6.033 in the news

APRIL 30, 2021

Scientists discover new vulnerability affecting computers globally

by Audra Book, University of Virginia School of Engineering and Applied Science

A team of University of Virginia School of Engineering computer science researchers has uncovered a line of attack that breaks all Spectre defenses, meaning that billions of computers and other devices across the globe are just as vulnerable today as they were when Spectre was first announced. The team reported its discovery to international chip makers in April and will present the new challenge at a worldwide computing architecture conference in June.

The researchers, led by Ashish Venkat, William Wulf Career Enhancement Assistant Professor of Computer Science at UVA Engineering, found a whole new way for hackers to exploit something called a "micro-op cache," which speeds up computing by storing simple commands and allowing the processor to fetch them quickly and early in the speculative execution process. Micro-op caches have been built into Intel computers manufactured since 2011.

Venkat's team discovered that hackers can steal data when a processor fetches commands from the micro-op cache.

https://techxplore.com/news/2021-04-scientists-vulnerability-affecting-globally.html

6.033 in the news

I See Dead μops: Leaking Secrets via Intel/AMD Micro-Op Caches

Xida Ren
University of Virginia
renxida@virginia.edu

Logan Moody
University of Virginia
lgm4xn@virginia.edu

Mohammadkazem Taram
University of California, San Diego
mtaram@cs.ucsd.edu

Matthew Jordan
University of Virginia
mrj3dd@virginia.edu

Dean M. Tullsen
University of California, San Diego
tullsen@cs.ucsd.edu

Ashish Venkat
University of Virginia
venkat@virginia.edu

Performance Counter-Based Monitoring. A more light-weight alternative to disabling and flushing the micro-op cache is to leverage performance counters to detect anomalies and/or potential malicious activity in the micro-op cache. For instance, sudden jumps in the micro-op cache misses may reveal a potential attack. However, such a technique is not only inherently prone to misclassification errors, but may also be vulnerable to *mimicry* attacks [79], [80] that can evade detection. Moreover, to gather fine-grained measurements, it is imperative that performance counters are probed frequently, which could in turn have a significant impact on performance.

Potential Mitigations

This section discusses potential attack mitigations that could block information leakage over the micro-op cache.

Flushing the Micro-Op Cache at Domain Crossings. Cross-domain information leakage via this side channel may be prevented by flushing the micro-op cache at appropriate protection domain crossings. This can be simply accomplished with current hardware by flushing the instruction Translation Lookaside Buffer (iTLB), which in turn forces a flush of the entire micro-op cache. Intel SGX already does this at enclave entry/exit points, and as a result both the enclave and the non-enclave code leave no trace in the micro-op cache for side-channel inference.

However, frequent flushing of the micro-op cache could severely degrade performance. Furthermore, given that current

Privilege Level-Based Partitioning. Intel micro-op caches are statically partitioned to allow for performance isolation of SMT threads that run on the same physical core, but different logical cores. However, this does not prevent same address-space attacks where secret information is leaked within the same thread, but across protection domains, by unauthorized means. A countermeasure would be to extend this partitioning based on the current privilege-level of the code, so for example, kernel and user code don't interfere with each other in the micro-op cache. However, this may not be scalable as the number of protection domains increase, and considering the relatively small size of the micro-op cache, such a partitioning scheme would result in heavy underutilization of the micro-op cache, negating much of its performance advantages.

https://www.cs.virginia.edu/venkat/papers/isca2021a.pdf

6.033 Spring 2021

Lecture #21: Low-level Exploits

smashing stacks, trusting trust

Katrina LaCurts | lacurts@mit.edu | 6.033 2021

username	salt	slow_hash(password|salt)
user1	TU6kbcuPm7jA./IQYZG.8O	rBda9fbnXhUCWi6c9MjlUtQFlK1I4Sq
user2	y7oSC2QsrvXTzEZlDZFdwu	tjFcpSrZN6ryOYueyrAtUfFnFaOui2C
user3	4ncYRSB5v3rWiU1nPpA0iu	hacrgRlfU44c9XnBckef2fu.ifuB.Ya
user4	SK9H4x4Ha0Owz4NOTwj20.	wWk3GjGeMspoqy3VcMghpbkE5OjHQXS
user5	j8YyeDX.9GnsT5Hu94z7tO	ViflhwGHl.5H3j0mawzBPdKTiXf5L6.
user6	CIqY72CGM8KNQId1CqXY7.	stk3mDJDaaH9Nfgf/ePJrkRoK15.Heu
user7	OGtMXrEZEx0L544Odvrhbe	A.7NaJc21Y6I3J6rdJtiIJXVpMvaMgG
user8	RFeT9TVol8cmpQdhqMCV5.	yVzcp0OjXBoNjcMWHpAxVulFqdM5W9m
user9	rDAhDK5V6n3TUS3ahf2Z9e	Af4wBH1YqLTvxrhBgVGP85IALXRya3C
user10	nvOYvT0/oczOW5lmbVZSUO	b4miFmYcRy0/TFVhttntbrrLPLjFDKu
user11	yNL/e3PpBsfBYgwi0Ai/gu	bbT5sTcmsklsyXVILfVdJ/HAIEonb..
user12	1zroUl0scwDzgG3GY86pFO	MG5LtQ6m/c4gVxbLalpPIJ4O3eXFPry
user13	TAkv7nBQ5amY4V.aIjez0u	LHPo8.0XJDGleWWgG87nPvY8/vNPa2G
user14	1J796dTzufUC8ItVIKIyOu	pAI7ZRWvVOhxBVW/sttFquJCl/74LTC
user15	/x.Vk/XhUILbk3XjgyVyfO	zx1P3YgW8d9m1n9lZ6GW7jsbBALniWi
user16	hyg8T0JPDX3dCf92Zkx4Yu	5Oh.8uSUrokBgqnByYYH/mDEH7my98C
user17	YbaYOSdkA01IF.drWa6CXO	ZKbZQtEh4UNoTflWsXs9hZ7wbnnzgC.
user18	yaE.gULeQg.K2SelX191Q.	E/syZIC.1.zg5.ZTMZwWX/RmkvpipNu
user19	NLt0SA/QPo2IIbtb7G56lO	eOX2p48XcKRXKFY87f56h3W.UEeO7Gi
user20	RFFSWUGGFeX5XNyW8rLToe	0W94ciFDN5stvqVzYsli4t/SNA2pwhS
user21	YWEgwinWuKrNUFvgzQKUNe	yatU0vWN//72Ul8OdxGHnClTLWdTfXe
user22	ukqUgoOZWCqIQjH3DwC4xe	jg1.OSatbZooR6l4taWv3HBpXNN5Xp2
user23	sPRFpmFnu5G4lAPUkV0wrO	mVpzAYGXEgs583nG894R98k1S3YmP1q

policy: provide **authentication** for users

threat model: adversary has access to the entire stored table

last time, our threat model allowed for an adversary that had access to some sensitive data stored on our machine

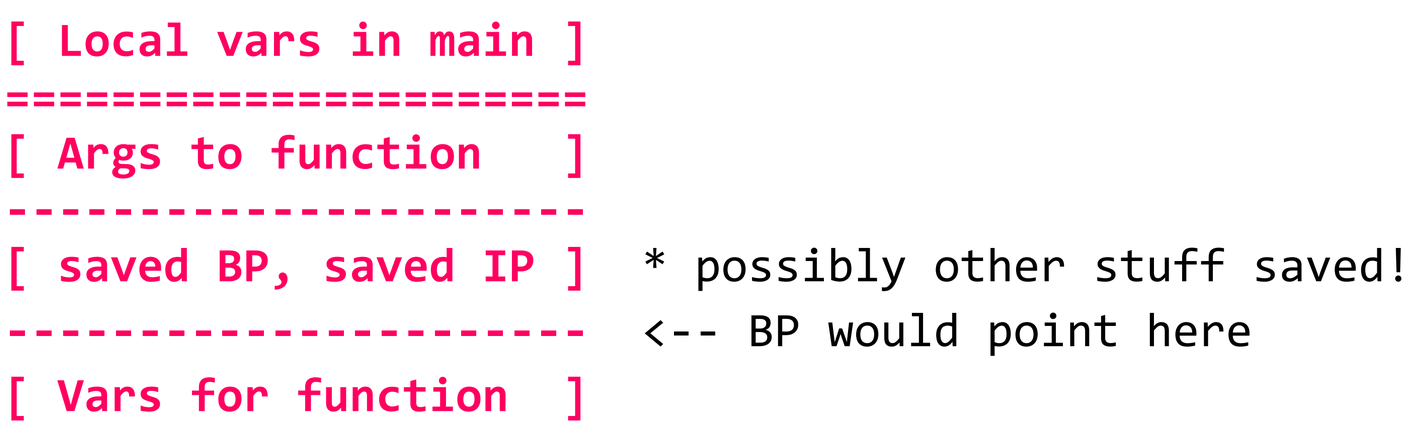
a straightforward adversary in this case is someone like a system administrator, who is intended to have access to this data

today, we'll look at how an adversary that is *not* intended to have access to this data might get it

our threat model for most of today is an adversary with the ability to run code on our machine, but not necessarily any particular privileges (e.g., root access)

```
void function(int a) {
  // do whatever
}

void main() {
  int x = 0;
  function(7);
  // maybe other stuff here
}
```

when we're in **function()**, **IP** (the instruction pointer) points to the instruction we're executing. when **function()** ends, we need to be able to get back to where we were in **main()**



IP = Instruction pointer

BP = Base pointer ("frame pointer")

adversary's goal: input a string that changes `modified`

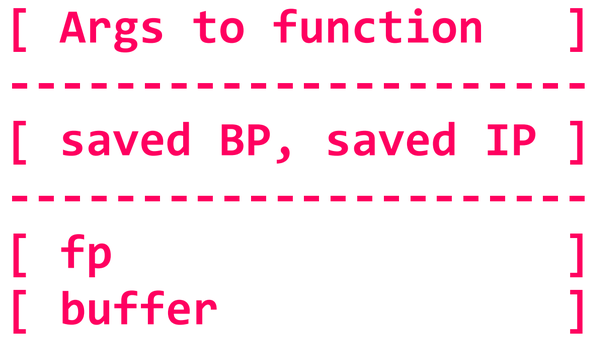
```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  volatile int modified;
  char buffer[64];

  modified = 0;
  gets(buffer);

  if(modified != 0) {
      printf("you have changed the 'modified' variable\n");
  } else {
      printf("Try again?\n");
  }
}
```

IP = Instruction pointer

BP = Base pointer ("frame pointer")

adversary's goal: input a string that overwrites `fp` so that the code jumps into `win`

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void win()
{
  printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
  volatile int (*fp)();
  char buffer[64];

  fp = 0;

  gets(buffer);

  if(fp) {
      printf("calling function pointer, jumping to 0x%08x\n", fp);
      fp();
  }
}
```



adversary's goal: input a string that overwrites the saved instruction pointer so that the code jumps into `win`

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void win()
{
  printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
  char buffer[64];
  gets(buffer);
}
```

```
 [ Args to function   ]
 ----------------------
 [ saved BP, saved IP ]
 ---------------------- <-- BP would point here
 [ buffer             ]
```

modern linux has **protections** in place to prevent the attacks on the previous slides, but there are **counter-attacks** to those protections

example protections: non-executable stacks, address space layout randomization, etc.

example counter-attacks: arc-injection (“return-to-libc”), heap smashing, pointer subterfuge

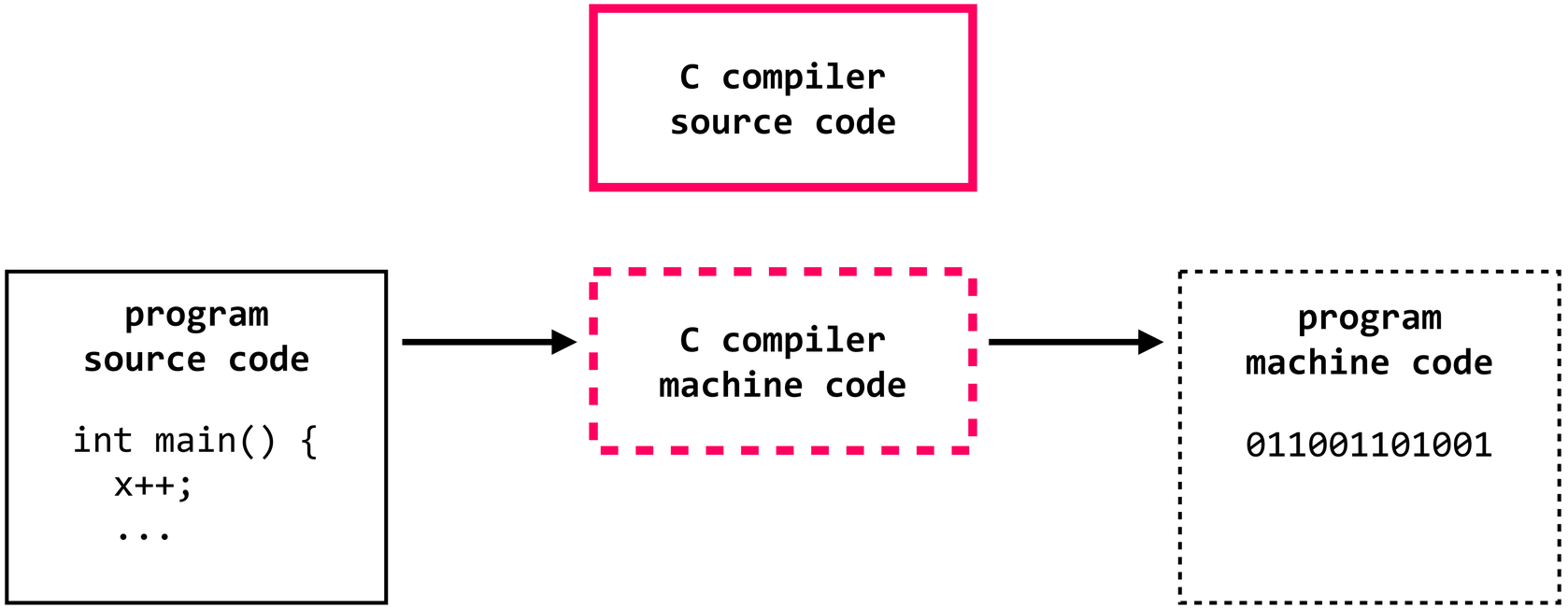
bounds-checking is one solution, but it ruins the ability to create compact C code (note the trade-off of **security vs. performance**)

```
struct record {
  int age;
  int sal;
  char name[1];
};

struct record *r;
char buf[100];
read(socket, buf, 100)
r = (struct record *)buf;
printf ("%d,%d,%s\n",r->age,r->sal,r->name);
```

for example, here is some network I/O code in C (exactly what it does doesn’t matter at all for this example). this generates very compact assembly, and takes hundreds of lines in Java.

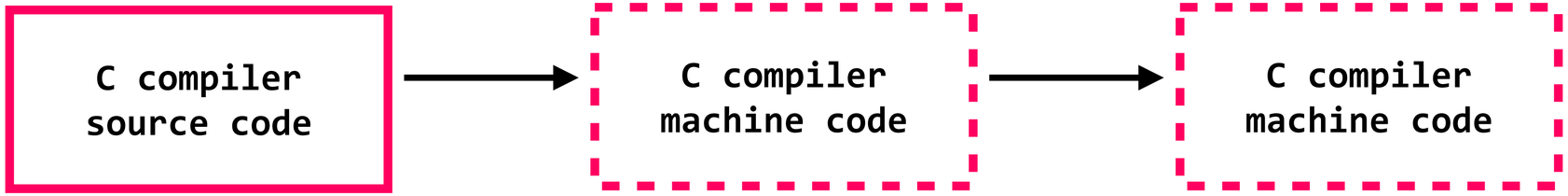
compilers: can we trust them?

compilers take source code as an input, and output machine code



compilers: can we trust them?

compilers take source code as an input, and output machine code

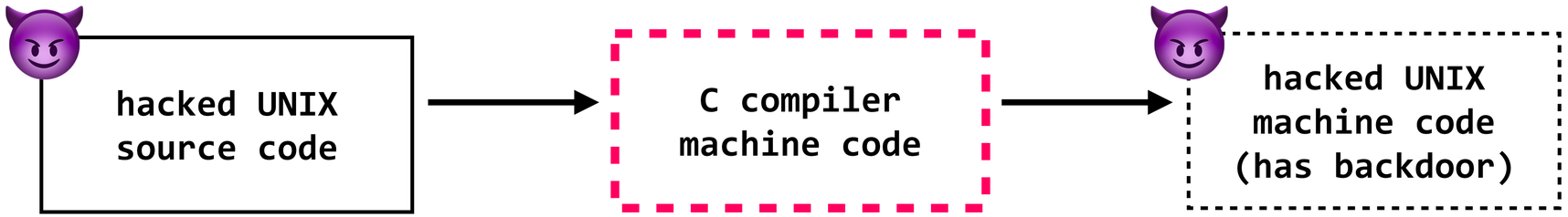


compilers: can we trust them?

compilers take source code as an input, and output machine code

compilers: can we trust them?

compilers take source code as an input, and output machine code

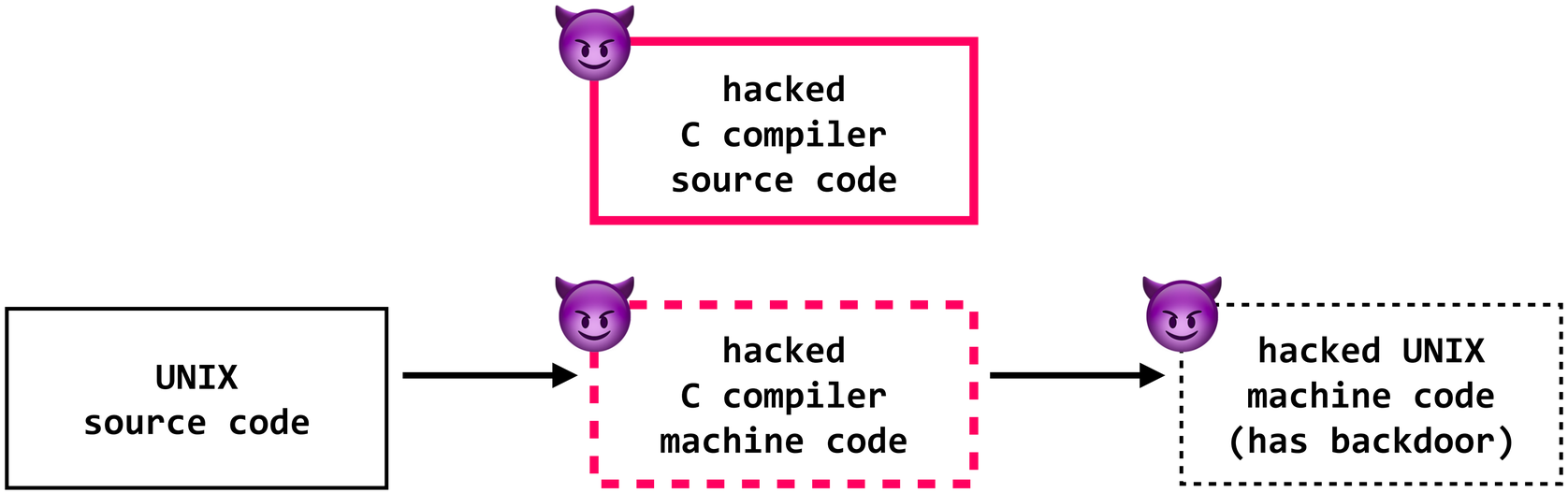


this backdoor is easily discovered in the hacked UNIX source

key point: we can determine whether source code is hacked by just reading code itself
(the code that inserts a backdoor would be obvious to someone familiar with the UNIX source)

compilers: can we trust them?

compilers take source code as an input, and output machine code



the hacked C compiler has code that
inserts a backdoor into UNIX

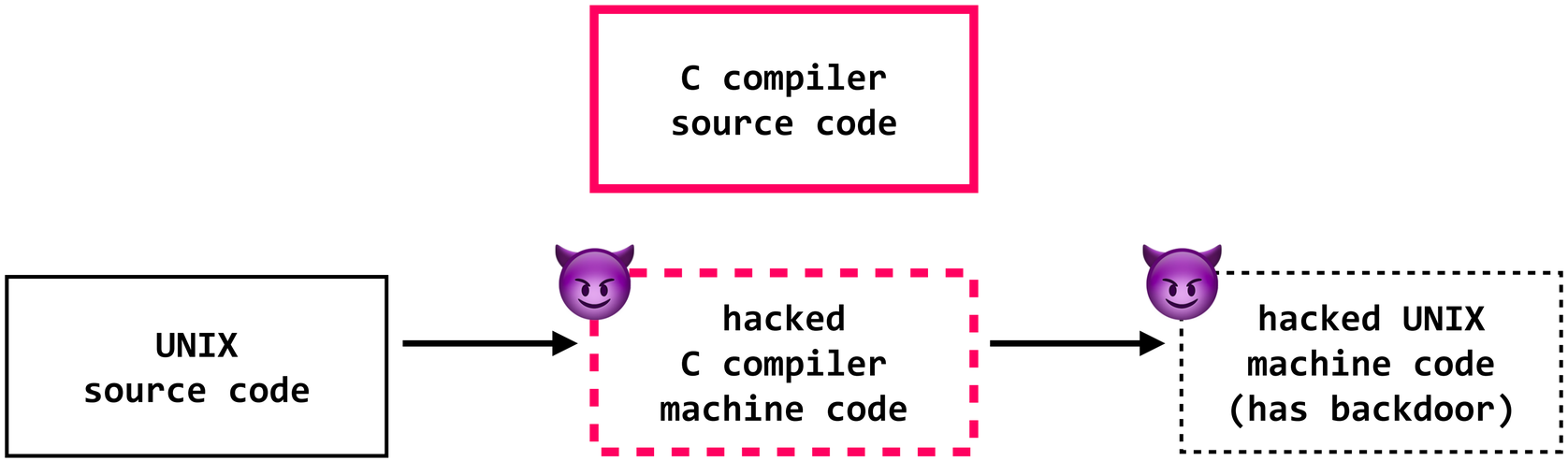
this backdoor *does not* exist in the UNIX source…

but it does exist in the hacked C compiler source

key point: we can determine whether source code is hacked by just reading code itself
(the code that inserts a backdoor would be obvious to someone familiar with the UNIX source)

compilers: can we trust them?

compilers take source code as an input, and output machine code



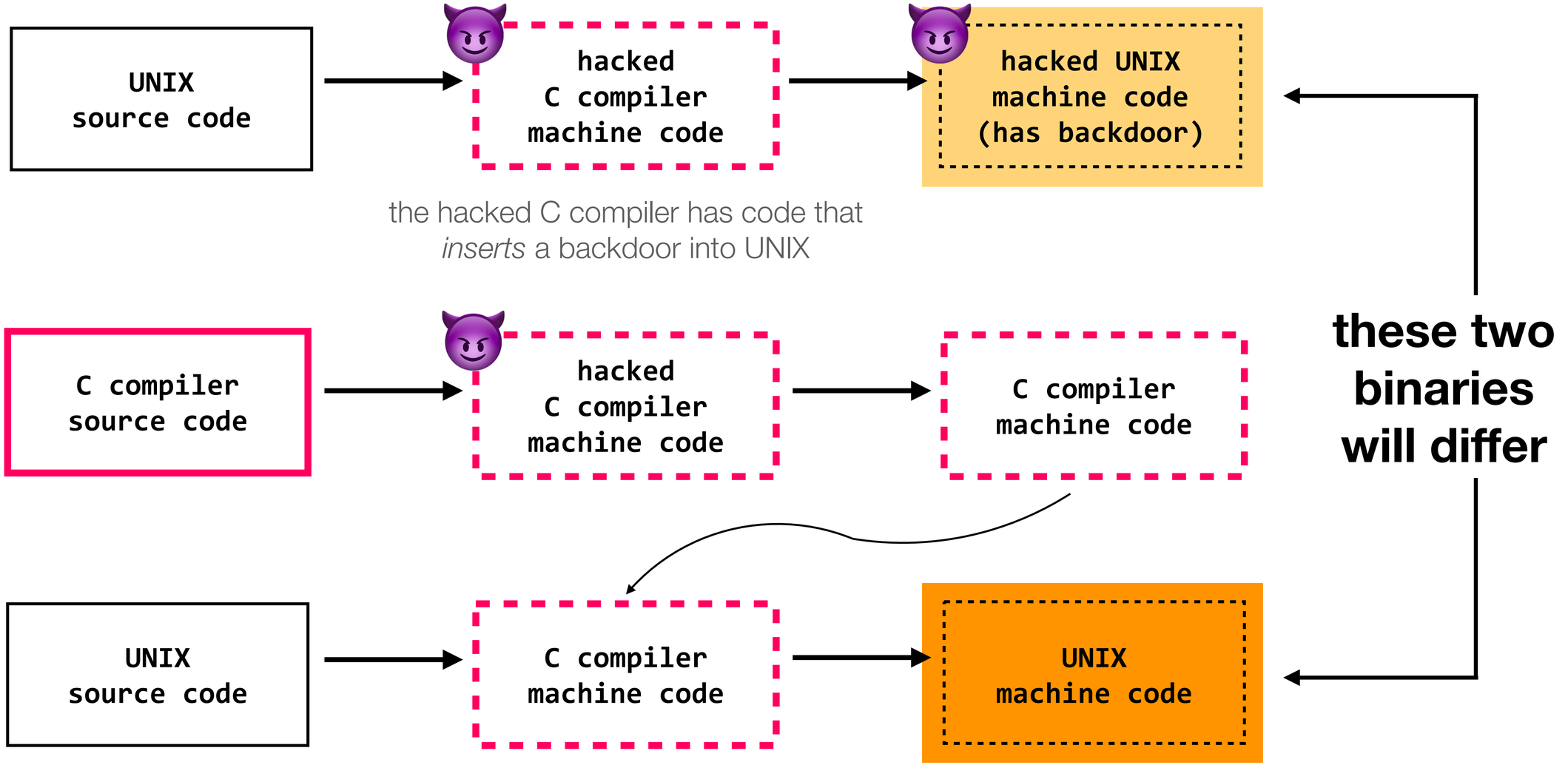
the hacked C compiler has code that
inserts a backdoor into UNIX

suppose the adversary lies, and tells you that the clean C compiler source is what generated the hacked C compiler; can you detect this lie?

key point: we can determine whether source code is hacked by just reading code itself
(the code that inserts a backdoor would be obvious to someone familiar with the UNIX source)

compilers: can we trust them?

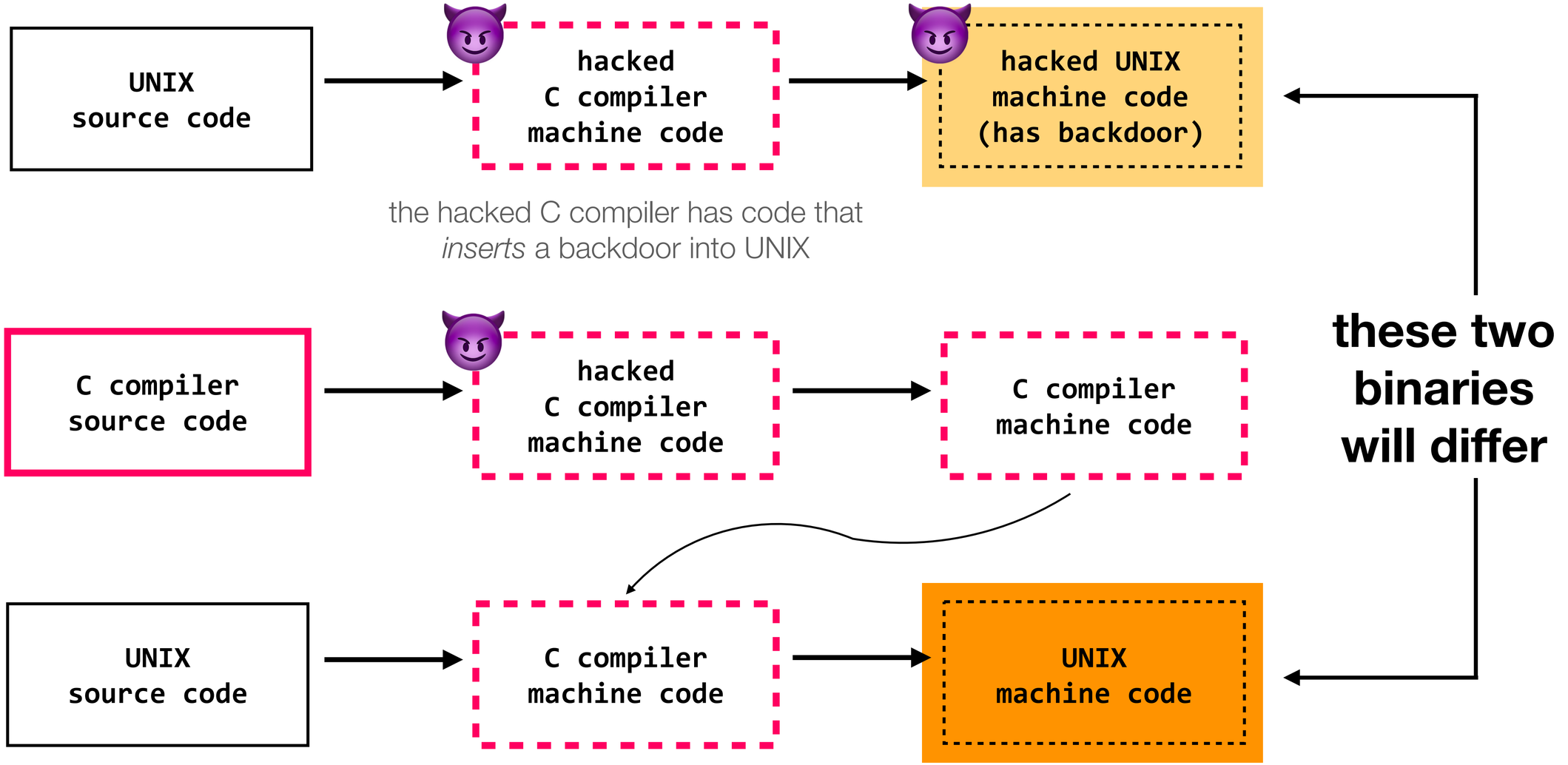
compilers take source code as an input, and output machine code



key point: we can determine whether source code is hacked by just reading code itself
(the code that inserts a backdoor would be obvious to someone familiar with the UNIX source)

compilers: can we trust them?

compilers take source code as an input, and output machine code

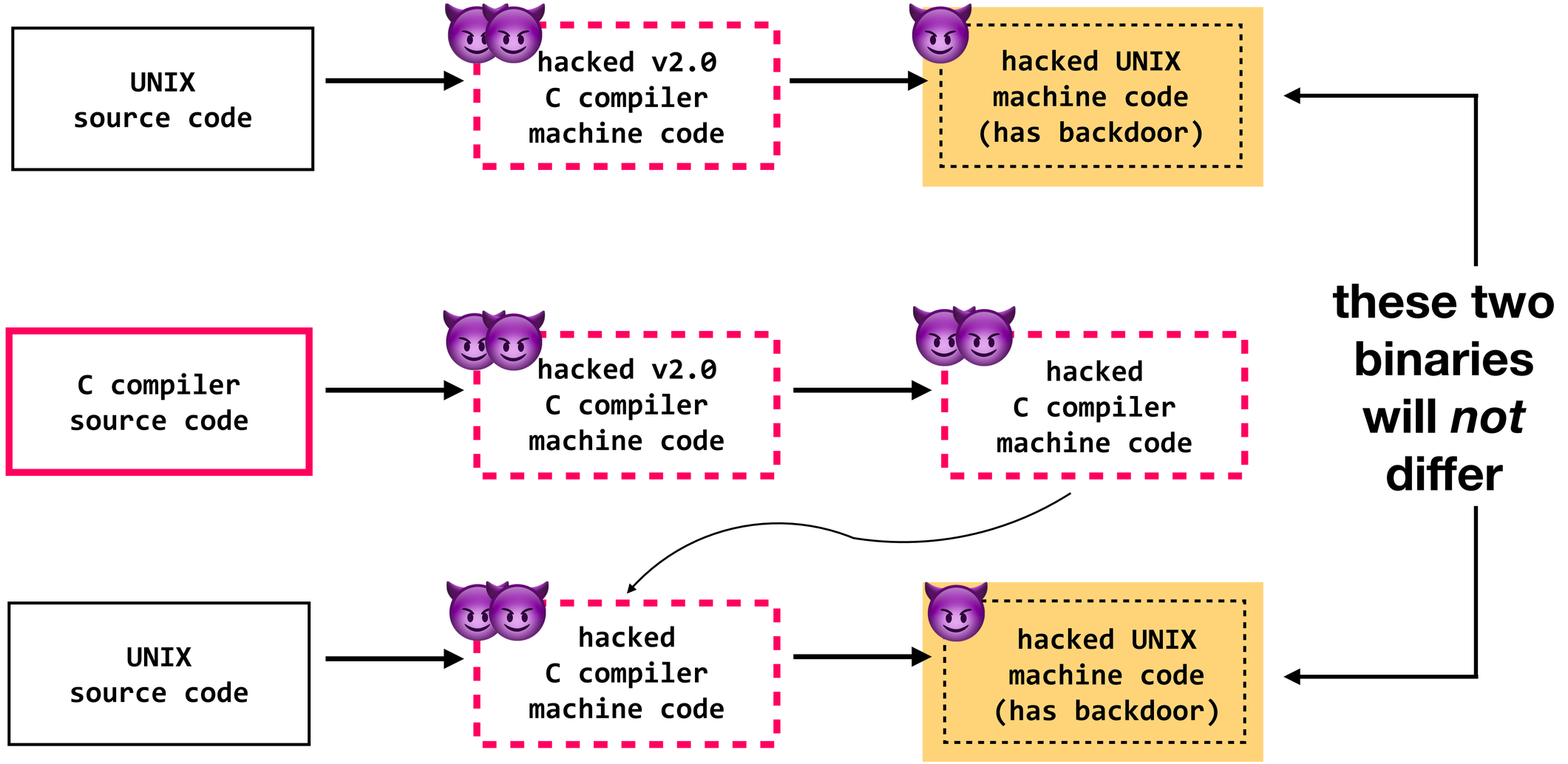


key point: we can detect a hacked compiler by recompiling a clean compiler, using that to compile UNIX, and testing the output against what the hacked compiler produced

compilers: can we trust them?

compilers take source code as an input, and output machine code

the hacked v2.0 C compiler has code that *inserts* a backdoor into UNIX *and* code to insert backdoor-inserting code into C compilers



key point: we can detect ***the original*** hacked compiler by recompiling a clean compiler, using that to compile UNIX, and testing the output against what the hacked compiler produced

compilers: can we trust them?

compilers take source code as an input, and output machine code

REFERENCES

1. Bobrow, D.G., Burchfiel, J.D., Murphy, D.L., and Tomlinson, R.S. TENEX, a paged time-sharing system for the PDP-10. *Commun. ACM 15*, 3 (Mar. 1972), 135–143.
2. Kernighan, B.W., and Ritchie, D.M. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, N.J., 1978.
3. Ritchie, D.M., and Thompson, K. The UNIX time-sharing system. *Commun. ACM 17*, (July 1974), 365–375.
4. Unknown Air Force Document.

Karger, P.A., and Schell, R.R.
Multics Security Evaluation: Vulnerability Analysis
ESD-TR-74-193, Vol II, June 1974, page 52

low-level attacks can be insidious; as we implement solutions, there are often counter-attacks, and many solutions come at the cost of performance

however, just because we can't achieve perfect security does not mean that we cannot make progress; more sophisticated attacks are often more difficult for adversaries to carry out, and in some cases might not be worth the effort

today's lecture + tomorrow's recitation should not stop you from ever touching a computer again

while **thompson's "hack"** (attack?) illustrates to us that, to some extent, we cannot trust code we didn't write ourselves, it also advocates for **policy-based solutions** rather than technology-based