ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting

Shiqing Ma, Xiangyu Zhang, Dongyan Xu

PURDUE
UNIVERSITY

Provenance Collection

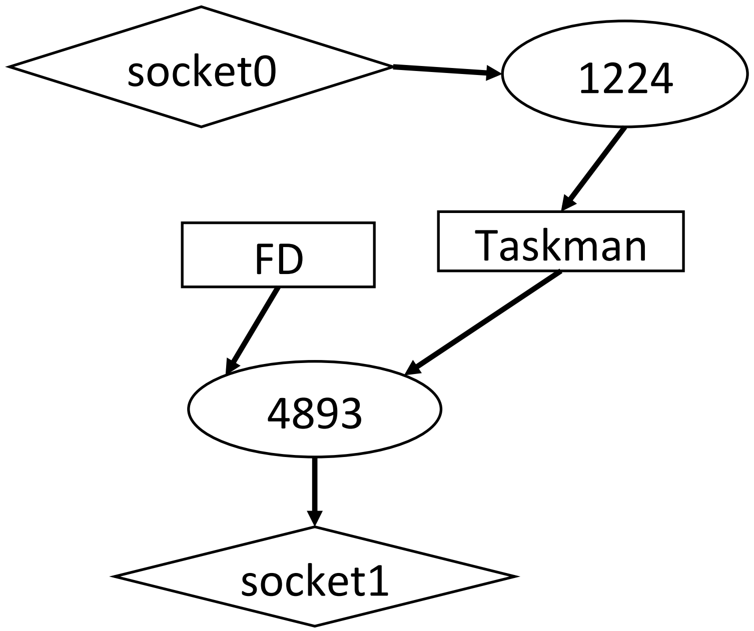
- Provenance, a.k.a. *lineage of data*
    - Data's life cycle
        - Origins
        - Accesses
        - Deletion
- Existing Approaches
    - Tainting
    - Audit Logging

Example:

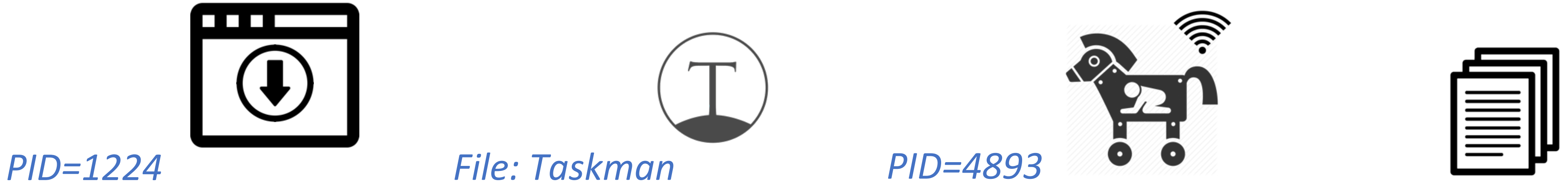
PID=1224 *File: Taskman* *PID=4893*

Logging

1. ……….
2. *PID=1224*, Receives from *socket0*
3. *PID=1224*, Writes to File *Taskman*
4. ……….
5. *PID=4893,* Starts from File *Taskman*
6. *PID=4893*, Reads file *FD*
7. *PID=4893*, Sends data to *socket1*
8. ……….



Example:



Data Leaked (taint *FD*)
== Taint set contains *{ FD }*
== T[Taskman], T[Data sent]

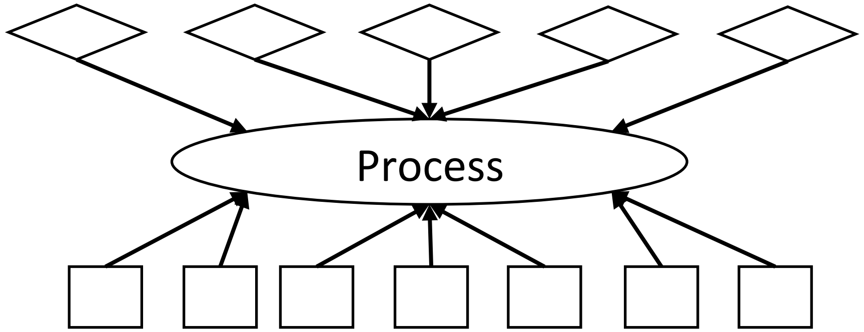
Affected by phishing website (tating *socket0*)
== Taint set contains *{ socket0 }*
== T[Browser], T[File:Taskman], T[Taskman], T[Data sent]

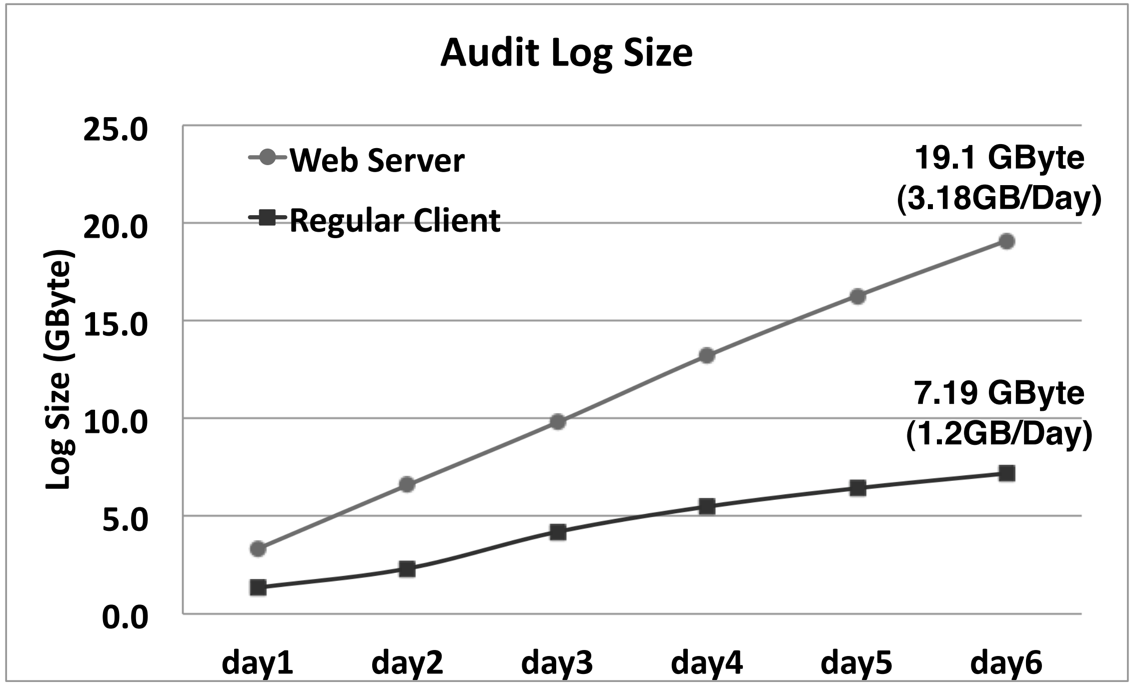
Tainting

1. ……….
2. T[Browser] = T[Browser] V *{ socket0 }* = *{ socket0 }*
3. T[File:Taskman] = T[Browser] = *{ socket0 }*
4. ……….
5. T[Taskman] = T[File:Taskman] = *{ socket0 }*
6. T[Taskman] = T[Taskman} V *{ FD }* = *{ socket0, FD }*
7. T[Data sent] = T[Taskman] = *{ socket0, FD }*
8. ……….

Limitations of *Audit Logging*

- Overhead [LogGC]
  - Linux Audit Framework: ***~40%*** run time slow down
    - Some low overhead system: Hi-Fi etc.
  - Storage: ***~2G*** per day
- ***Dependency Explosion*** Problem



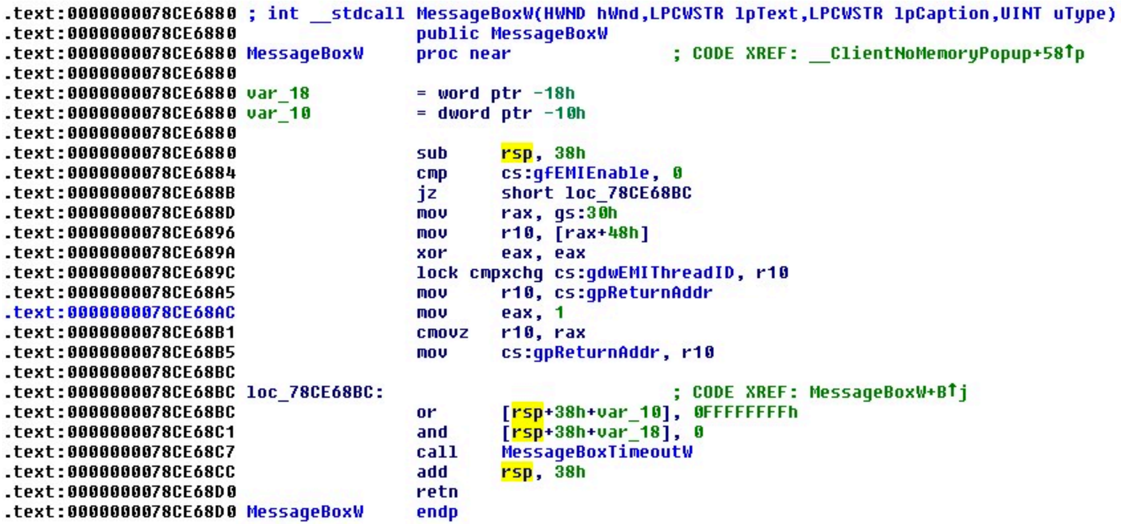


Limitations of *Tainting*

- Overhead
  - Most of existing approaches are ***instruction level*** tainting
  - Run time: multiple times slow down without hardware support [libbdf]

- Implicit flow
  - Information flow through control dependencies [DTA++]

- Implementation Complicity
  - Instrumentation for each instruction
  - Libraries and VMs
  - Different PLs and their run time



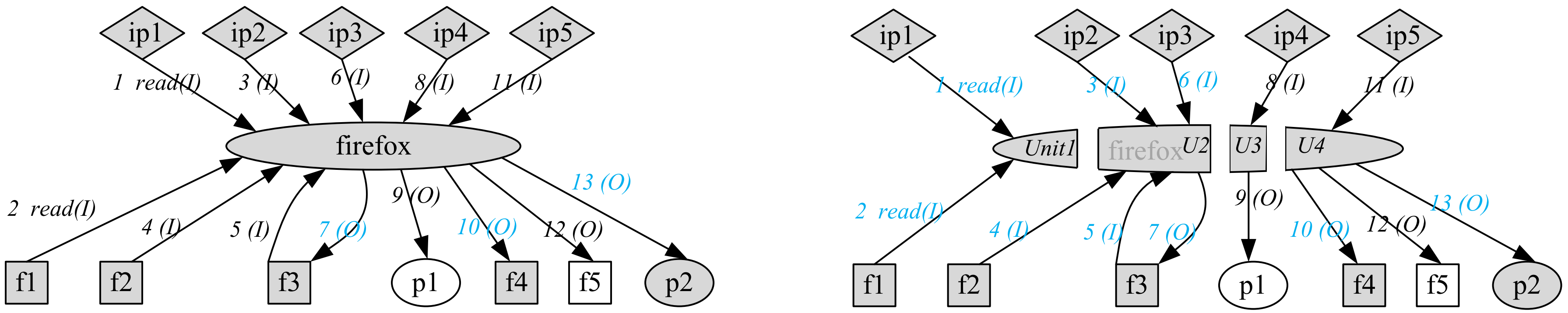
```
.text:0000000078CE6880 ; int __stdcall MessageBoxW(HWND hWnd,LPCWSTR lpText,LPCWSTR lpCaption,UINT uType)
.text:0000000078CE6880                 public MessageBoxW
.text:0000000078CE6880 MessageBoxW     proc near               ; CODE XREF: __ClientNoMemoryPopup+58↑p
.text:0000000078CE6880
.text:0000000078CE6880 var_18          = word ptr -18h
.text:0000000078CE6880 var_10          = dword ptr -10h
.text:0000000078CE6880
.text:0000000078CE6880                 sub     rsp, 38h
.text:0000000078CE6884                 cmp     cs:gfEMIEnable, 0
.text:0000000078CE688B                 jz      short loc_78CE68BC
.text:0000000078CE688D                 mov     rax, gs:30h
.text:0000000078CE6896                 mov     r10, [rax+48h]
.text:0000000078CE689A                 xor     eax, eax
.text:0000000078CE689C                 lock cmpxchg cs:gdwEMIThreadID, r10
.text:0000000078CE68A5                 mov     r10, cs:gpReturnAddr
.text:0000000078CE68AC                 mov     eax, 1
.text:0000000078CE68B1                 cmovz   r10, rax
.text:0000000078CE68B5                 mov     cs:gpReturnAddr, r10
.text:0000000078CE68BC
.text:0000000078CE68BC loc_78CE68BC:                           ; CODE XREF: MessageBoxW+B↑j
.text:0000000078CE68BC                 or      [rsp+38h+var_10], 0FFFFFFFFh
.text:0000000078CE68C1                 and     [rsp+38h+var_18], 0
.text:0000000078CE68C7                 call    MessageBoxTimeoutW
.text:0000000078CE68CC                 add     rsp, 38h
.text:0000000078CE68D0                 retn
.text:0000000078CE68D0 MessageBoxW     endp
```

Our Idea

- A combination of ***Auditing Logging*** and ***Tainting***

- Taints: ***objects*** (file, socket etc.) or ***subjects*** (process etc.)
  - ***NOT*** traditional ***instruction*** level tainting
  - ***Coarse grained, accurate*** taint tracing

Background: BEEP [NDSS'13]

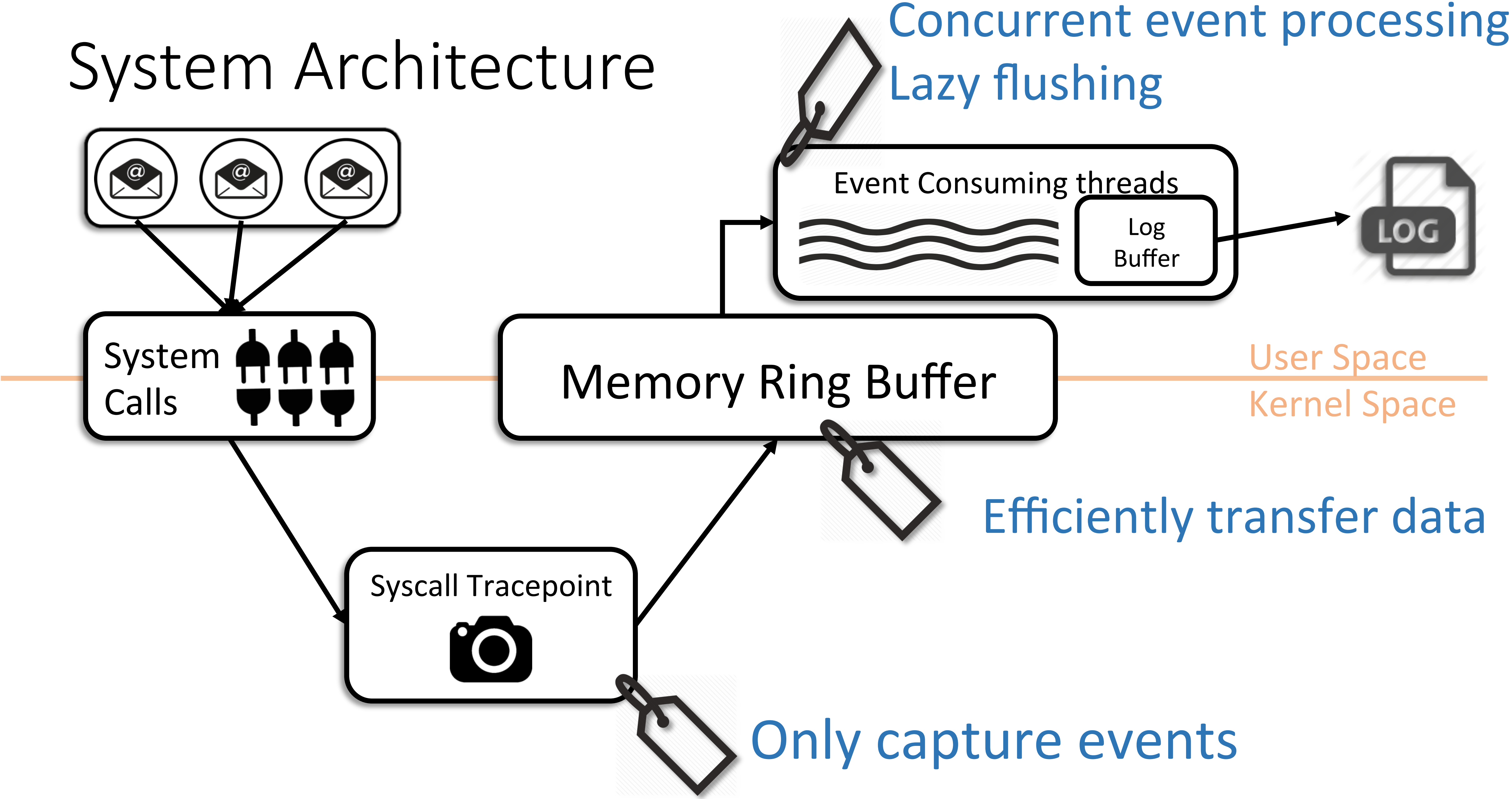


(a) before partition

(b) after partition

- Why using BEEP?
  - To solve the dependency explosion problem
  - ***Coarse grained, accurate*** taint tracing made possible

System Architecture



Concurrent event processing

Lazy flushing

Efficiently transfer data

Only capture events

Design: Kernel Space

- System call based approach
  - Linux system call table is relative stable

- System calls (can be easily extended) :
  - ***Process*** related operations: creation, and termination etc.
  - ***File*** descriptors operations: creation, and close etc.
  - For ***certain objects***: socket bind (*sys_bind*) etc.
  - ***Inter-process communication*** related system calls: pipe (*sys_pipe*) etc.
  - BEEP ***instrumented*** system calls: unit enter, unit end etc.

Design: User Space

- We consume events in user space by alternating between ***tainting*** and ***logging***.
- Principle:
  - When the effects of events are ***permanent***, we ***log***.
    - ***Permanent:*** writing to the disk.
  - When the effects of events are ***temporary***, we ***taint*** (to avoid unnecessary logging => less storage, less I/O, simpler graph).
    - ***Temporary***: IPC channel
- Propagation:
  - Follow the information flow

Example: Avoid *Redundant* Events

```
# vim opening a large file
    ...
    while((size = read(fd, buf)) > 0):
        add_node(root, buf)
    ...
    exit();
```

Logging

```
...
PID = 1483, TYPE = SYSCALL: Syscall = read
PID = 1483, TYPE = SYSCALL: Syscall = read
PID = 1483, TYPE = SYSCALL: Syscall = read
PID = 1483, TYPE = SYSCALL: Syscall = read
PID = 1483, TYPE = SYSCALL: Syscall = read
PID = 1483, TYPE = SYSCALL: Syscall = read
...
PID = 1483, TYPE = SYSCALL: Syscall = exit
```

ProTracer

```
...
T[ PID=1483 ] = { vim }
T[ PID=1483 ] = T[ PID=1483 ] V { fd } = { vim, fd }
T[ PID=1483 ] = T[ PID=1483 ] V { fd } = { vim, fd }
T[ PID=1483 ] = T[ PID=1483 ] V { fd } = { vim, fd }
T[ PID=1483 ] = T[ PID=1483 ] V { fd } = { vim, fd }
T[ PID=1483 ] = T[ PID=1483 ] V { fd } = { vim, fd }
T[ PID=1483 ] = T[ PID=1483 ] V { fd } = { vim, fd }
...
LogBuffer: T[ PID=1483 ] = { vim, fd }
```

Example: Lazy Flushing

```
# temporary files
   f = open(fname, create | write)
# File manipulation on the file
   while (not done)
       edit(f)
# delete temporary file
   delete(f)
```

Logging

...

TYPE = SYSCALL : **Syscall** = open, FD = 8

TYPE = SYSCALL: **Syscall** = write, FD = 8

……

TYPE = SYSCALL: **Syscall** = write, FD = 8

TYPE = SYSCALL: **Syscall** = unlink , FD = 8

...

ProTracer

...

T[**FD=8**] = ***{ }***

T[**FD=8**] = ***{ vim }***

LogBuffer: T[**FD=8**] = ***{ vim }***

T[**FD=8**] = T[FD=8] V ***{ vim }*** = ***{ vim }***

LogBuffer: T[**FD=8**] = ***{ vim }***

DEL: T[**FD=8**]

...

LogBuffer

~~T[FD=8] = { vim }~~

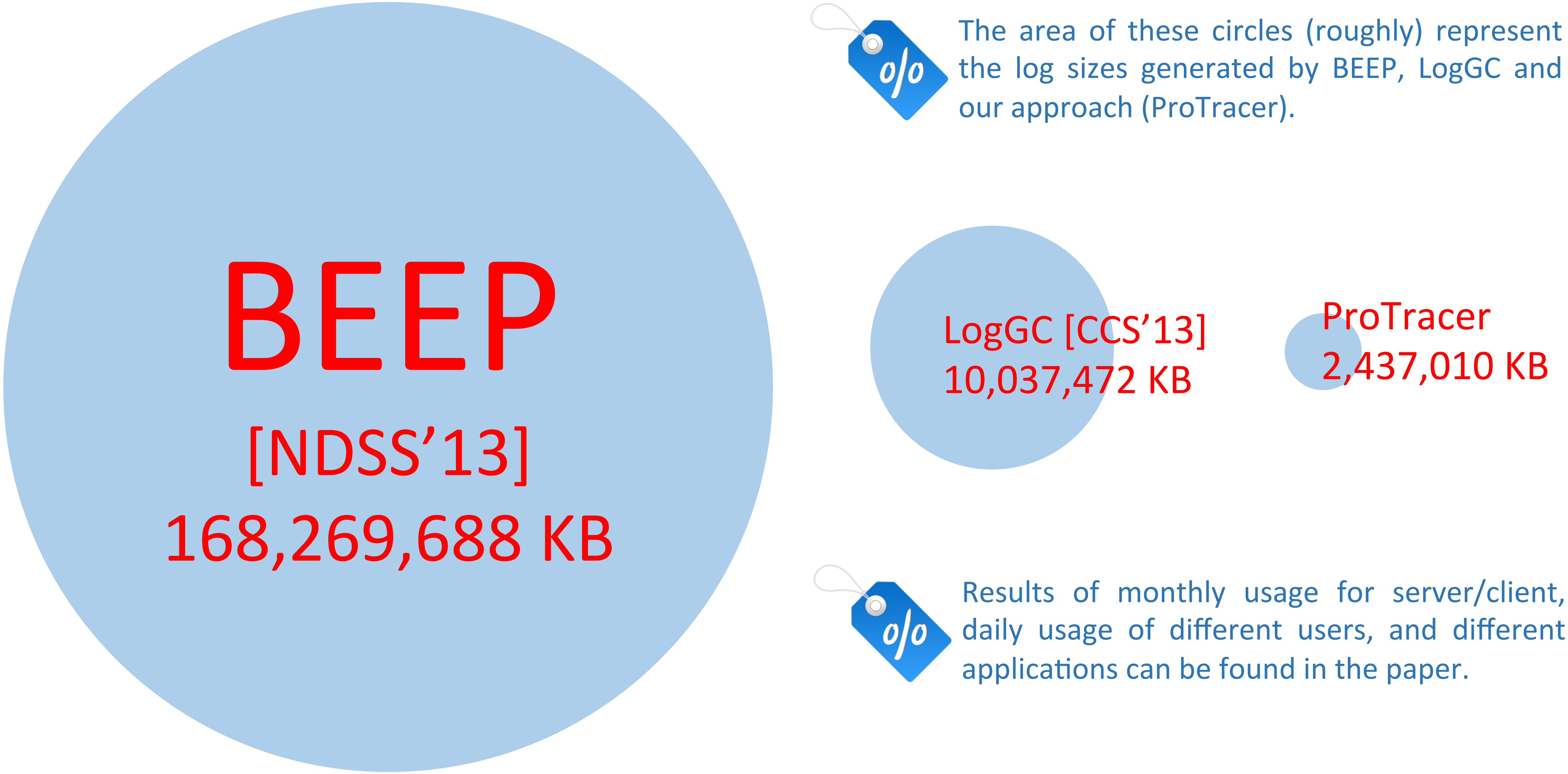
~~T[FD=8] = { vim }~~

Evaluation

- Storage Efficiency
- Run-time Efficiency
- Attack Investigation Cases

Evaluation: Storage Efficiency (3 months, client)

The area of these circles (roughly) represent the log sizes generated by BEEP, LogGC and our approach (ProTracer).



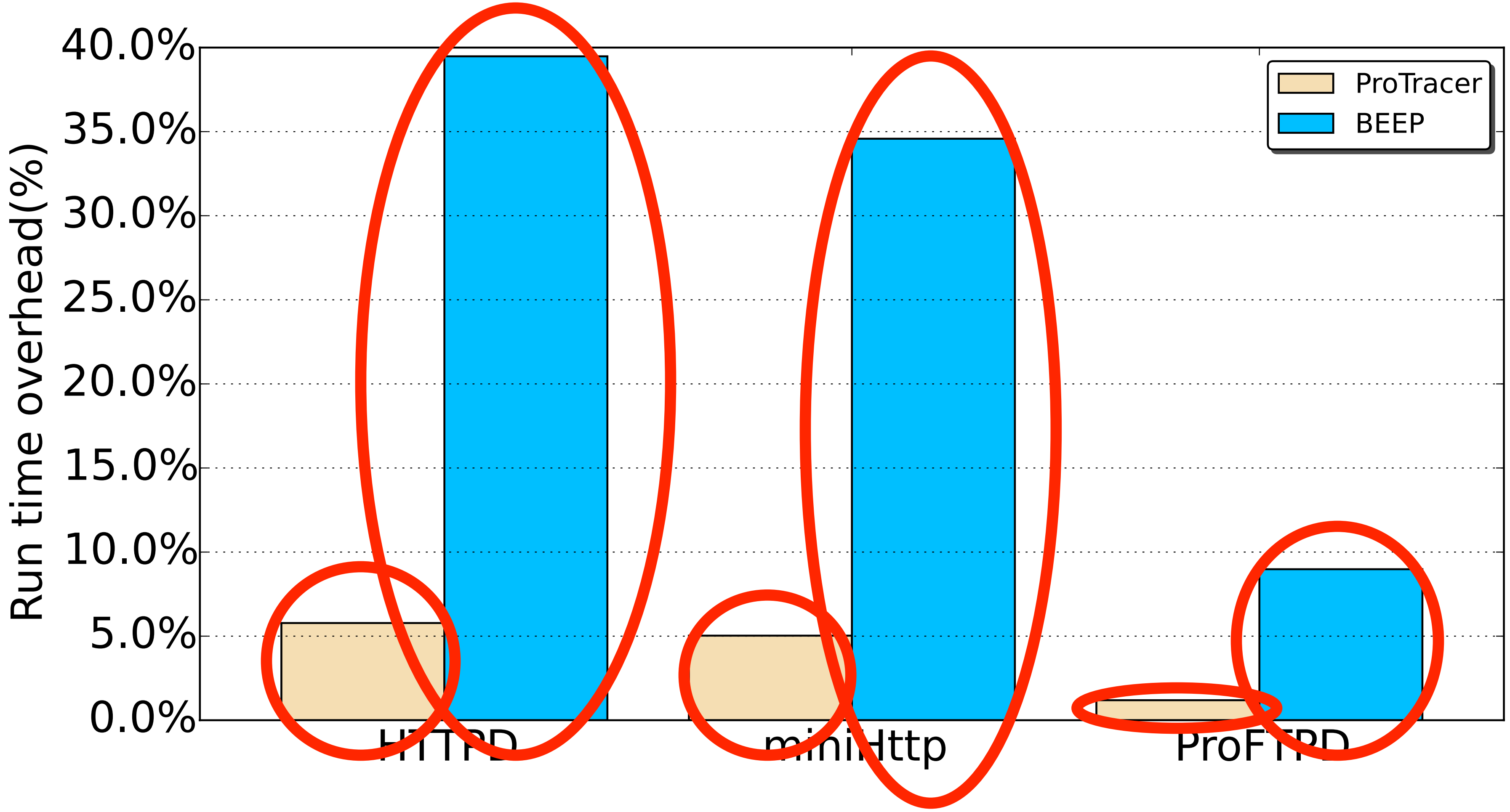
BEEP [NDSS'13] 168,269,688 KB

LogGC [CCS'13] 10,037,472 KB

ProTracer 2,437,010 KB

Results of monthly usage for server/client, daily usage of different users, and different applications can be found in the paper.

Evaluation: Run time Efficiency (Individual Servers)

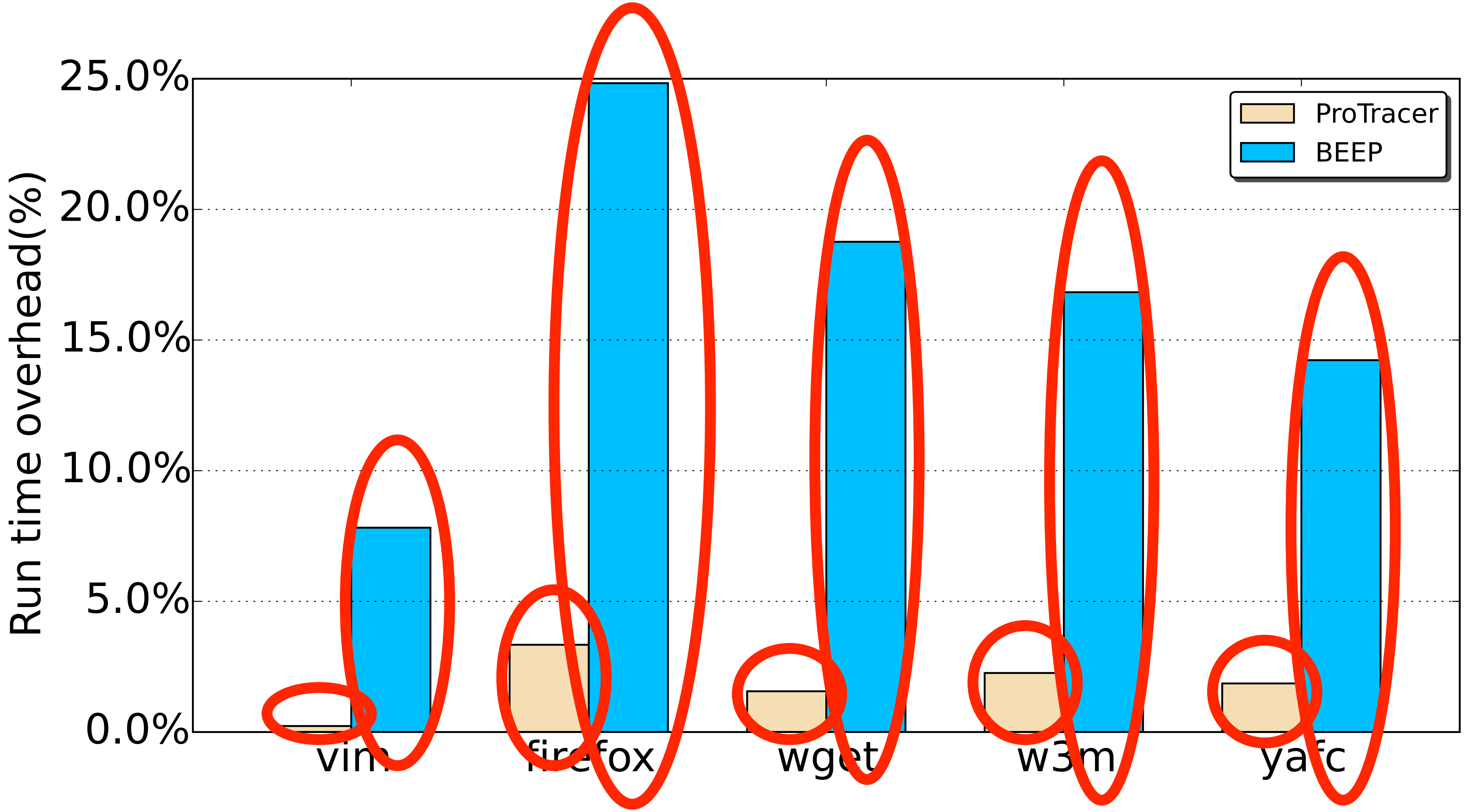


4.0%

v.s.

27.7%

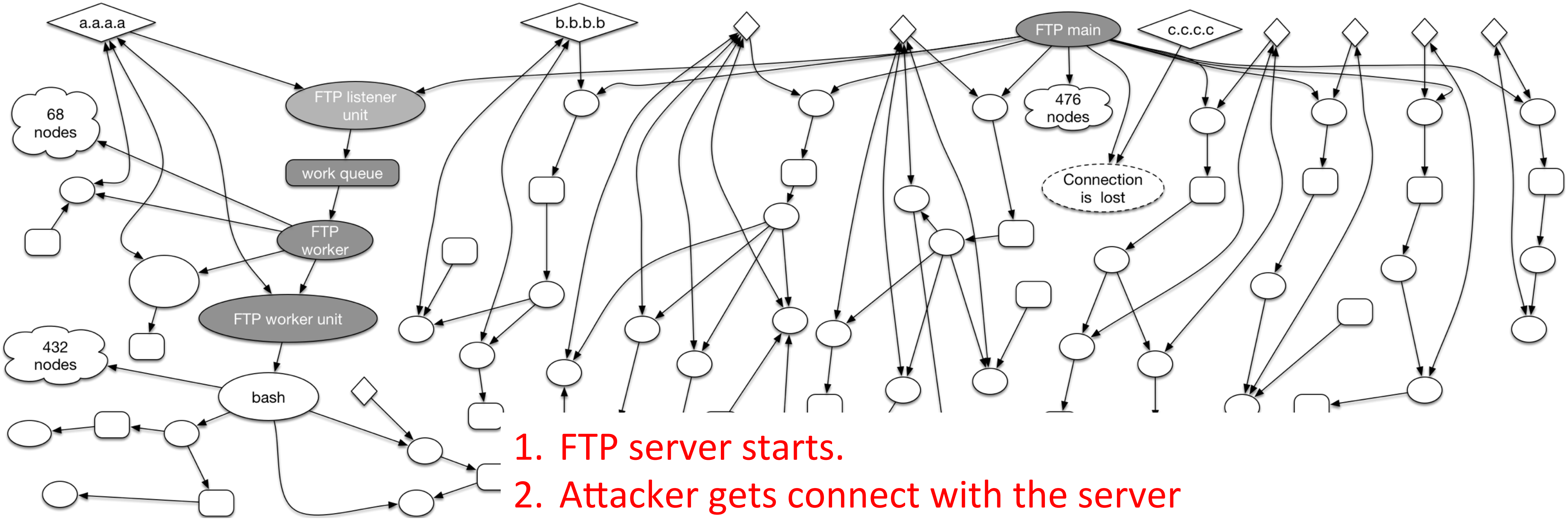
Evaluation: Run time Efficiency (Client Programs)



1.9%
v.s.
16.5%

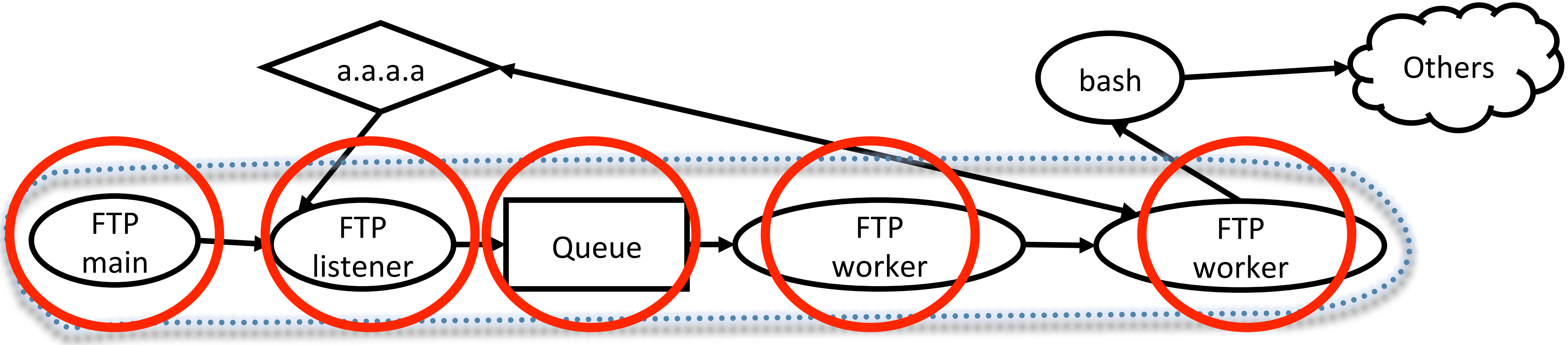
Whole system: 7% v.s. 40%

Evaluation: Attack Investigation Case - BEEP

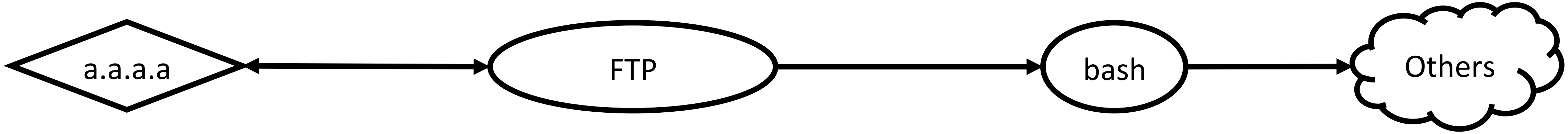


1. FTP server starts.
2. Attacker gets connect with the server
3. Attacker issues backdoor command to open the backdoor
4. Attacker gets a bash

Evaluation: Attack Investigation Case - ProTracer



More Cases in our paper.



Related Work

- Low Overhead System Logging
  - Butler [Security ’15, ACSAC ’12], Lee [ACSAC ‘15, NDSS ’13], Xu [ICDCS ’06], Lara [SOSP ’05], King [NDSS ’05, SOSP ’03]
- Tainting
  - Keromytis [NSDI ’12, VEE ’12], Smogor [USENIX ’09], Song [NDSS ’07], Mazieres [OSDI ’06], Kaashoek [SOSP ’05]
- Log storage and representation
  - Lee [ACSAC ’15, CCS ’13], Butler [ACSAC ’12], Zhou [SOSP ’11]
- Log integrity:
  - Moyer [Security ’15], Sion [ICDCS ’08]

Conclusion

- We developed ProTracer:
  - A provenance tracing system
- Key Components
  - A combination of ***logging*** and ***tainting***
  - A lightweight kernel module
  - Concurrent user space event processing
- Our evaluation
  - ***0.84G server*** side log data for ***3 months***
  - ***2.32G client*** side log data for ***3 months***
  - ***~7%*** run time overhead on average

Thank You

Q&A