/Overhead [15]
Dynamic Binary Instrumentation	?

Data Collection Program

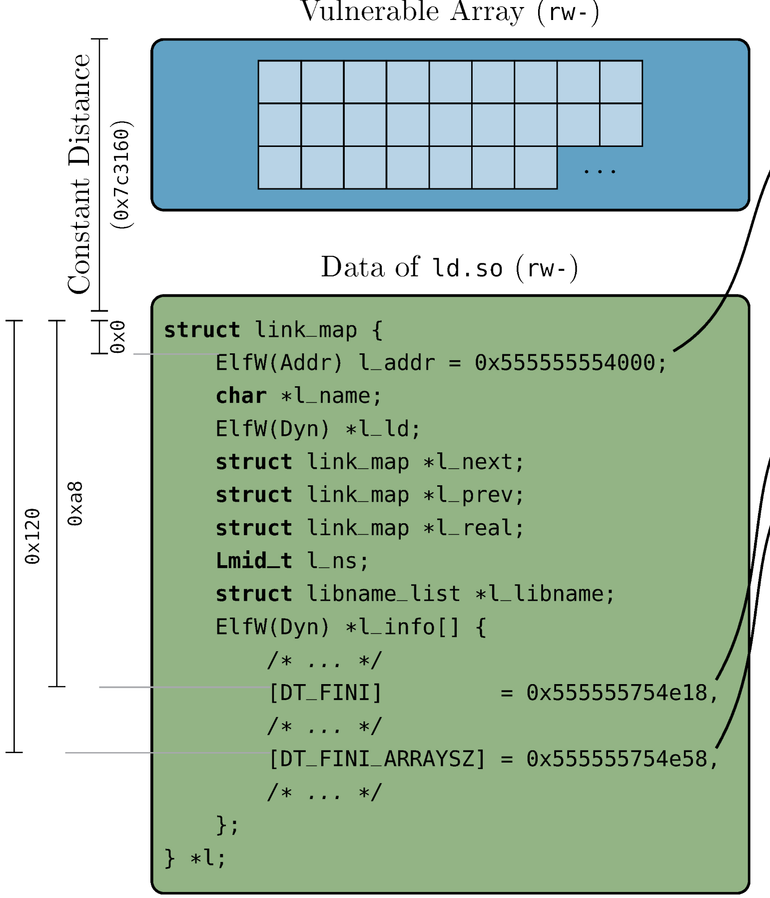
SMASHING THE STACK PROTECTOR FOR FUN AND PROFIT

- P1: Determine canary value
  - when spawning a new process.
  - when spawning a new thread.
  - when entering a new function.
- P2: Simulate buffer overflow from user-controllable
  - stack memory
  - thread local storage memory
  - global static memory
  - dynamically allocated memory

  } to reference value.
- P3: Corrupt canary and trace execution flow

The Wiedergänger Attack

DYNAMIC LOADER ORIENTED PROGRAMMING ON LINUX



Main ELF

ELF Base (r-x):

```
7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00
03 00 3e 00 01 00 00 00  80 05 00 00 00 00 00 00
40 00 00 00 00 00 00 00  98 19 00 00 00 00 00 00
⋮
```

.dynamic Section (r--):

```
01 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00
0c 00 00 00 00 00 00 00  28 05 00 00 00 00 00 00
0d 00 00 00 00 00 00 00  74 07 00 00 00 00 00 00
19 00 00 00 00 00 00 00  e8 0d 20 00 00 00 00 00
1b 00 00 00 00 00 00 00  08 00 00 00 00 00 00 00
1a 00 00 00 00 00 00 00  f0 0d 20 00 00 00 00 00
1c 00 00 00 00 00 00 00  08 00 00 00 00 00 00 00
f5 fe ff 6f 00 00 00 00  98 42 55 55 55 55 00 00
05 00 00 00 00 00 00 00  78 43 55 55 55 55 00 00
06 00 00 00 00 00 00 00  b8 42 55 55 55 55 00 00
0a 00 00 00 00 00 00 00  8b 00 00 00 00 00 00 00
0b 00 00 00 00 00 00 00  18 00 00 00 00 00 00 00
15 00 00 00 00 00 00 00  40 e1 48 ad 3a 7f 00 00
03 00 00 00 00 00 00 00  00 50 75 55 55 55 00 00
02 00 00 00 00 00 00 00  30 00 00 00 00 00 00 00
⋮
```

Tag	Value
DT_NEEDED:	Shared library
DT_INIT:	0x528
DT_FINI:	0x774
DT_INIT_ARRAY:	0x200de8
DT_INIT_ARRAYSZ:	8 Bytes
DT_FINI_ARRAY:	0x200df0
DT_FINI_ARRAYSZ:	8 Bytes
DT_GNU_HASH:	0x555555554298
DT_STRTAB:	0x555555554378
DT_SYMTAB:	0x5555555542b8
DT_STRSZ:	0x8b
DT_SYMENT:	0x18
DT_DEBUG:	0x7f3aad48e140 (ld.so:_r_debug= =ld.so+0x225140=libc.so+0x5c2140)
DT_PLTGOT:	0x555555755000
DT_PLTRELSZ:	0x30

Possible due to Constant Offset

The Wiedergänger Attack

DYNAMIC LOADER ORIENTED PROGRAMMING ON LINUX