PalanTir: Optimizing Attack Provenance with Hardware-enhanced System Observability

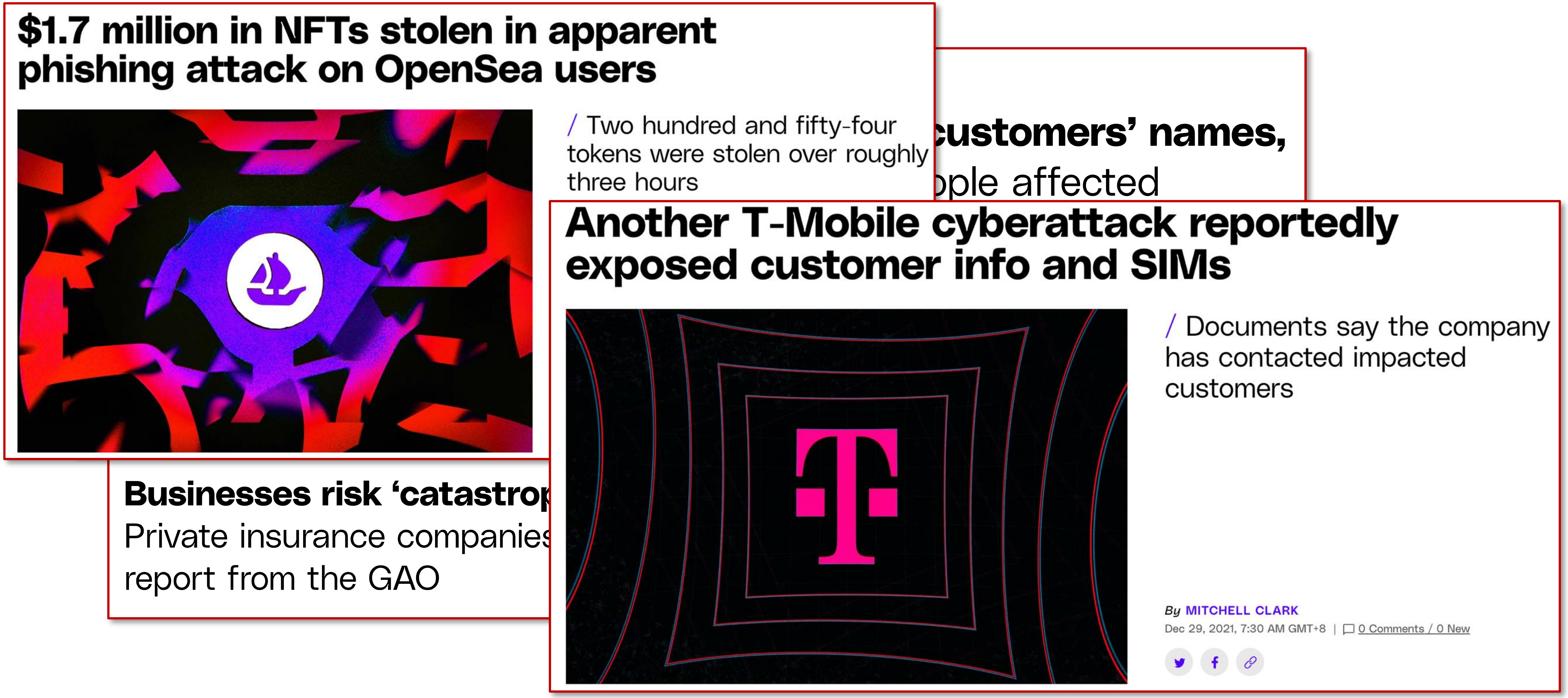
Jun Zeng[*], ***Chuqi Zhang***[*], *and Zhenkai Liang*

ACM CCS, November 2022

Los Angeles, U.S.A.

NUS
National University of Singapore

Advanced Cyber Attacks in Enterprises



$1.7 million in NFTs stolen in apparent phishing attack on OpenSea users

/ Two hundred and fifty-four tokens were stolen over roughly three hours

customers' names,

ople affected

Businesses risk 'catastrop

Private insurance companies

report from the GAO

Another T-Mobile cyberattack reportedly exposed customer info and SIMs

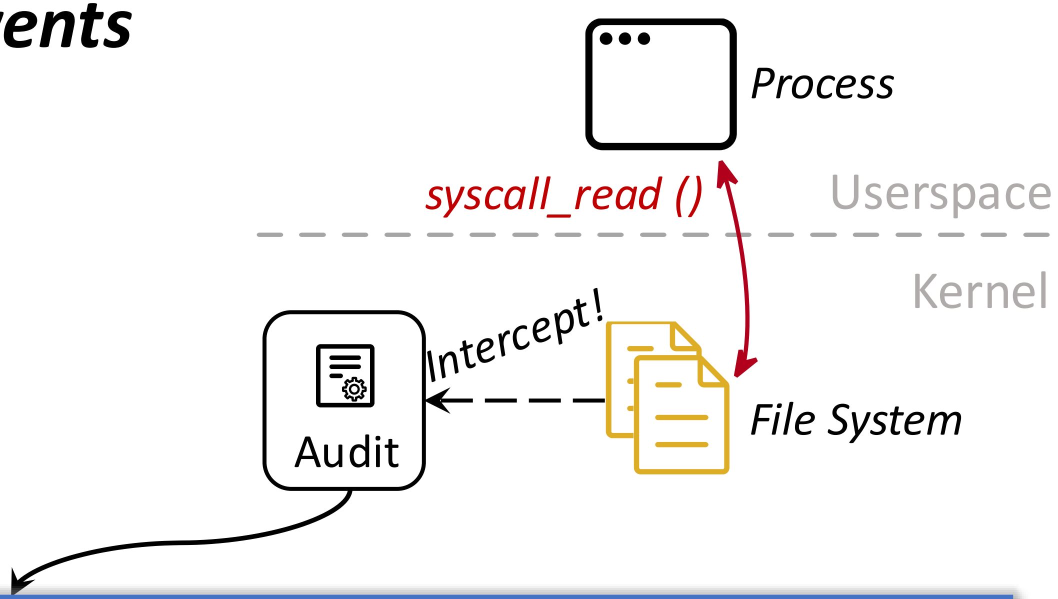
/ Documents say the company has contacted impacted customers

By MITCHELL CLARK

Dec 29, 2021, 7:30 AM GMT+8 | 0 Comments / 0 New

System Auditing: the Foundation of Attack Investigation

- System auditing records ***OS-level events*** (system entity ***interactions***)
  - e.g., system call



```
syscall=read exit=0x100 a0=0x3 a1=… ... pid=12566 auid=chuqiz sess=6150
type=SYSCALL msg=audit(30/01/22 12:56:15.383:98866813) arch=x86_64
```

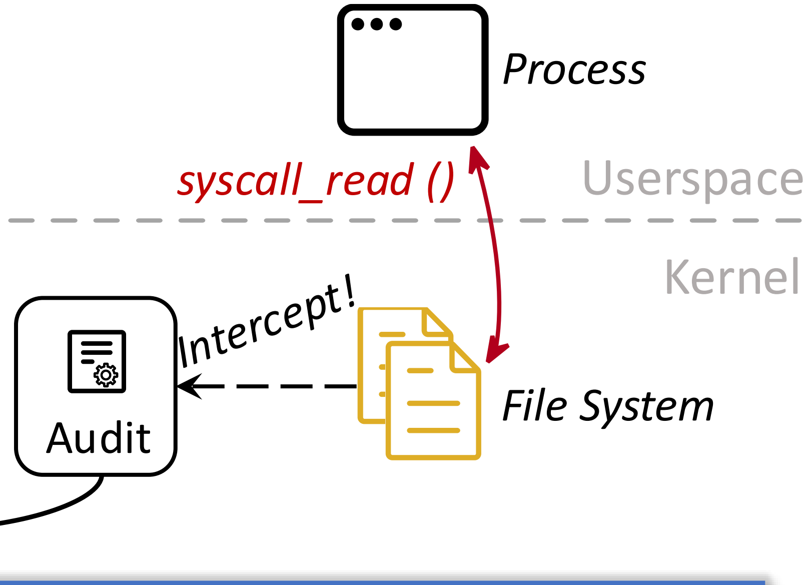
System Auditing: the Foundation of Attack Investigation

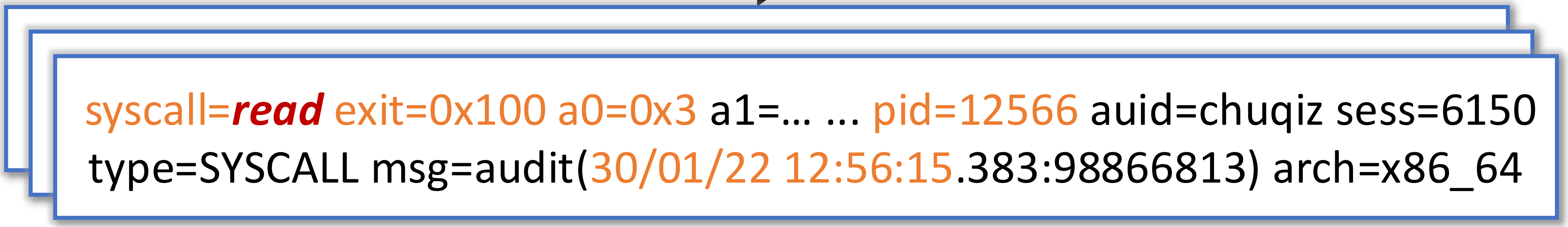
- System auditing records ***OS-level events*** (system entity ***interactions***)
  - e.g., system call

- Audit logs can be used for:

  ✓ ***Root cause analysis***

  ✓ ***Ramification discovery***

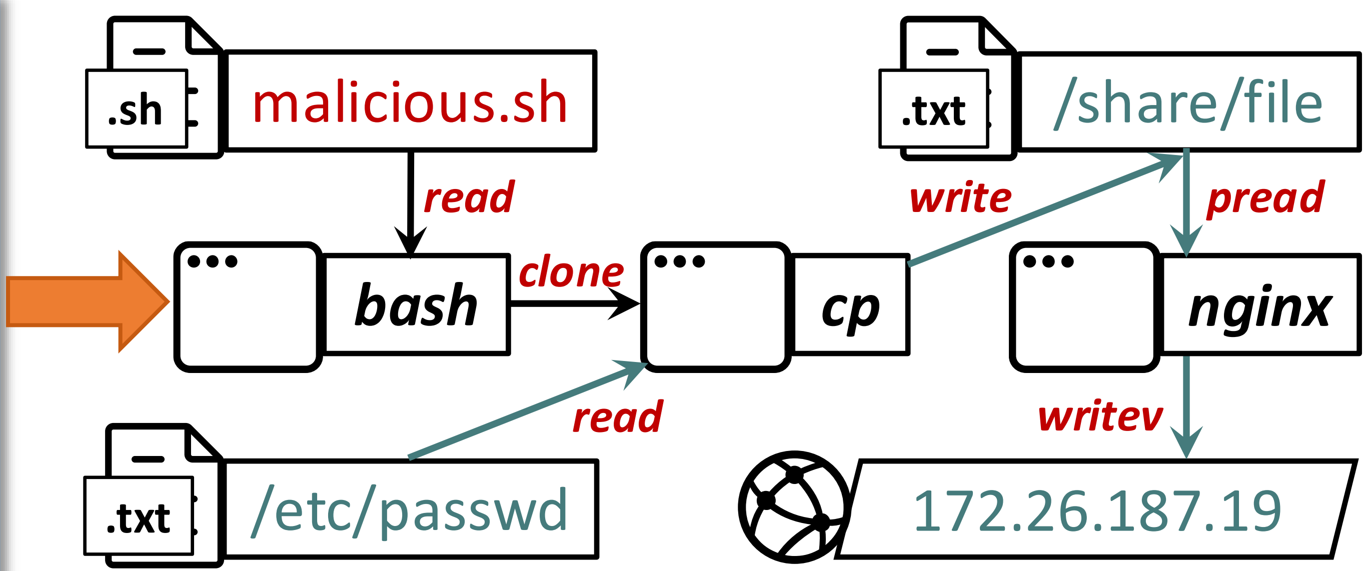




```
syscall=read exit=0x100 a0=0x3 a1=... ... pid=12566 auid=chuqiz sess=6150
type=SYSCALL msg=audit(30/01/22 12:56:15.383:98866813) arch=x86_64
```

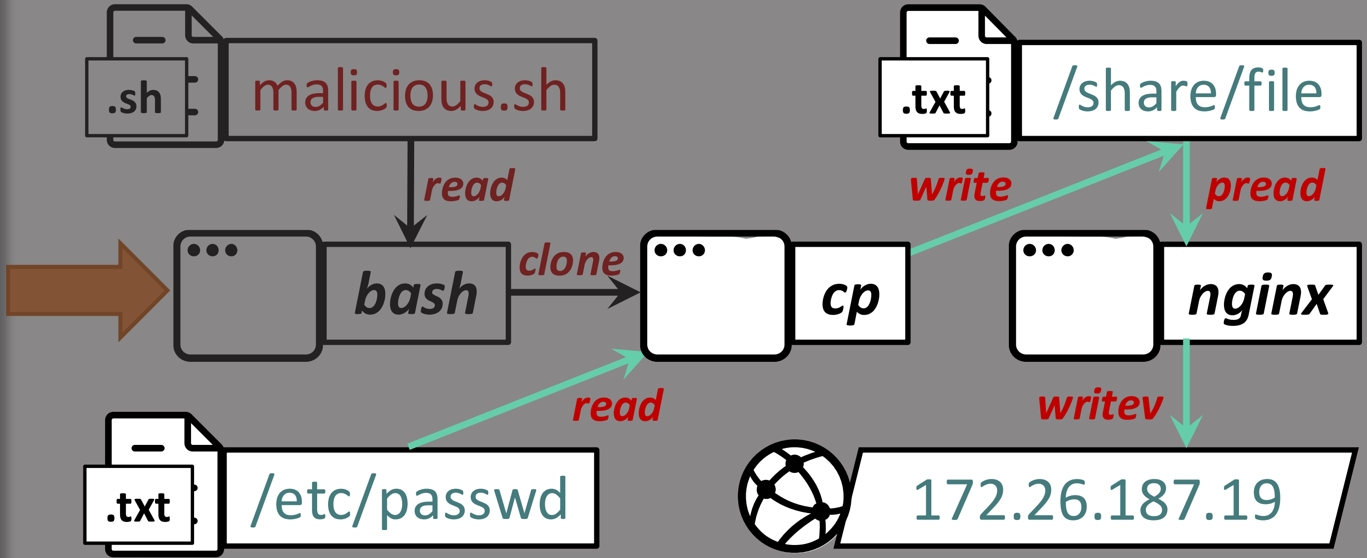
Provenance Graph from Audit Logs

```
…
1. bash, read, malicious.sh
2. bash, clone, cp
3. cp, read, /etc/passwd
4. cp, write, /share/file
5. nginx, pread, /share/file
6. nginx, writev, 172.26.187.19
…
```



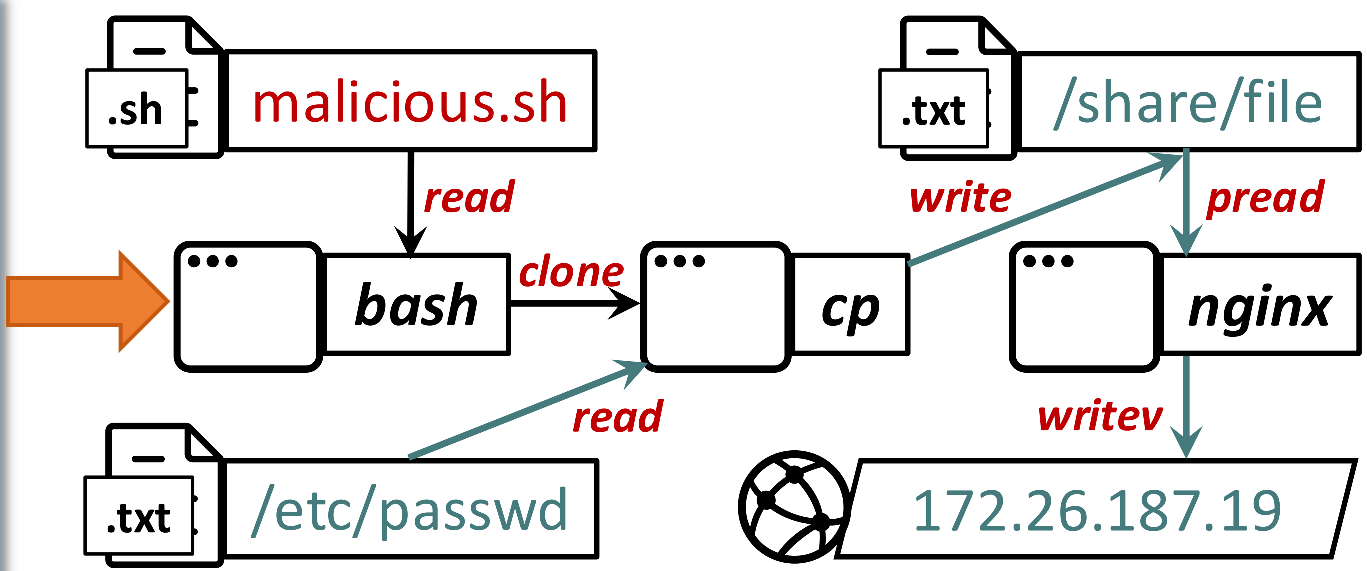
Provenance Graph from Audit Logs

```
…
1. bash, read, malicious.sh
2. bash, clone, cp
3. cp, read, /etc/passwd
4. cp, write, /share/file
5. nginx, pread, /share/file
6. nginx, writev, 172.26.187.19
…
```



Provenance Graph from Audit Logs

```
…
1. bash, read, malicious.sh
2. bash, clone, cp
3. cp, read, /etc/passwd
4. cp, write, /share/file
5. nginx, pread, /share/file
6. nginx, writev, 172.26.187.19
…
```



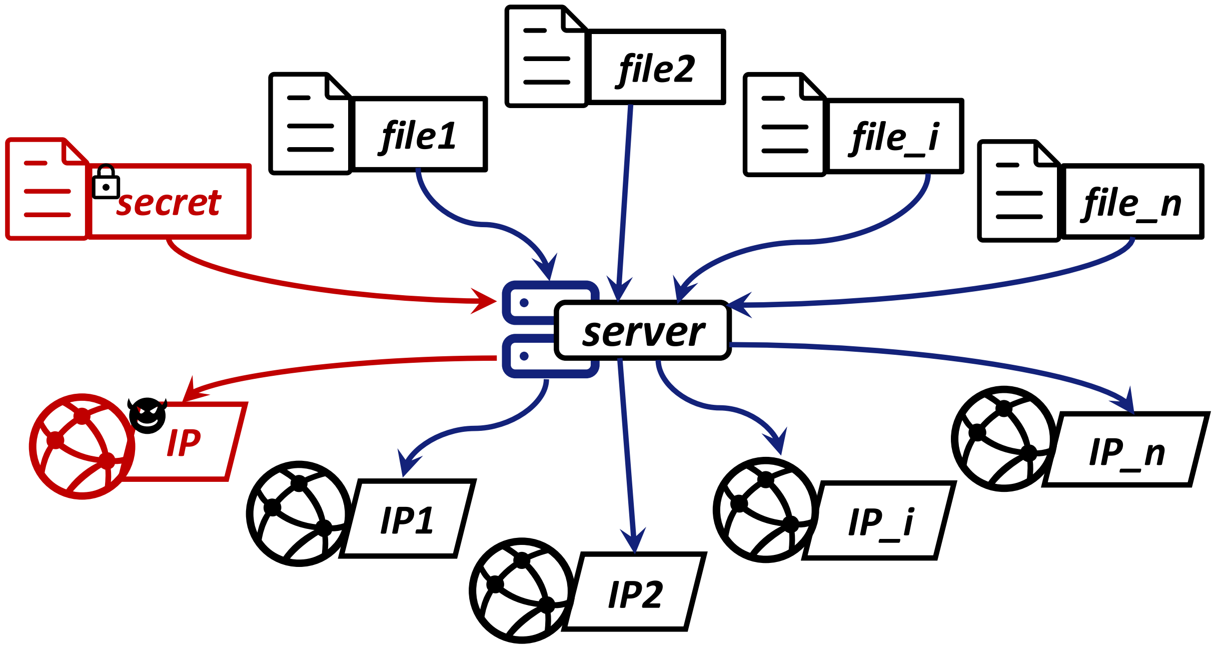
✓***Provenance Graph*** constructs the ***overall attack scenario*** by ***combining*** historic audit logs***!***

Challenges of Provenance Tracking

Simplified code for a ***web server*** *program*

```
// handle connections
while ((connection_t *) conn) {
    request_t *r = conn->req;
     // handle requested file
    int fd = open(r->req_file);
    read(fd, r->buf, …);
    send(conn->sock_fd, r->buf, …);
}
…
```

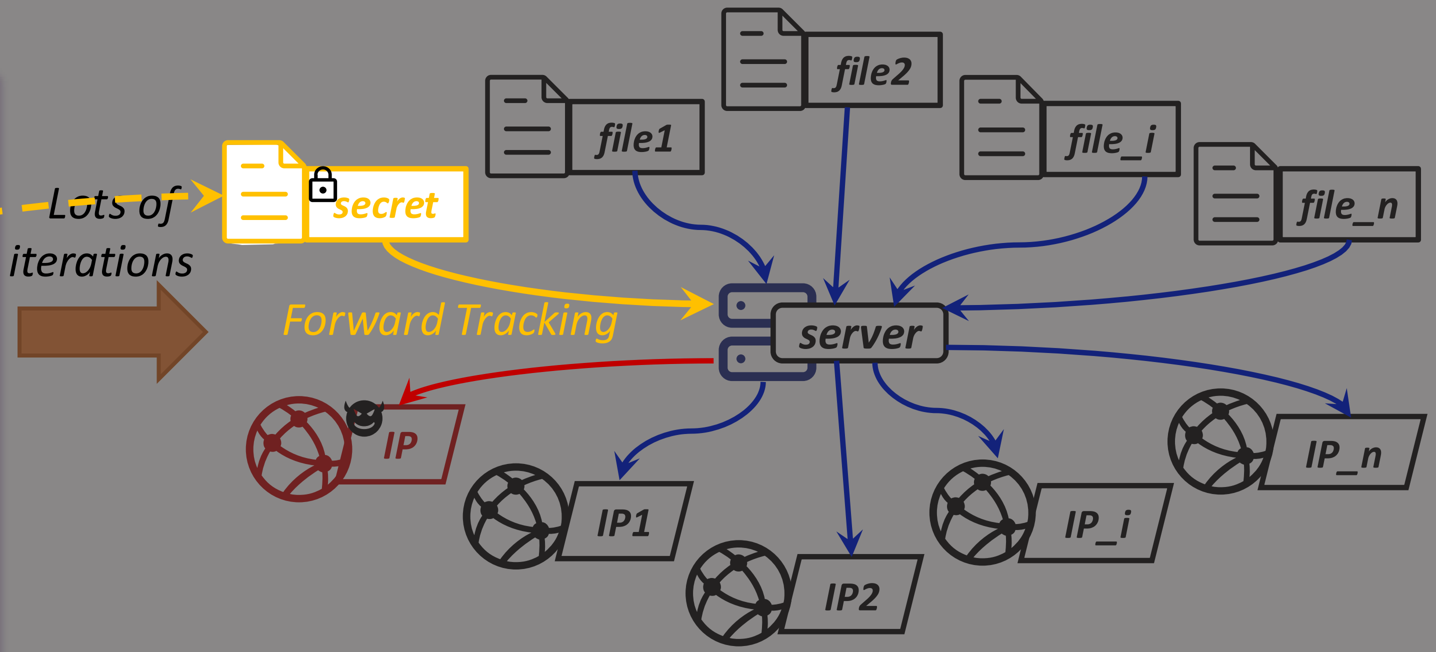
Lots of iterations



Challenges of Provenance Tracking

*Simplified code for a **web server** program*

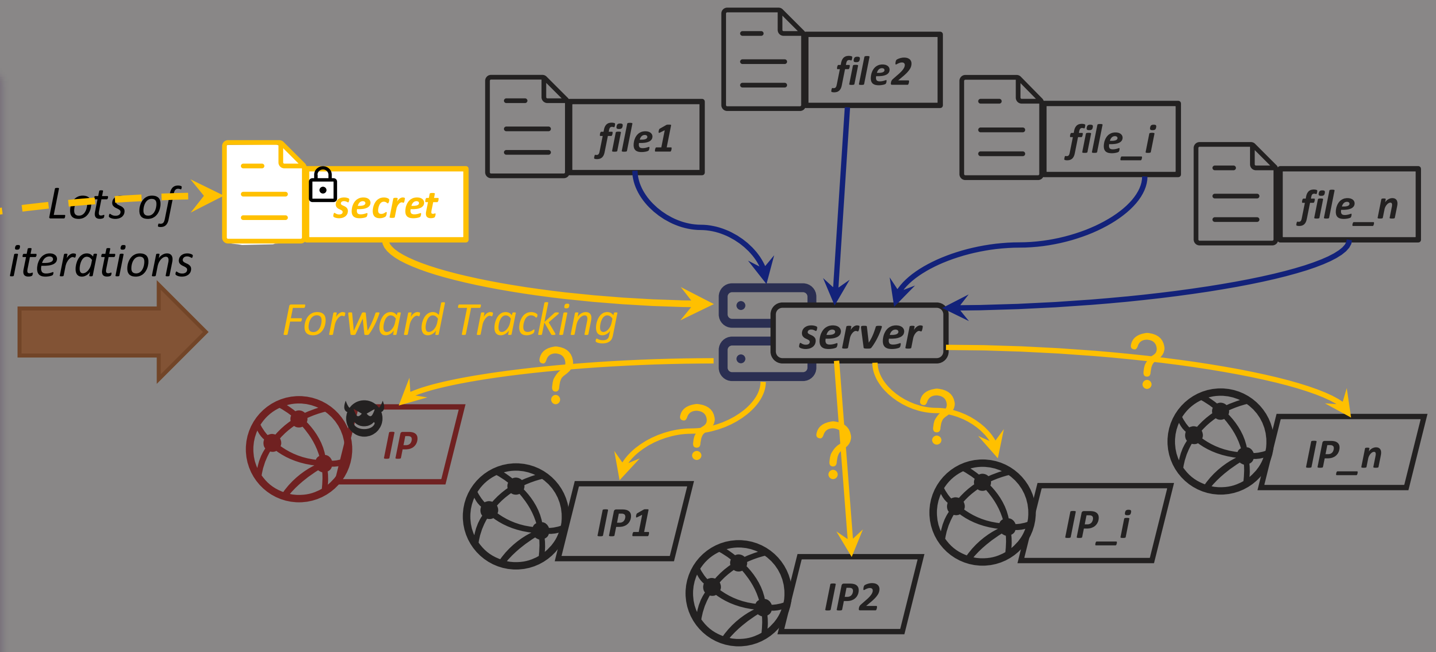
```
// handle connections
while ((connection_t *) conn) {
    request_t *r = conn->req;
    // handle requested file
    int fd = open(r->req_file);
    read(fd, r->buf, …);
    send(conn->sock_fd, r->buf, …);
}
…
```



Challenges of Provenance Tracking

*Simplified code for a **web server** program*

```
// handle connections
while ((connection_t *) conn) {
    request_t *r = conn->req;
    // handle requested file
    int fd = open(r->req_file);
    read(fd, r->buf, …);
    send(conn->sock_fd, r->buf, …);
}
…
```



CAN NOT identify the correct ***descendant.***

✗ No conclusion** of **TRUE provenance.

Dependency Explosion Problem !

Related Work

- ***Execution Unit Partitioning*** [NDSS’13, Security’16, NDSS’21, …]:
  - Partition program into units by instrumentation or built-in application logs
  - Intrusive to program or error-prone units
- ***Causality Inference*** [ASPLOS’16, NDSS’18, …]:
  - Train a causality model based on dual execution to infer true dependencies
  - Inadequate for high-concurrency programs
- ***Record-and-Replay*** [CCS’17, Security’18, …]:
  - Record non-deterministic program behaviors and replay with taint analysis
  - Fine-grained but intrusive to program, and incur high overhead

Related Work

- ***Execution Unit Partitioning*** [NDSS’13, Security’16, NDSS’21, …]:
  - P
- ***Ca***
- ***Record-and-Replay*** [CCS’17, Security’18, …]:
  - Record non-deterministic program behaviors and replay with taint analysis
  - Fine-grained but intrusive to program, and incur high overhead

💡 ***Ideal Solution:***

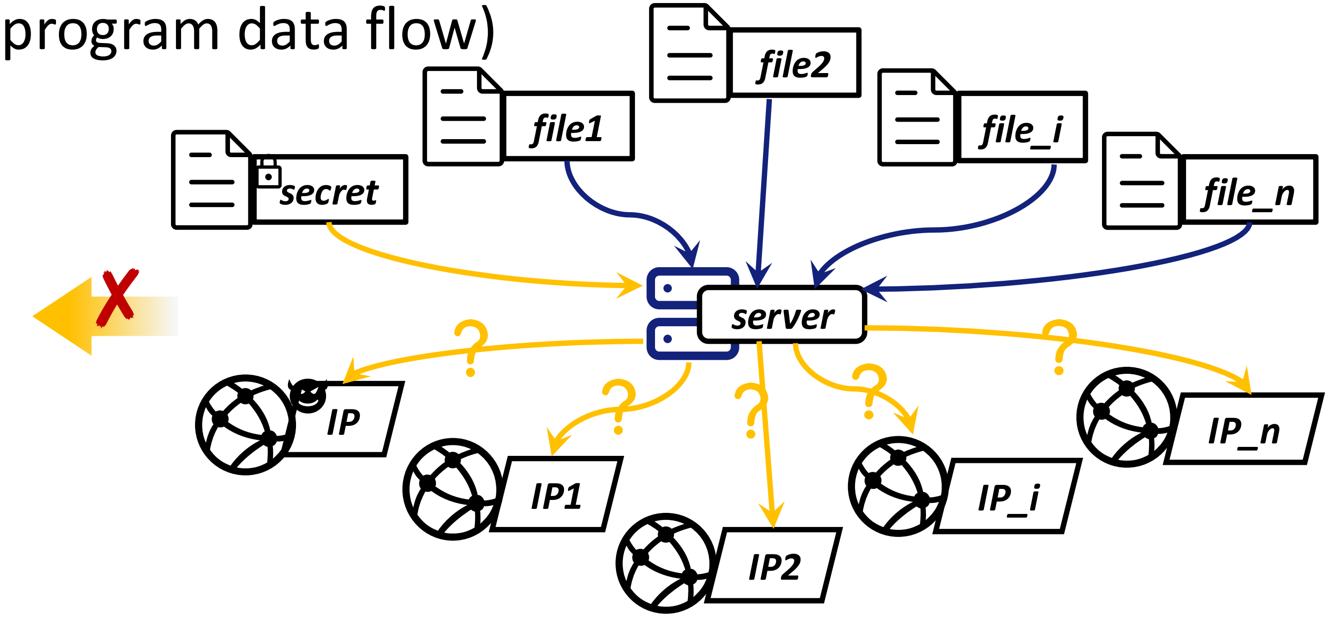
- ***Non-intrusive*** to program (i.e., instrumentation free)
- Fine-grained (i.e., ***pinpoint dependency***) provenance

Motivation: Enhance Observability

- Audit log ONLY records OS-level events => ***coarse-grained provenance***
  - ✗ ***NO fine-grained provenance*** (program data flow)

✗

```
while ((connection_t *) conn) {
  request_t *r = conn->req;
  int fd = open(r->req_file);
  read(fd, r->buf, …);
  send(conn->sock_fd, r->buf, …);
}
…
```

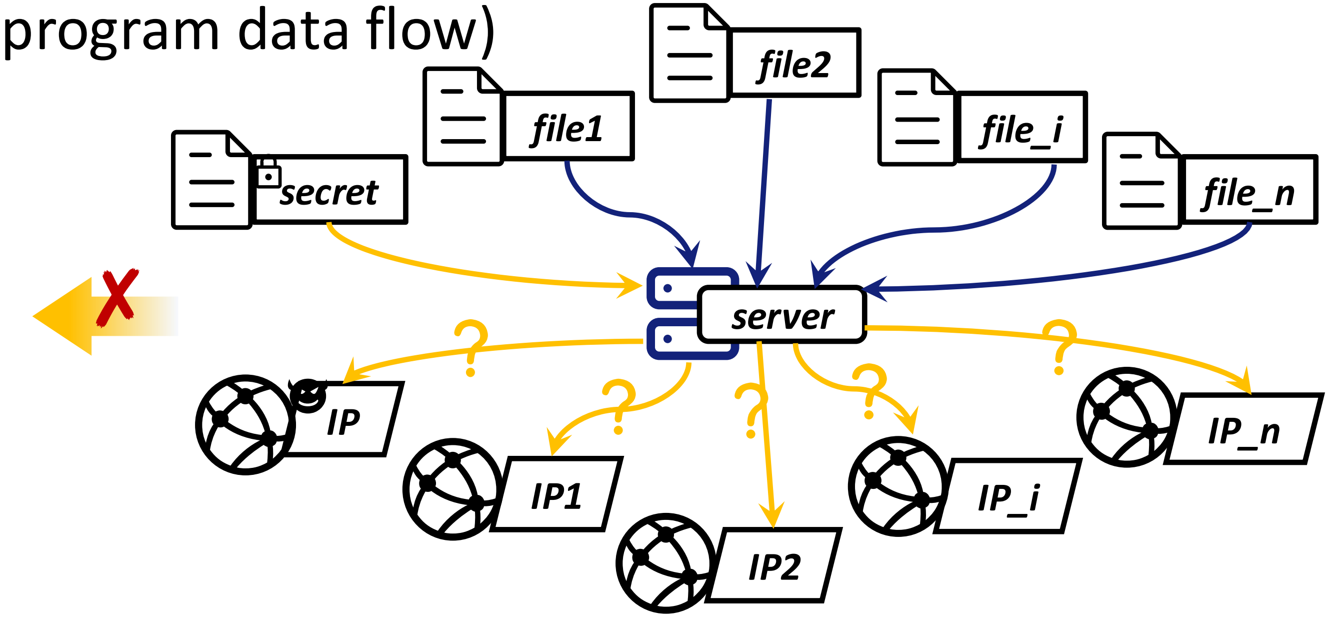


Motivation: Enhance Observability

- Audit log ONLY records OS-level events => ***coarse-grained provenance***

  ✗ ***NO fine-grained provenance*** (program data flow)

```
while ((connection_t *) conn) {
  request_t *r = conn->req;
  int fd = open(r->req_file);
  read(fd, r->buf, …);
  send(conn->sock_fd, r->buf, …);
}
…
```

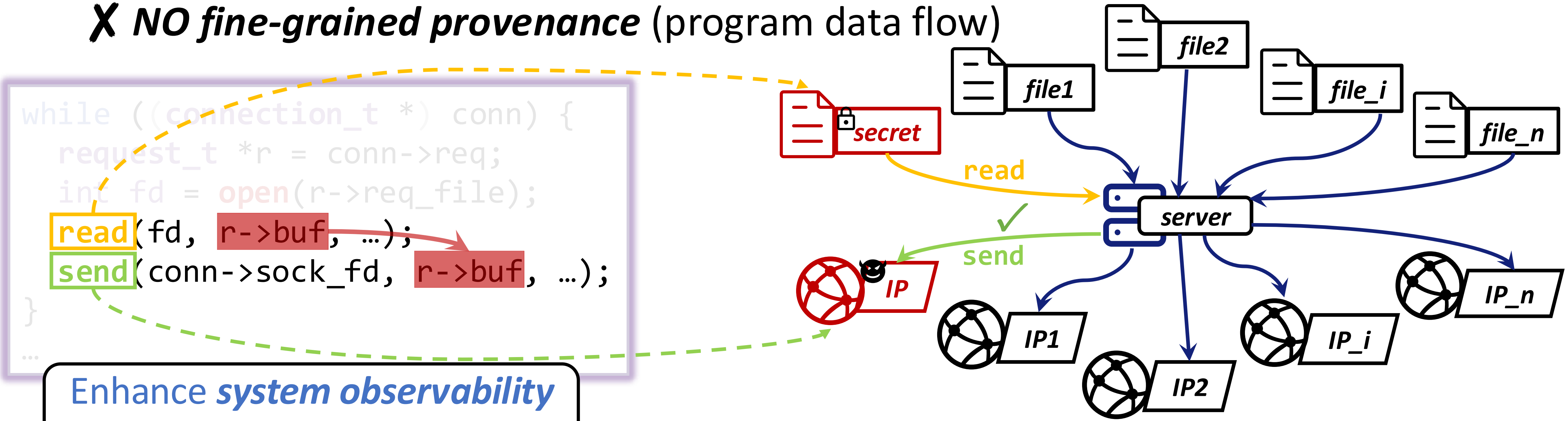


- 💡***Motivation***: Enhance audit logs with program data flow to achieve ***high system observability***

Motivation: Enhance Observability

- Audit log ONLY records OS-level events => ***coarse-grained provenance***

  ✗ ***NO fine-grained provenance*** (program data flow)



```
while ((connection_t *) conn) {
  request_t *r = conn->req;
  int fd = open(r->req_file);
  read(fd, r->buf, …);
  send(conn->sock_fd, r->buf, …);
}
…
```

Enhance ***system observability***
(with program data flow)

- 💡***Motivation***: Enhance audit logs with program data flow to achieve ***<u>high system observability</u>***

Fine-grained Provenance

- ***Ideal observability:*** Enhance the provenance with ***syscall-to-syscall taints*** (i.e., instruction-level data flow)
- Enhance observability and resolve fine-grained provenance:

```
while ((connection_t *) conn) {
  request_t *r = conn->req;
  int fd = open(r->req_file);
  read(fd, r->buf, …);
  send(conn->sock_fd, r->buf, …);
}
…
```

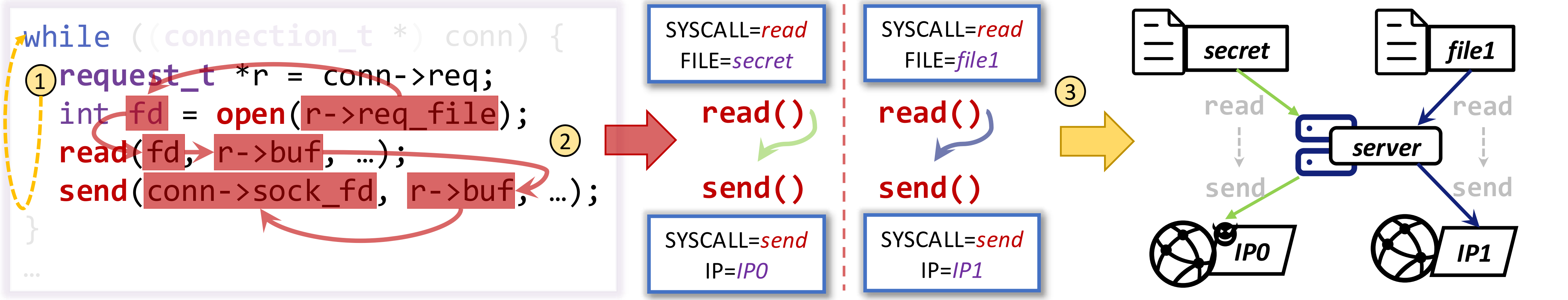
Fine-grained Provenance

- ***Ideal observability:*** Enhance the provenance with ***syscall-to-syscall taints*** (i.e., instruction-level data flow)
- Enhance observability and resolve fine-grained provenance:

① ***Control flow tracing:*** trace runtime execution history

② ***Data flow analysis:*** recover syscall-to-syscall taints

③ ***Optimization:*** incorporate audit logs with the taints



```
while ((connection_t *) conn) {
  request_t *r = conn->req;
  int fd = open(r->req_file);
  read(fd, r->buf, …);
  send(conn->sock_fd, r->buf, …);
}
…
```

SYSCALL=*read*
FILE=*secret*

SYSCALL=*send*
IP=*IP0*

SYSCALL=*read*
FILE=*file1*

SYSCALL=*send*
IP=*IP1*

Our Key Idea

① Control flow tracing

***Online** program runtime recording*

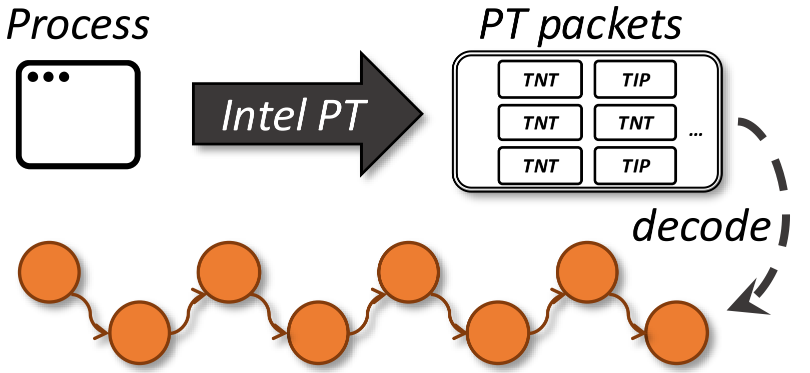
Insight: Hardware Tracing

=> Intel® Processor Tracing (PT)

to trace control flow transfer

✓ ***Trivial** runtime overhead*

✓ ***Non-intrusive** to program*



Execution Trace: A sequence of basic blocks

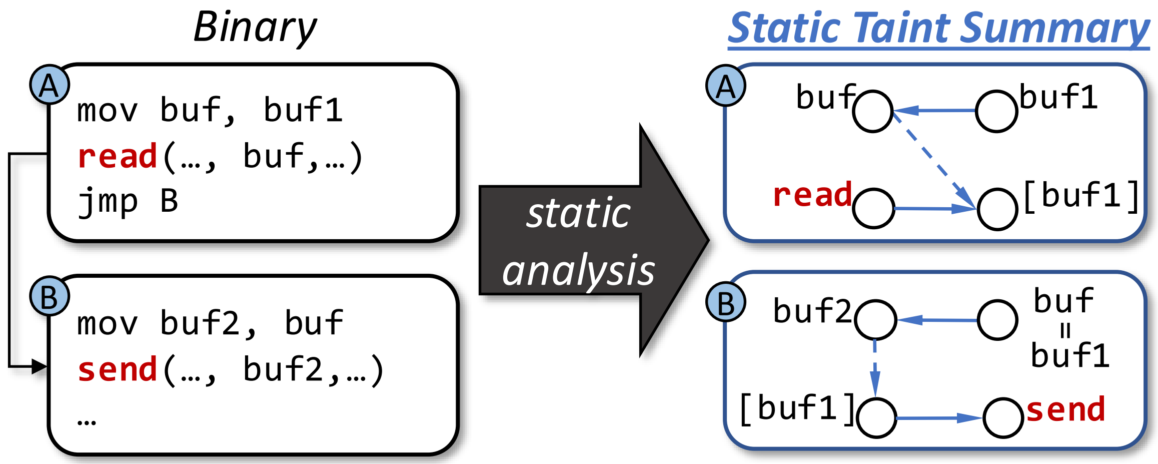
② Data flow analysis

***Offline** computationally expensive analysis*

Insight: Static Taint Summary

*=> Pre-summarize **taint propagation logic** per basic block via **static binary analysis***

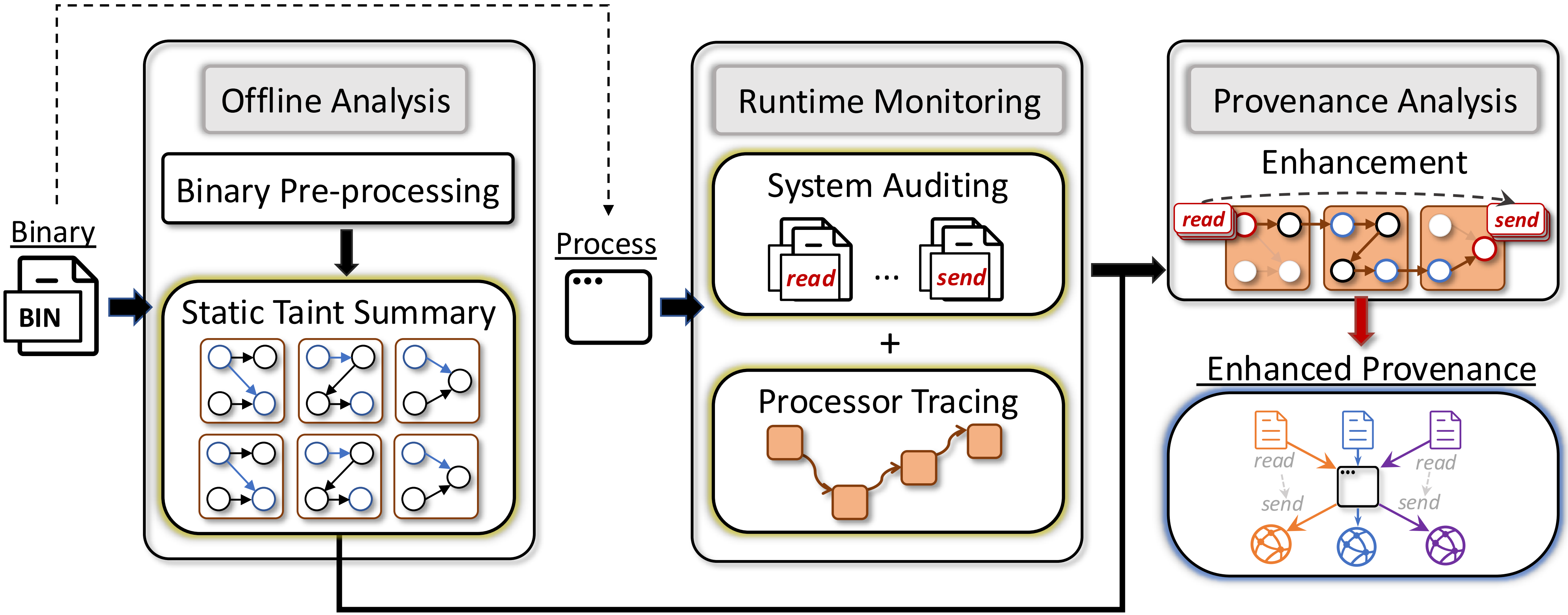
✓ ***Segregate** offline analysis cost*



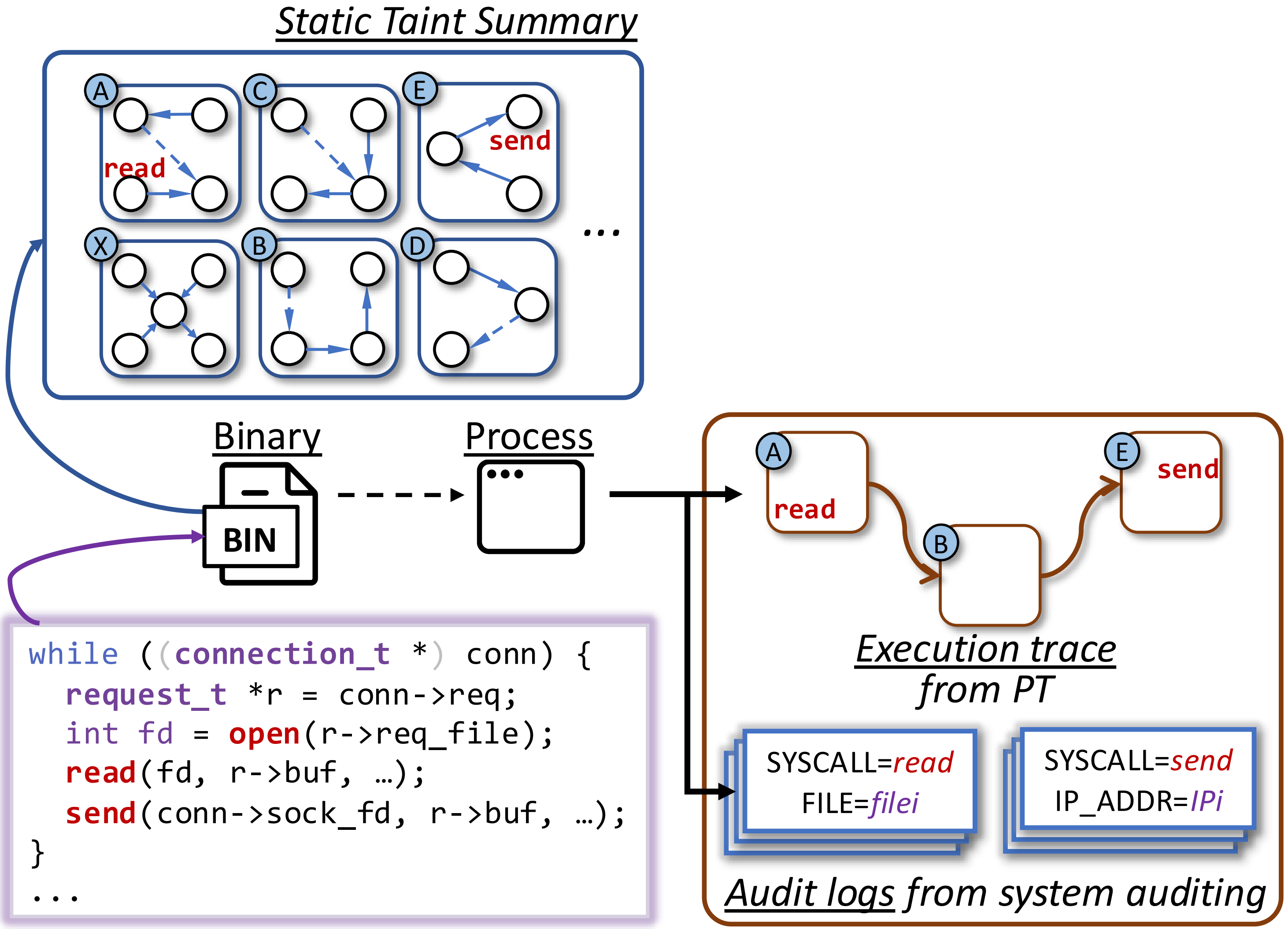
```
mov buf, buf1
read(…, buf,…)
jmp B
```

```
mov buf2, buf
send(…, buf2,…)
…
```

PALANTIR: System Overview



Running Example: Provenance Enhancement

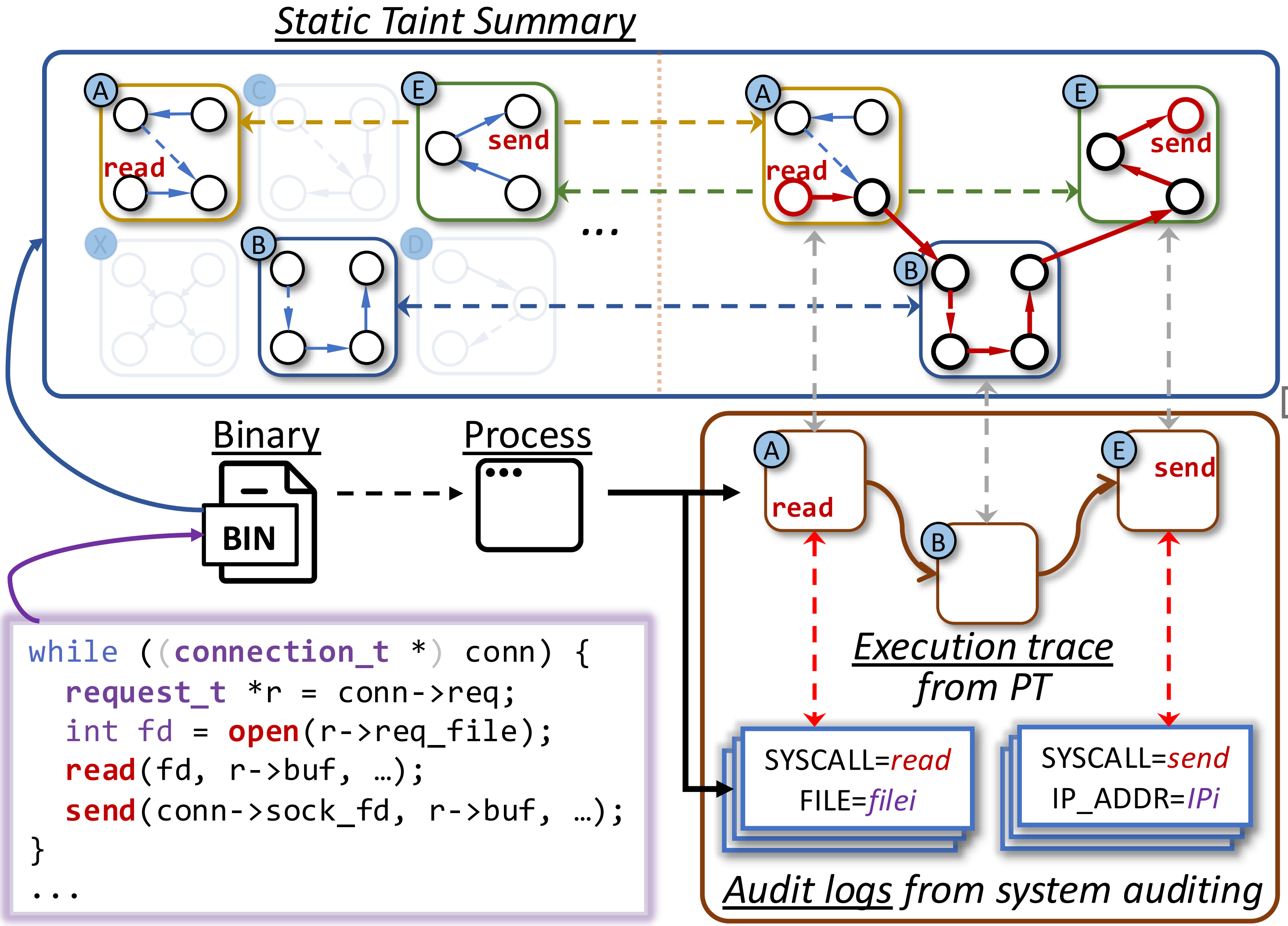


```
while ((connection_t *) conn) {
  request_t *r = conn->req;
  int fd = open(r->req_file);
  read(fd, r->buf, …);
  send(conn->sock_fd, r->buf, …);
}
...
```

SYSCALL=read
FILE=filei

SYSCALL=send
IP_ADDR=IPi

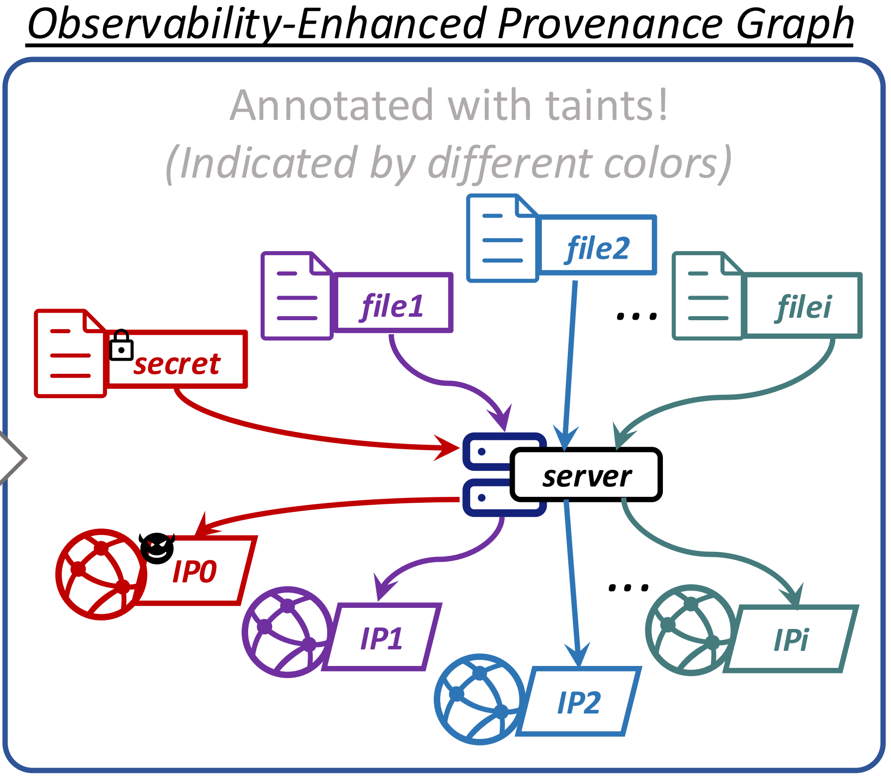
Running Example: Provenance Enhancement



```
while ((connection_t *) conn) {
  request_t *r = conn->req;
  int fd = open(r->req_file);
  read(fd, r->buf, …);
  send(conn->sock_fd, r->buf, …);
}
...
```

```
SYSCALL=read
FILE=filei
```

```
SYSCALL=send
IP_ADDR=IPi
```


Observability-Enhanced Provenance Graph
Annotated with taints!
(Indicated by different colors)
secret
file1
file2
...
filei
server
IP0
IP1
...
IP2
IPi

✓ ***Fine-grained Provenance*** is ***optimized*** with ***instruction-level Observability***

Evaluation Settings

- ***Evaluation Aspects***
  - ***How efficient*** is PALANTIR at attack investigation?
  - ***What*** is the ***runtime performance*** of PALANTIR?

- ***Evaluation Dataset***
  - ***Four real-world cyber-attacks*** simulated in a testbed:

    Watering-hole, Data Leakage, Insider Threat, and Phishing Email
  - ***SPEC CPU 2006*** benchmarks & real-world ***common programs***

Attack Investigation

- ***Identify true causality*** among system events and dependencies

Attack Scenario	Program	Audit Logs	PT Packets	Instructions	Investigation Time (s)
Watering Hole	Wget	10,256	62,175,669	1,329,321,333	12.05
	Nginx	1,830	401,708	5,160,695	2.86
Data Leakage	Curl	10,309	1,882,471	17,516,456	9.39
	Pure-ftpd	25,562	21,402,396	2,833,740,916	2.85
Insider Threat	Cp	1,814	134,161	1,048,907	0.20
	Lighttpd	4,800	499,995	5,448,715	0.58
Phishing Email	Sendmail	29,433	7,488,895	120,264,352	18.09

✓ PALANTIR achieves a high efficiency in attack investigation

Attack Investigation - Comparison

- ***Compare*** with Dynamic Information Flow Tracking (DIFT)-based system

Attack Scenario	Program	Investigation Time (s)	
		PALANTIR	RTAG
Watering Hole	Wget	12.05	67.93
	Nginx	2.86	37.50
Data Leakage	Curl	9.39	50.03
	Pure-ftpd	2.85	78.16
Insider Threat	Cp	0.20	0.89
	Lighttpd	0.58	12.13
Phishing Email	Sendmail	18.09	238.20

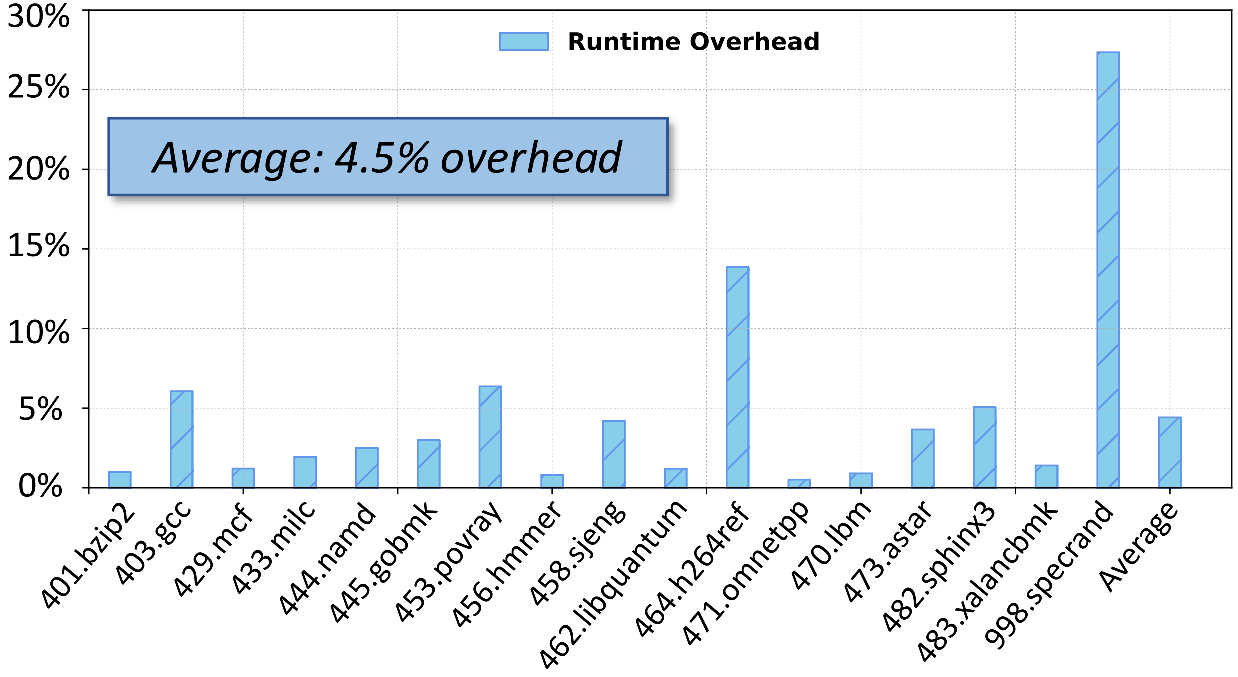
RTAG [Security'18]

- Record-and-replay
- DIFT with libdft

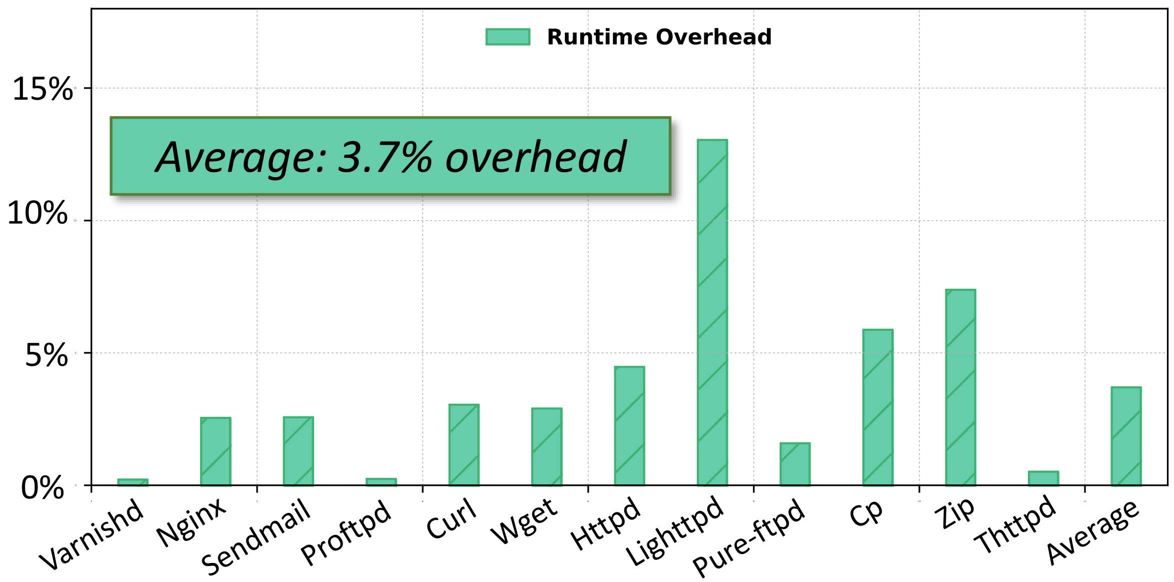
✓ PALANTIR reduces 77%-96% time from DIFT-based provenance tracking

Runtime Performance

Runtime Overhead on ***SPEC CPU 2006 benchmarks***



Runtime Overhead on ***real-world programs***



✓ PALANTIR's hardware PT incurs <5% runtime-overhead for processor tracing

Conclusion

- A major challenge in provenance analysis is dependency explosion.
- Insights:
  - Hardware-assisted approach provides efficient runtime performance
  - Static taint summarization can segregate offline overhead
- We propose PALANTIR:
  - Optimize attack provenance by hardware-enhanced system observability
  - Resolve dependency explosion by using instruction-level data flow

PALANTIR: Optimizing Attack Provenance with Hardware-enhanced System Observability

Thank You!

chuqiz@comp.nus.edu.sg

🔮

Artifact Available: *https://github.com/Icegrave0391/Palantir*

Backup Slides

Storage Cost

- ***Whole system*** storage cost
  - Overall PT trace storage: *98.4GB-111.6GB/day*
- ***Running server*** storage cost
  - Request 10,000 times within 32 concurrent connections

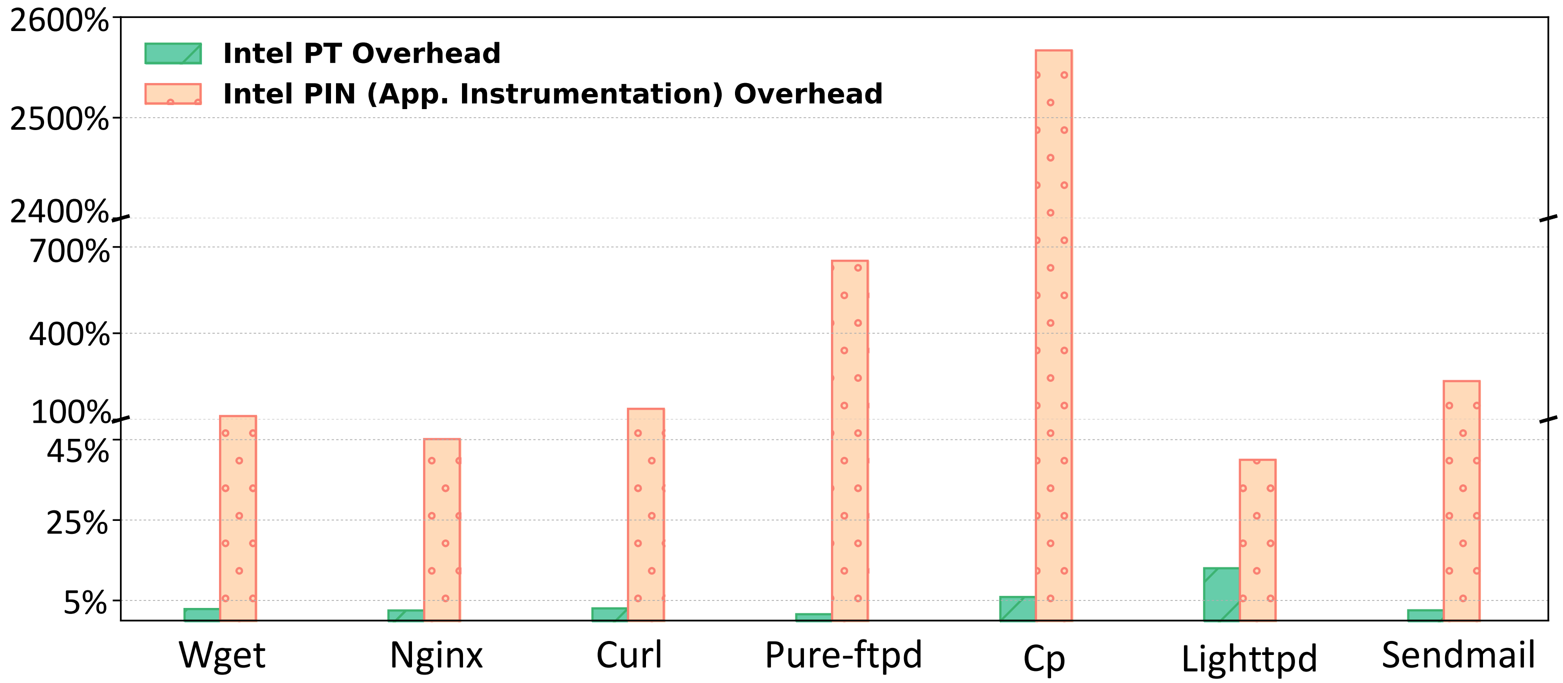
*PT Storage Cost (MB) to trace web servers (**10,000 requests**)*

Program	100KB File	512KB File	1MB File	10MB File
Nginx	74	74	75	83
Httpd	274	275	274	304
Lighttpd	42	51	55	107
Thttpd	46	50	50	57

✓ ***Acceptable,*** PALANTIR can free the storage after provenance optimizing

Runtime Performance - Comparison

- ***Hardware Processor Tracing*** vs. ***Dynamic Instrumentation***



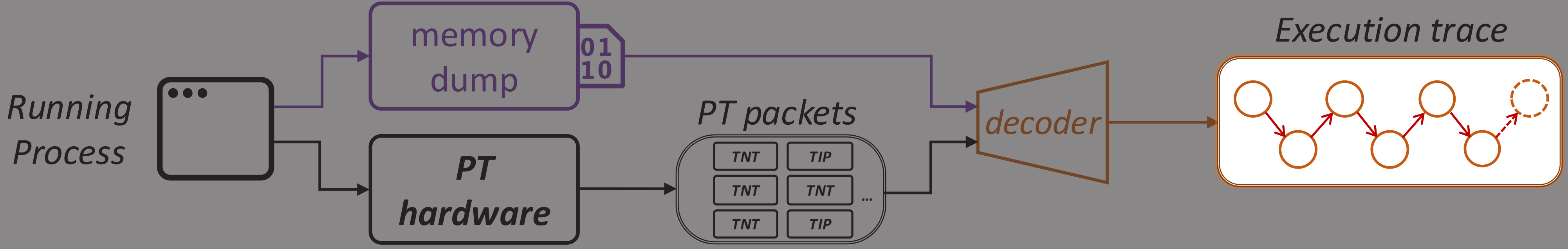
✓ Hardware PT is 3x-436x faster than instrumentation-based runtime tracing

Hardware is the new software: *Processor Tracing with runtime efficiency*

- ***Intel® Processor Tracing (Intel® PT):***

  Record program ***control flow transfer*** within ***trivial (< 7%) runtime overhead***

  ✓ Recover program ***execution history*** with runtime ***memory layout***



Execution Trace

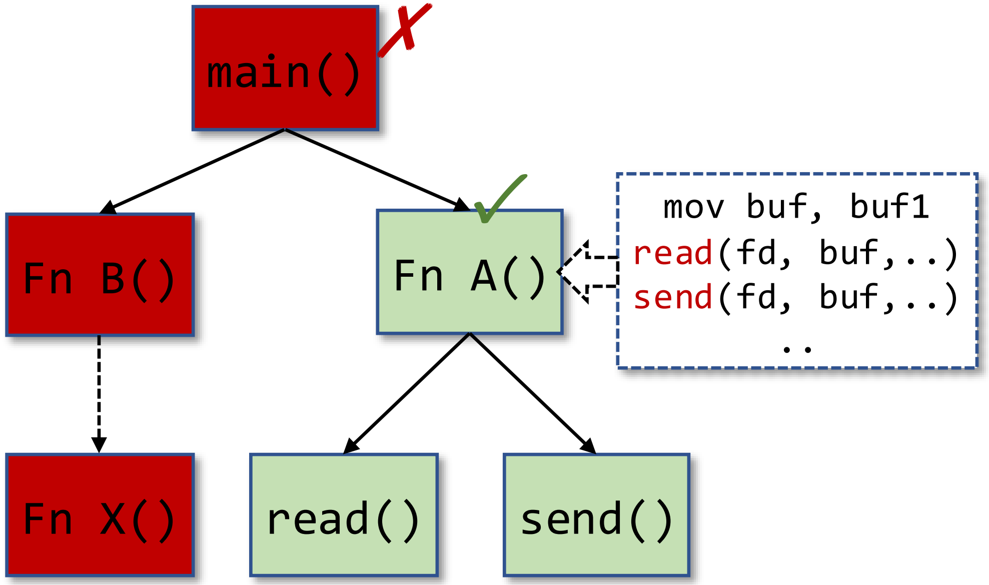
- A Sequence of ***basic blocks*** indicates program ***control flow*** and ***execution history***

✓ Enable ***offline*** data flow analysis to ***recover instruction-level data flow***

Scope Refinement:
Cut down offline computation overhead

- Whole program binary/execution trace: a massive codebase

  Observation: Only a small segment of code is related to the taints of system calls

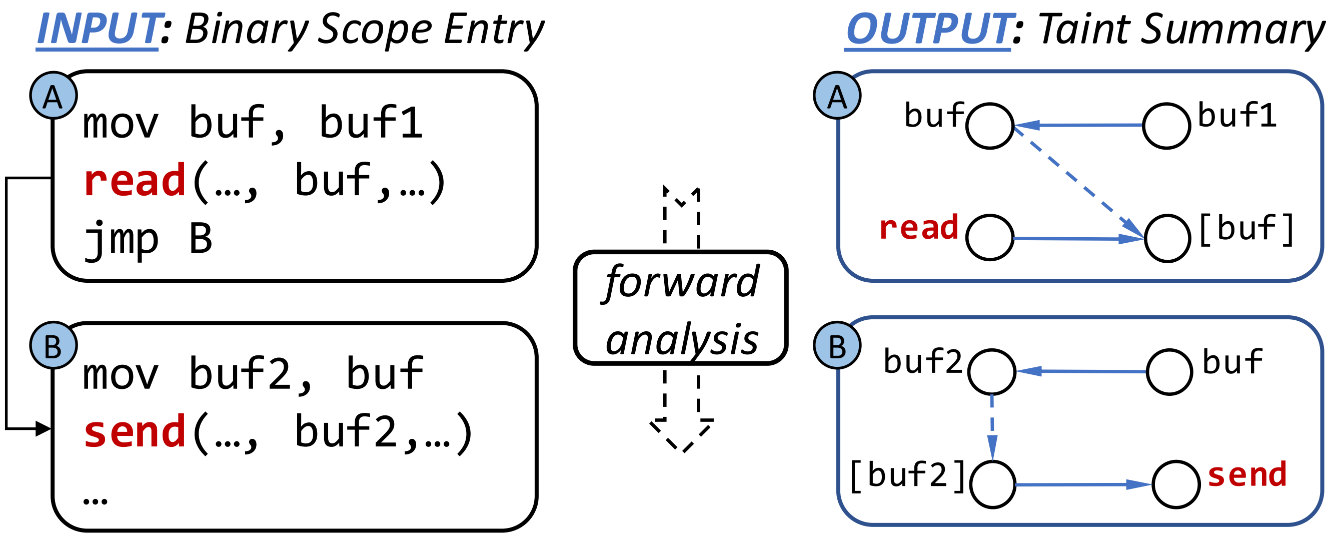


Call Graph

- ✓***Taint Scope:***
  - ***Introduce*** taint sources (e.g., read)
  - ***Propagate*** the taints (data flow)
  - ***Reach*** taint sinks (e.g., send)
- Refine the scope via the ***call graph traversal***
- The offline analysis only needs to be performed inside the scope

Static Taint Summary: segregate offline computation overhead

- Directly recover syscall taints on the ***execution trace*** is time-, memory-consuming (low efficiency) ☹
  - 💡 ***Pre-summarize*** taint propagation logic ***per basic block*** by performing the forward inter-procedural ***static binary analysis***


INPUT: Binary Scope Entry

```
A
mov buf, buf1
read(…, buf,…)
jmp B

B
mov buf2, buf
send(…, buf2,…)
…
```

Static analysis is context-, field-, and path-sensitive.

Assign ***symbolic value*** for unknown variables to mitigate memory alias.