Security-**O**riented **A**nalysis of **A**pplication **P**rograms (SOAAP)

Robert Watson, Khilan Gudka, Steven Hand,
Ben Laurie (Google), Anil Madhavapeddy

Workshop on Adaptive Host and Network Security
Lyon, France
Friday 14th September 2012

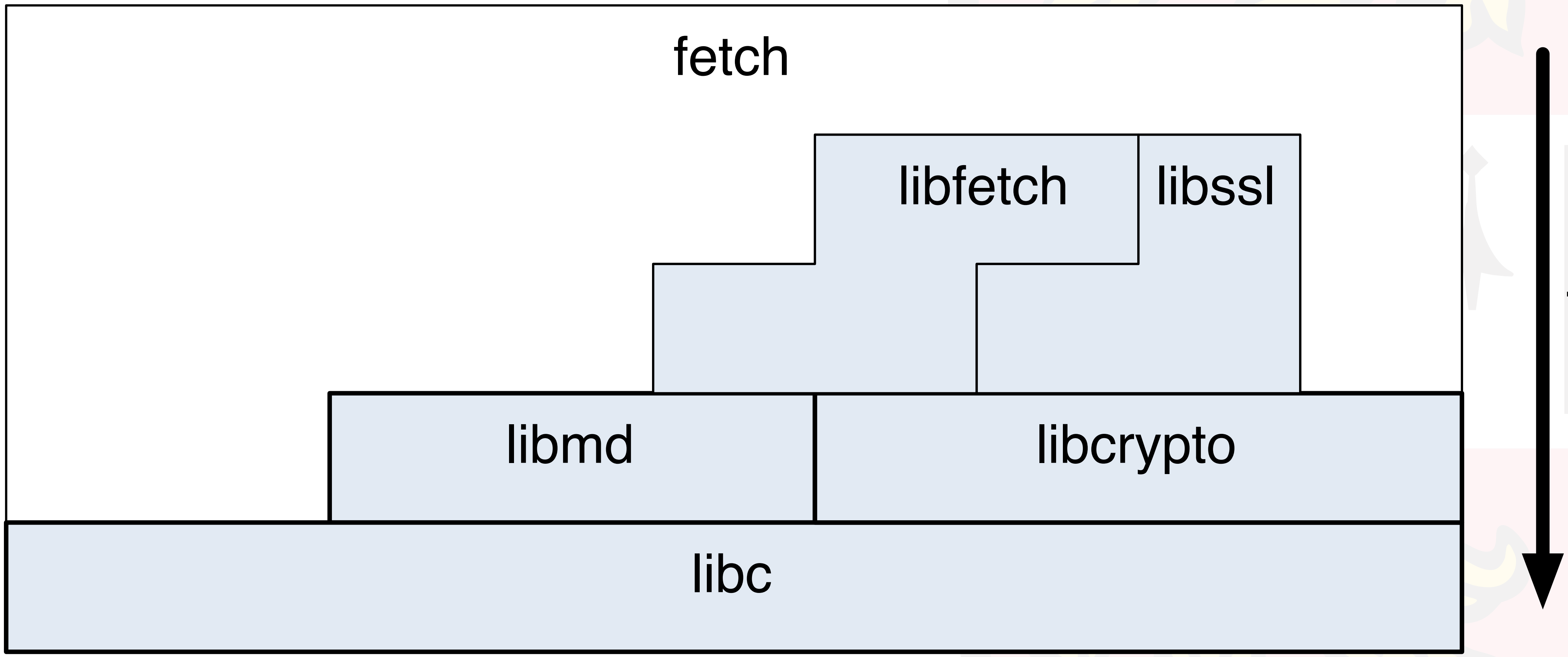
Approved for public release. This research is sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this article/presentation are those of the author/presenter and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

UNIVERSITY OF CAMBRIDGE

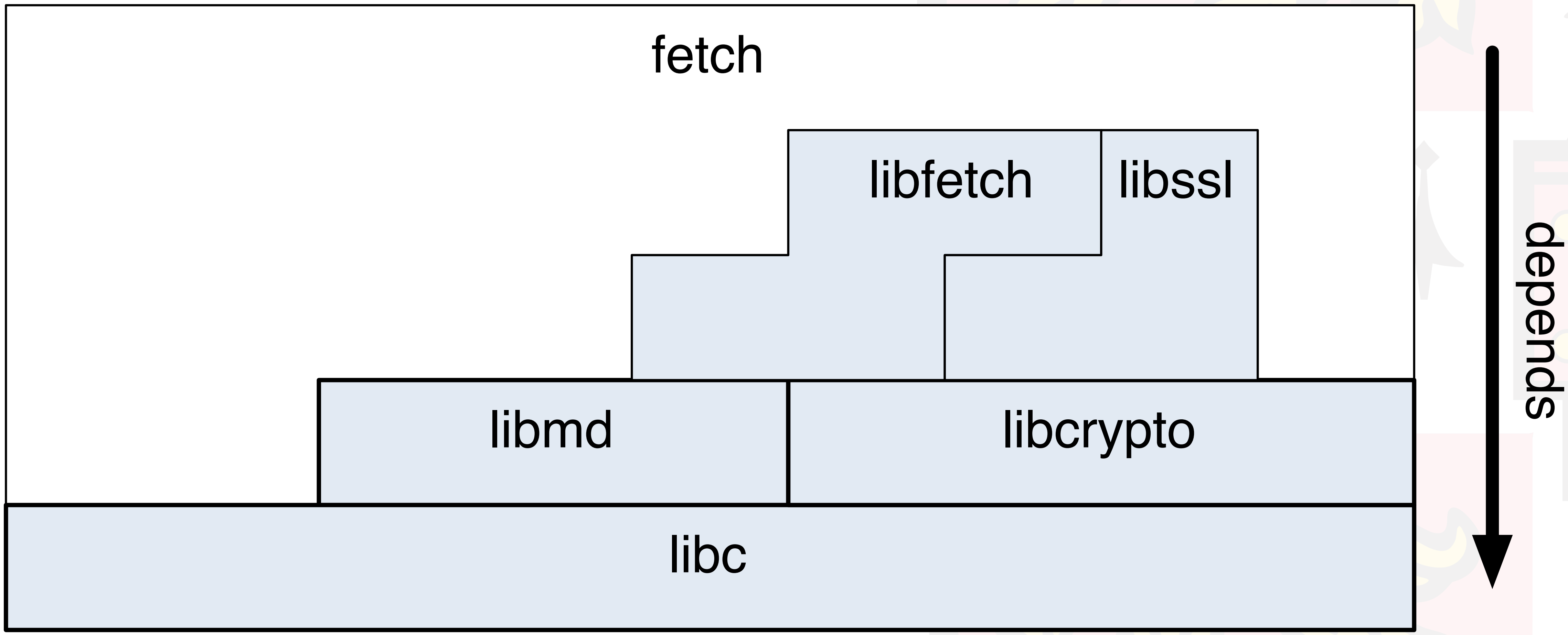
The Enemy ®

- Today's adversary is very sophisticated and able to execute arbitrary code through many means
- It's not just about buffer overflows any more
- Software stack comprises components from numerous **untrusted** origins. How do we know they do not contain **trojans** or **backdoors**? [Android hoover problem]

The Battlefield ®



The Battlefield ®



We need mitigation techniques that can work for both **known** and **unknown** vulnerabilities

Principle of least privilege

[Saltzer and Schroeder, 1975]

- Traditionally, UNIX processes run with the ambient rights of the user executing them
- An exploited vulnerability leaks all ambient rights
- But capturing policy for required rights is very difficult

Principle of least privilege

[Saltzer and Schroeder, 1975]

- Traditionally, UNIX processes run with the ambient rights of the user executing them
- An exploited vulnerability leaks all ambient rights

Solution: minimise the amount of code that runs with elevated privileges (TCB) and execute the rest in least privileged sandboxes with limited rights: E.g. "only read file X and write file Y"

- But capturing policy for required rights is very difficult

Principle of least privilege

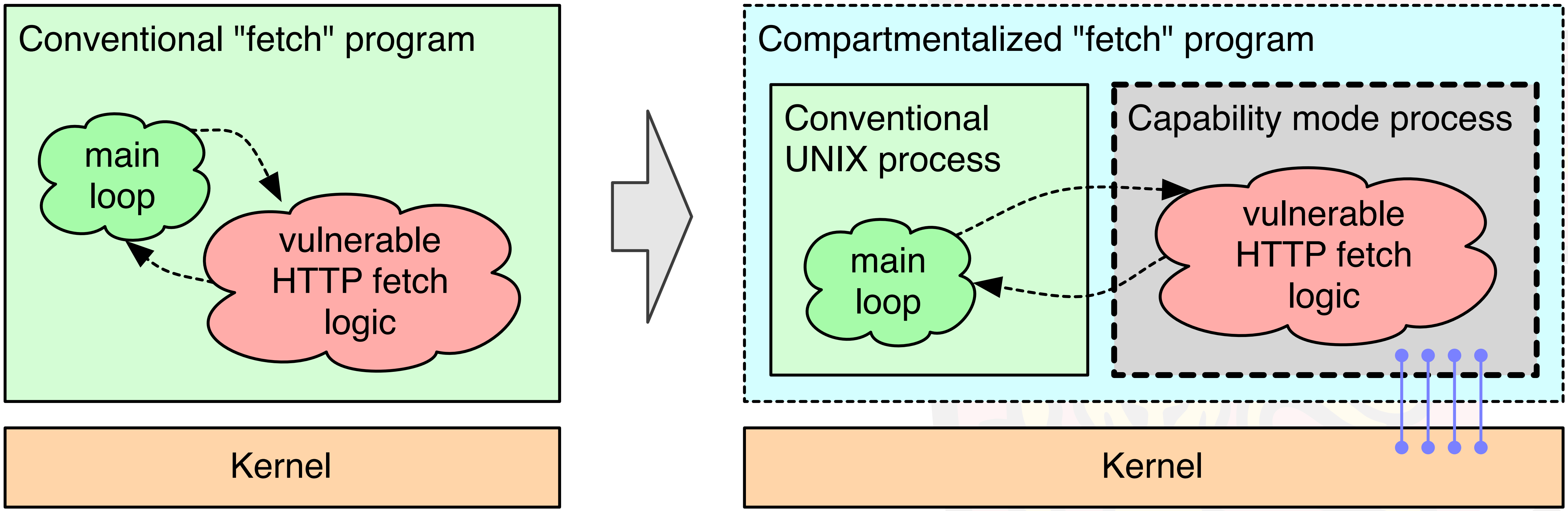
[Saltzer and Schroeder, 1975]

- Traditionally, UNIX processes run with the ambient rights of the user executing them
- An exploited vulnerability leaks all ambient rights

Solution: minimise the amount of code that runs with elevated privileges (TCB) and execute the rest in least privileged sandboxes with limited rights: E.g. "only read file X and write file Y"

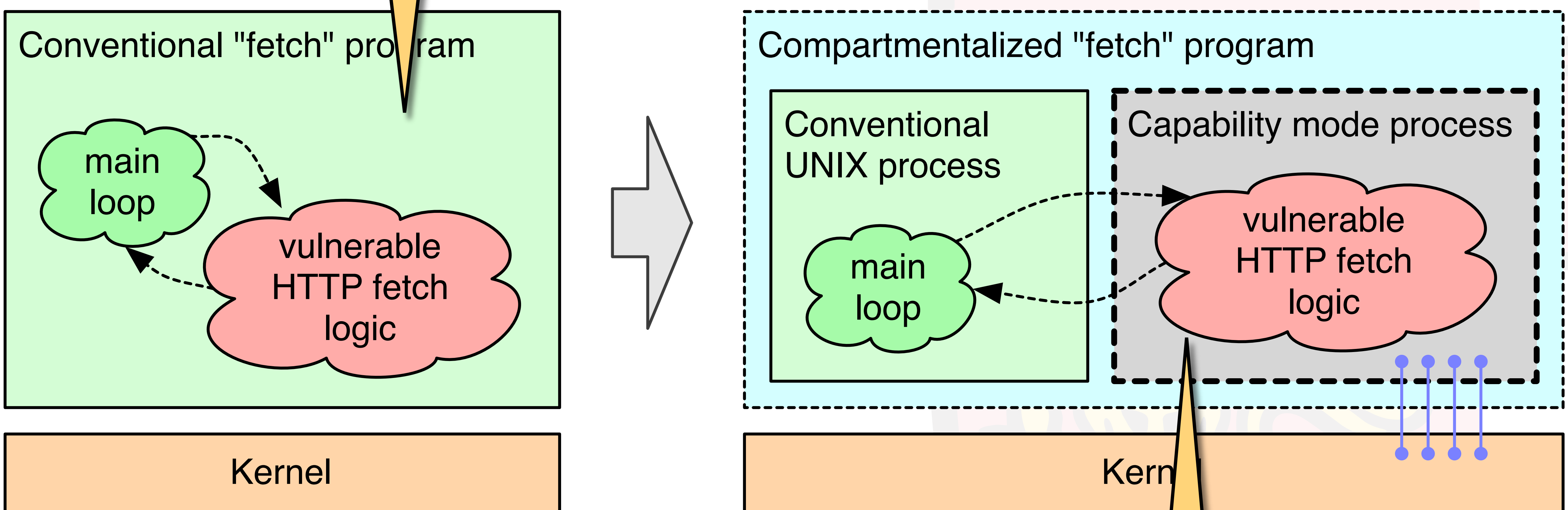
- But capturing policy for required rights is very difficult

Software compartmentalisation



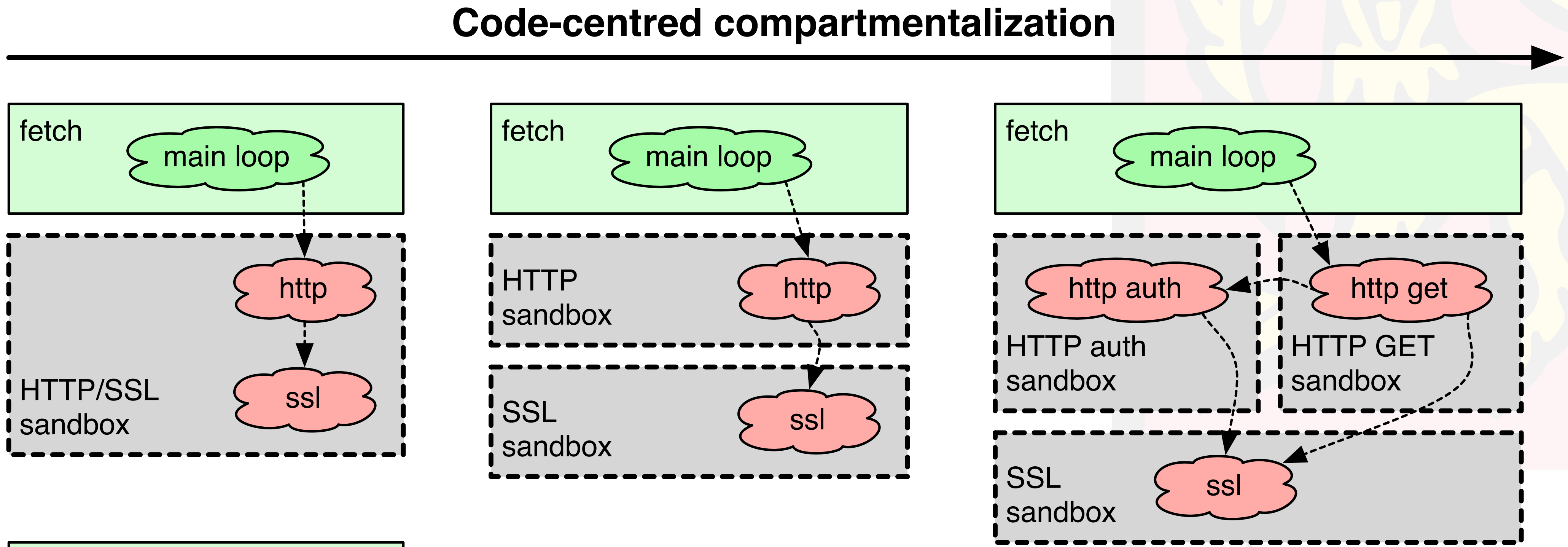
- Software compartmentalisation decomposes applications into many isolated components
- Each runs with only the rights required to perform its function

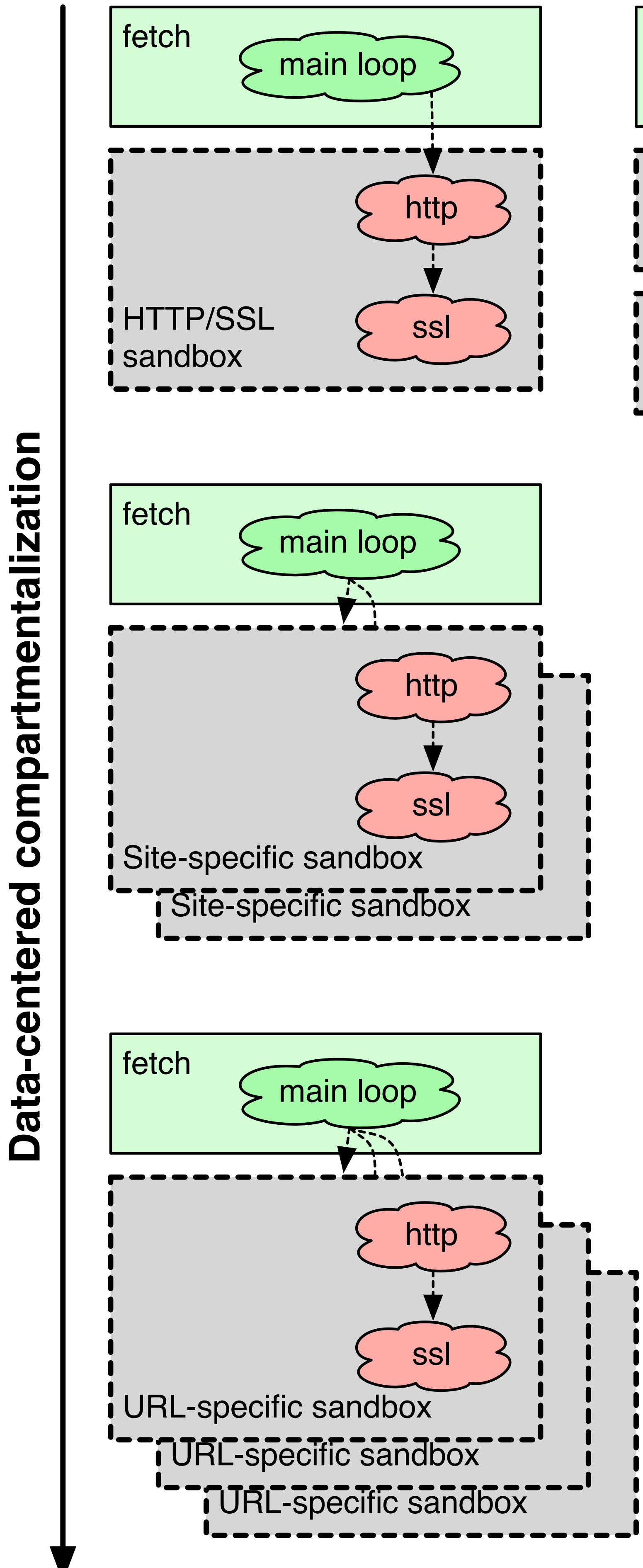
When a conventional application is compromised, its ambient rights are leaked to the attacker, e.g., full network and file system access.



When a compartmentalised application is compromised, only rights held by the exploited component leak to the attacker.

Most vulnerabilities will no longer yield significant rights, and attackers must exploit many vulnerabilities to meet their goals.





- Applications can be compartmentalized in many different ways, trading off **security**, **performance** and **complexity**.
- Finer-grained decompositions **mitigate** vulnerabilities better, as attacks yield fewer rights.
- The combination of code-centered and data-centered compartmentalisations aligns with the **object-capability** model

Lessons from Capsicum

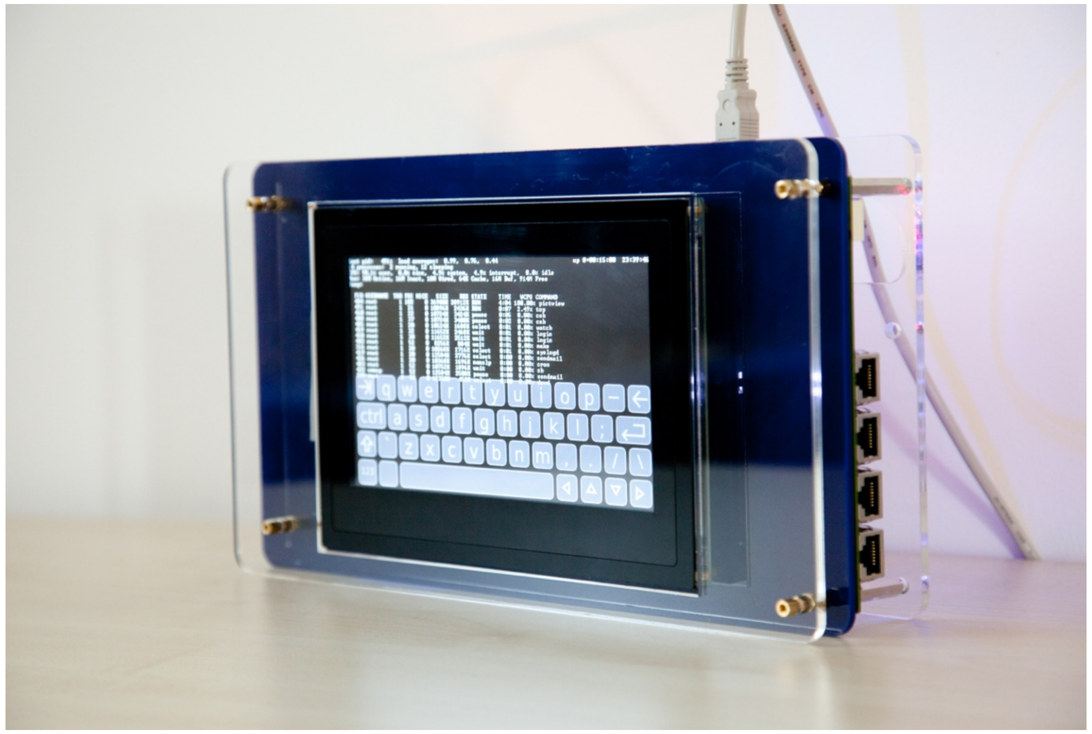
- Multi-year Cambridge/Google research project into the structure of operating system security (Watson, Anderson, Laurie, Kennaway)
- **Capsicum**: new operating system primitives for application compartmentalisation, reference application suite including Chromium

Lessons from Capsicum

- Multi-year Cambridge/Google research project into the structure of operating system security (Watson, Anderson, Laurie, Kennaway)
- **Capsicum**: new operating system primitives for application compartmentalisation, reference application suite including Chromium

Lesson: software designs that employ the principle of least privilege are neither **easily** nor **efficiently** represented in current hardware

Capability Hardware Enhanced RISC Instructions (CHERI)



- Joint SRI/Cambridge project
- **Modify hardware platform to enforce program protection**
- Capability registers, tagged memory
- Replace context switches with hardware message passing within an address space
- Apply RISC design philosophy: minimal, compiler-friendly hardware support to provide efficient protection

Compartmentalisation is hard!

- Compartmentalisation turns a "local" program into a distributed one
- Have to preserve functional correctness
  - e.g. data synchronisation/consistency
- Many different compartmentalisations present trade-offs: **performance**, **security** and **complexity**
- Have to find a mapping from intended goals to the underlying sandboxing technology

Gzip

- Compartmentalisation helps to mitigate vulnerabilities:

  "The gzip program contains a stack modification vulnerability that may allow an attacker to ***execute arbitrary code****, or create a denial-of-service condition..."*

  [Source: http://www.kb.cert.org/vuls/id/381508]

Gzip

- But getting it right is difficult, even for simple programs!

  "In adapting gzip, we were initially surprised to see a performance improvement; investigation of this unlikely result revealed that we had ***failed to propagate the compression level*** *(a global variable) into the sandbox, leading to the incorrect algorithm selection."*

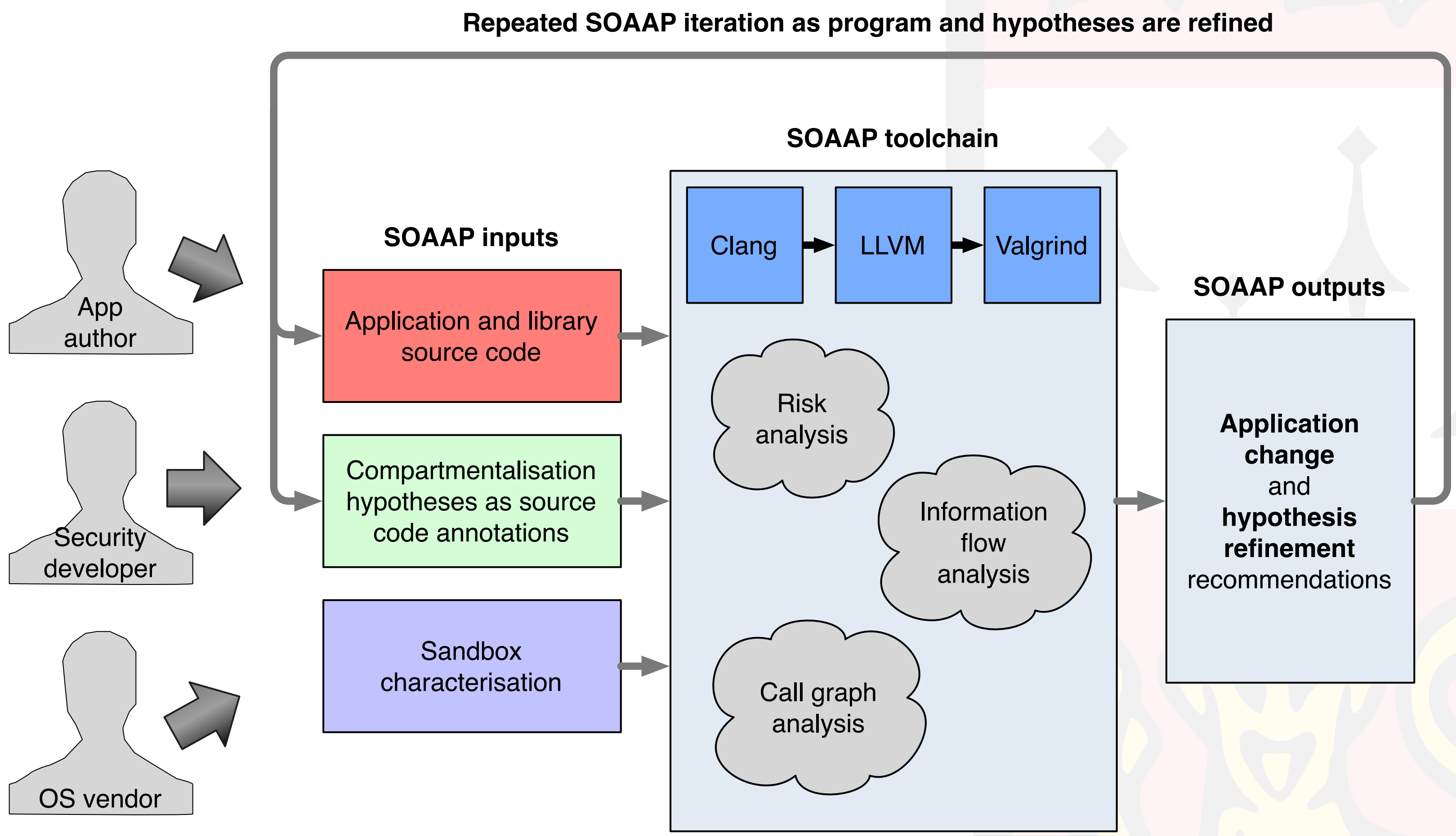
  [Watson et al. "Capsicum: practical capabilities for UNIX," USENIX Security 2010]

Sandboxing platforms: Chromium

	OS	Sandbox	LoC	FS	IPC	NET	S≠S′	Priv
DAC	Windows	DAC ACLs	22,350	⚠	⚠	✘	✘	✔
	Linux	chroot()	600	✔	✘	✘	✔	✘
MAC	Mac OS X	Sandbox	560	✔	⚠	✔	✔	✔
	Linux	SELinux	200	✔	⚠	✔	✘	✘
Cap	Linux	seccomp	11,300	⚠	✔	✔	✔	✔
	FreeBSD	Capsicum	100	✔	✔	✔	✔	✔

UNIVERSITY OF CAMBRIDGE

Security-oriented analysis of application programs (SOAAP)



- Motivated by the programmability problem in application compartmentalisation
- Allow application programmers to easily evaluate trade-offs through semi-automated analysis of possible compartmentalisations
- Annotation-driven static and dynamic program analysis and refinement of source code – and eventually program transformation

UNIVERSITY OF CAMBRIDGE

Security-oriented analysis of application programs (SOAAP)

- Given compartmentalisation goals, SOAAP allows the programmer to annotate:

UNIVERSITY OF CAMBRIDGE

Security-oriented analysis of application programs (SOAAP)

- Given compartmentalisation goals, SOAAP allows the programmer to annotate:
  - Functions that should run sandboxed

Security-oriented analysis of application programs (SOAAP)

- Given compartmentalisation goals, SOAAP allows the programmer to annotate:
  - Functions that should run sandboxed
  - Global state that can be accessed by sandboxes

Security-oriented analysis of application programs (SOAAP)

- Given compartmentalisation goals, SOAAP allows the programmer to annotate:
  - Functions that should run sandboxed
  - Global state that can be accessed by sandboxes
  - Descriptors that can be read/written by sandboxes

Security-oriented analysis of application programs (SOAAP)

- Given compartmentalisation goals, SOAAP allows the programmer to annotate:
  - Functions that should run sandboxed
  - Global state that can be accessed by sandboxes
  - Descriptors that can be read/written by sandboxes
  - System calls accessible to sandboxes

Security-oriented analysis of application programs (SOAAP)

- Given compartmentalisation goals, SOAAP allows the programmer to annotate:
    - Functions that should run sandboxed
    - Global state that can be accessed by sandboxes
    - Descriptors that can be read/written by sandboxes
    - System calls accessible to sandboxes
    - Privileges available via RPC interfaces

UNIVERSITY OF CAMBRIDGE

Security-oriented analysis of application programs (SOAAP)

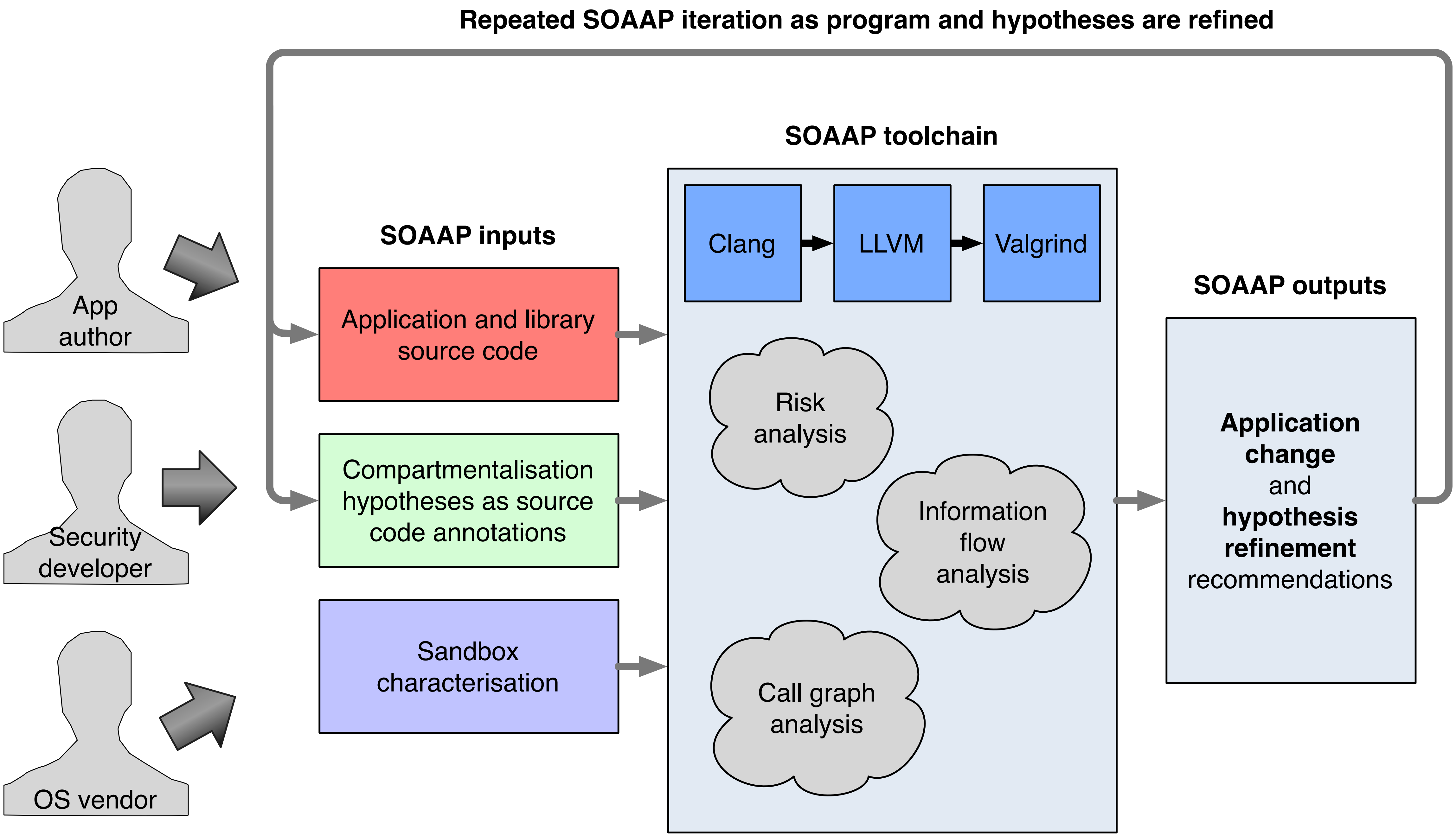
- Given compartmentalisation goals, SOAAP allows the programmer to annotate:
  - Functions that should run sandboxed
  - Global state that can be accessed by sandboxes
  - Descriptors that can be read/written by sandboxes
  - System calls accessible to sandboxes
  - Privileges available via RPC interfaces
  - Data that is confidential and should not be leaked

UNIVERSITY OF CAMBRIDGE

Security-oriented analysis of application programs (SOAAP)

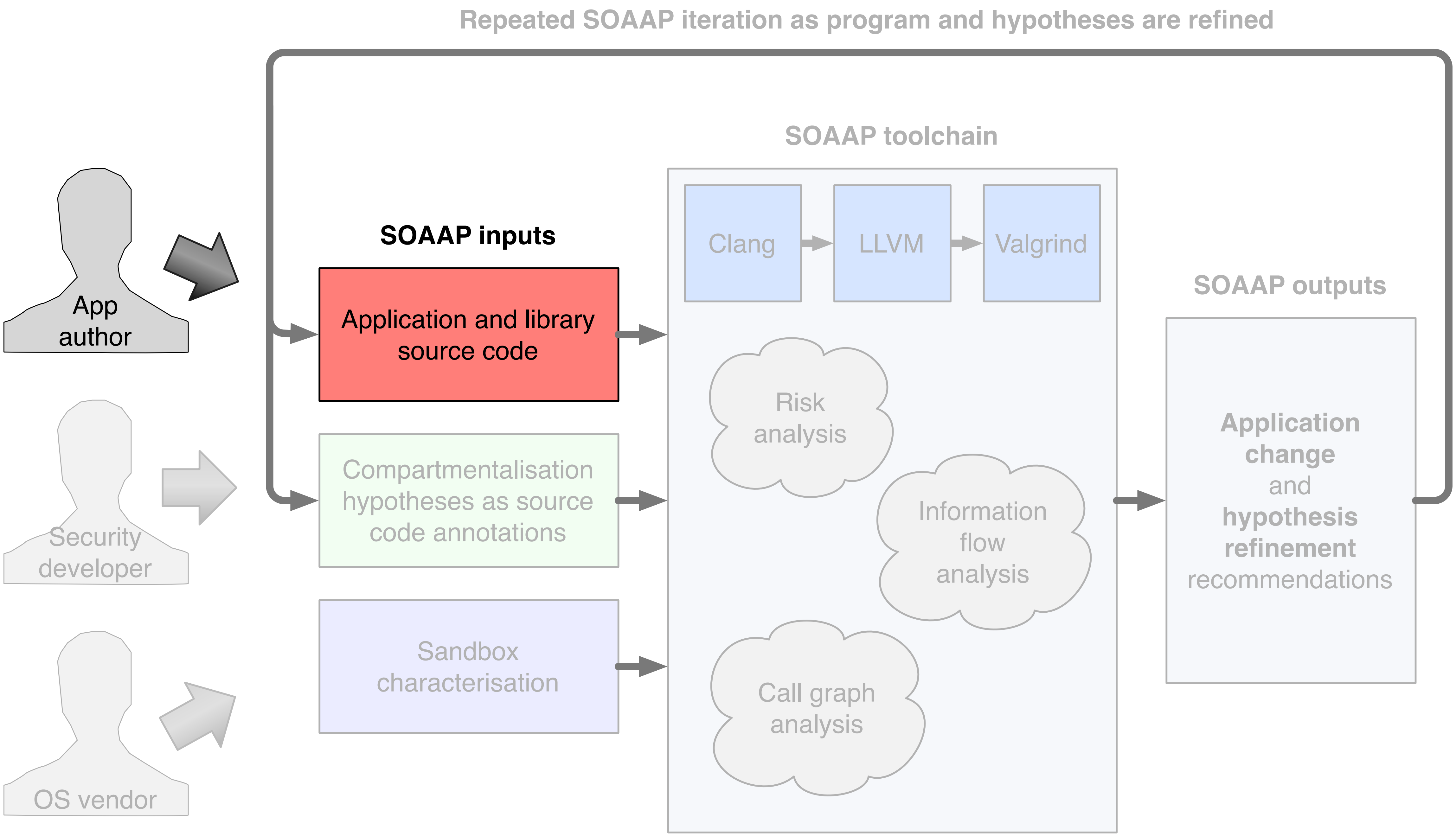
- Given compartmentalisation goals, SOAAP allows the programmer to annotate:
  - Functions that should run sandboxed
  - Global state that can be accessed by sandboxes
  - Descriptors that can be read/written by sandboxes
  - System calls accessible to sandboxes
  - Privileges available via RPC interfaces
  - Data that is confidential and should not be leaked
  - Code that is deemed risky

Security-oriented analysis of application programs (SOAAP)



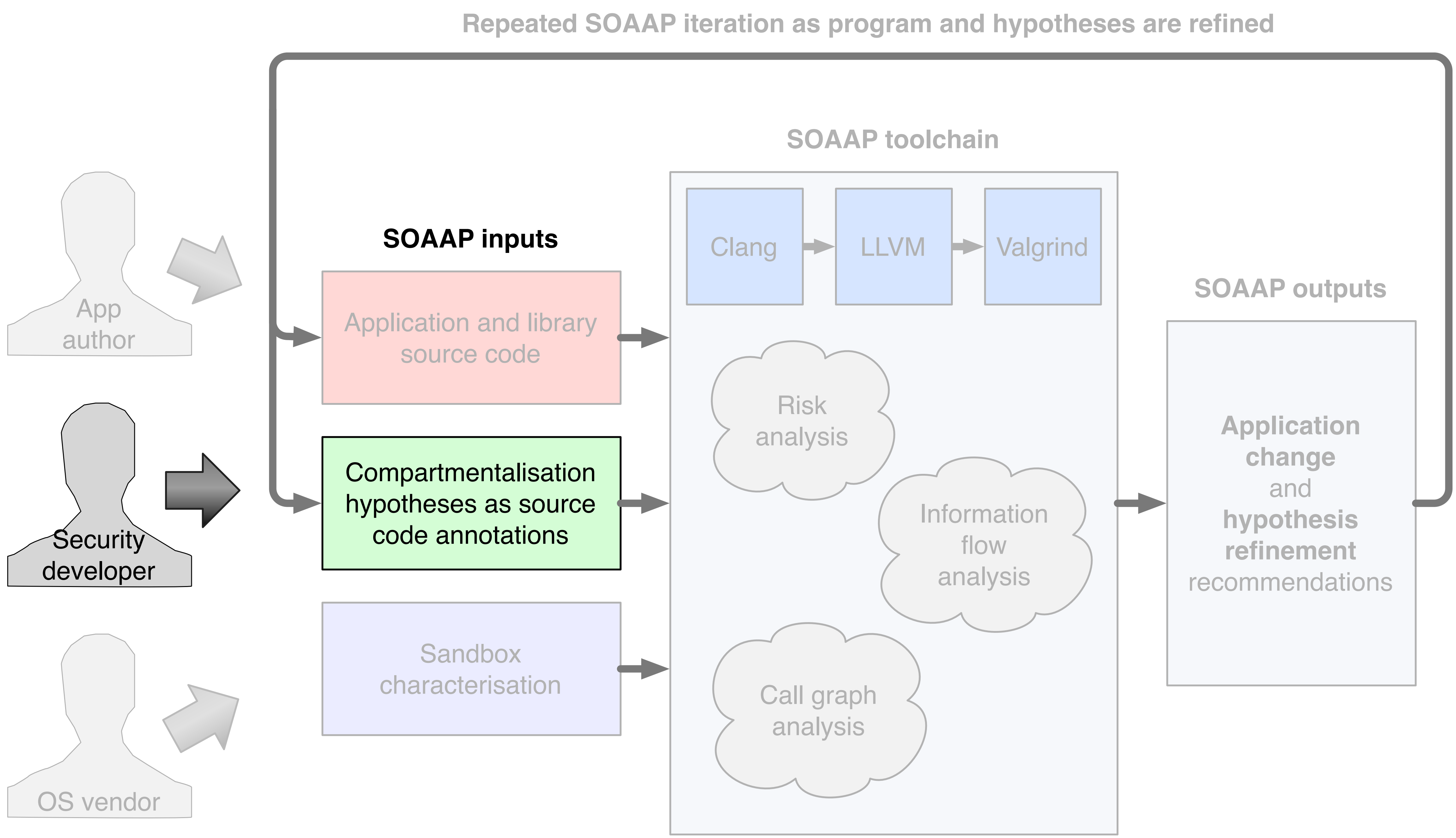
UNIVERSITY OF CAMBRIDGE

Security-oriented analysis of application programs (SOAAP)

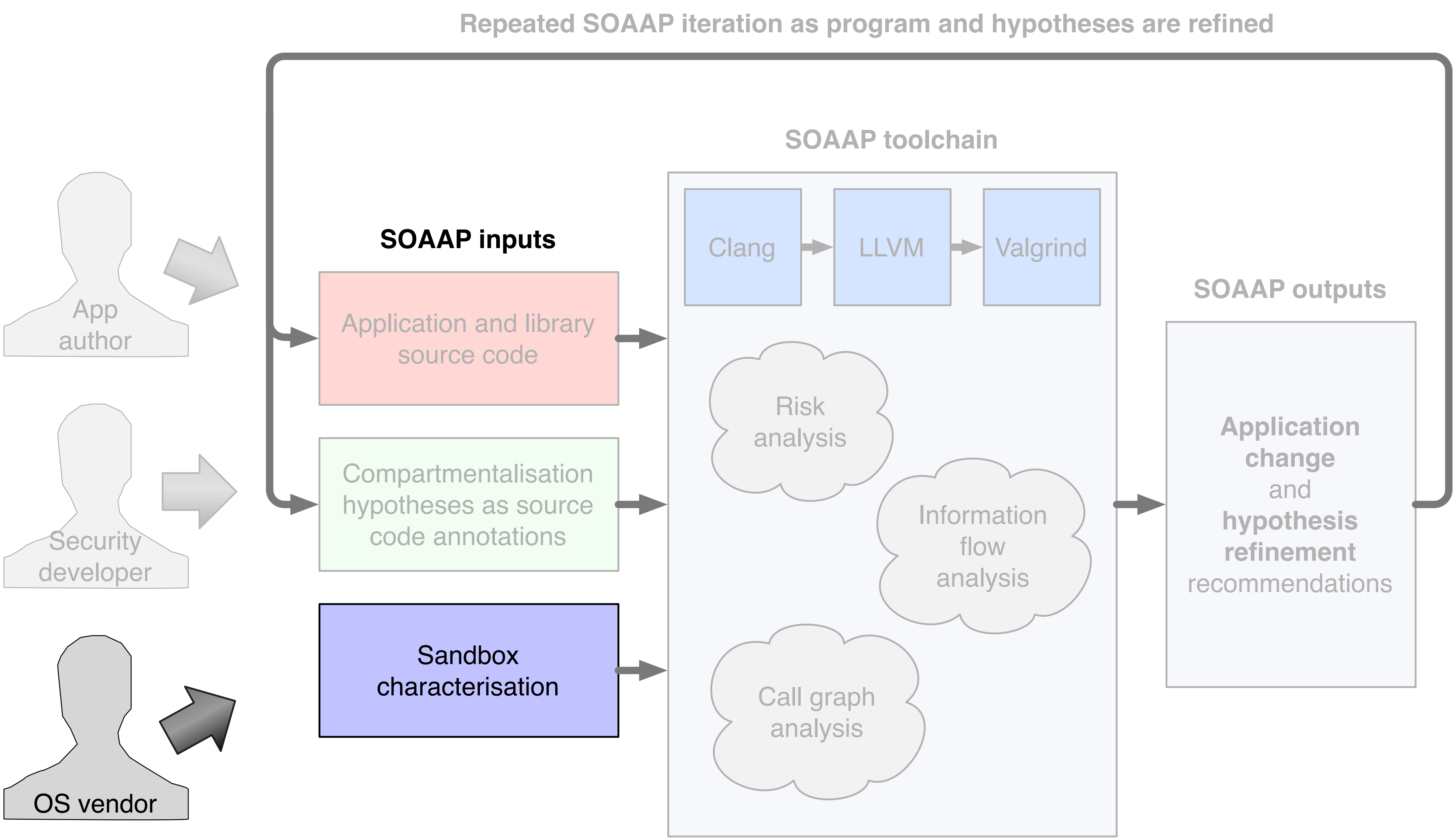


UNIVERSITY OF CAMBRIDGE

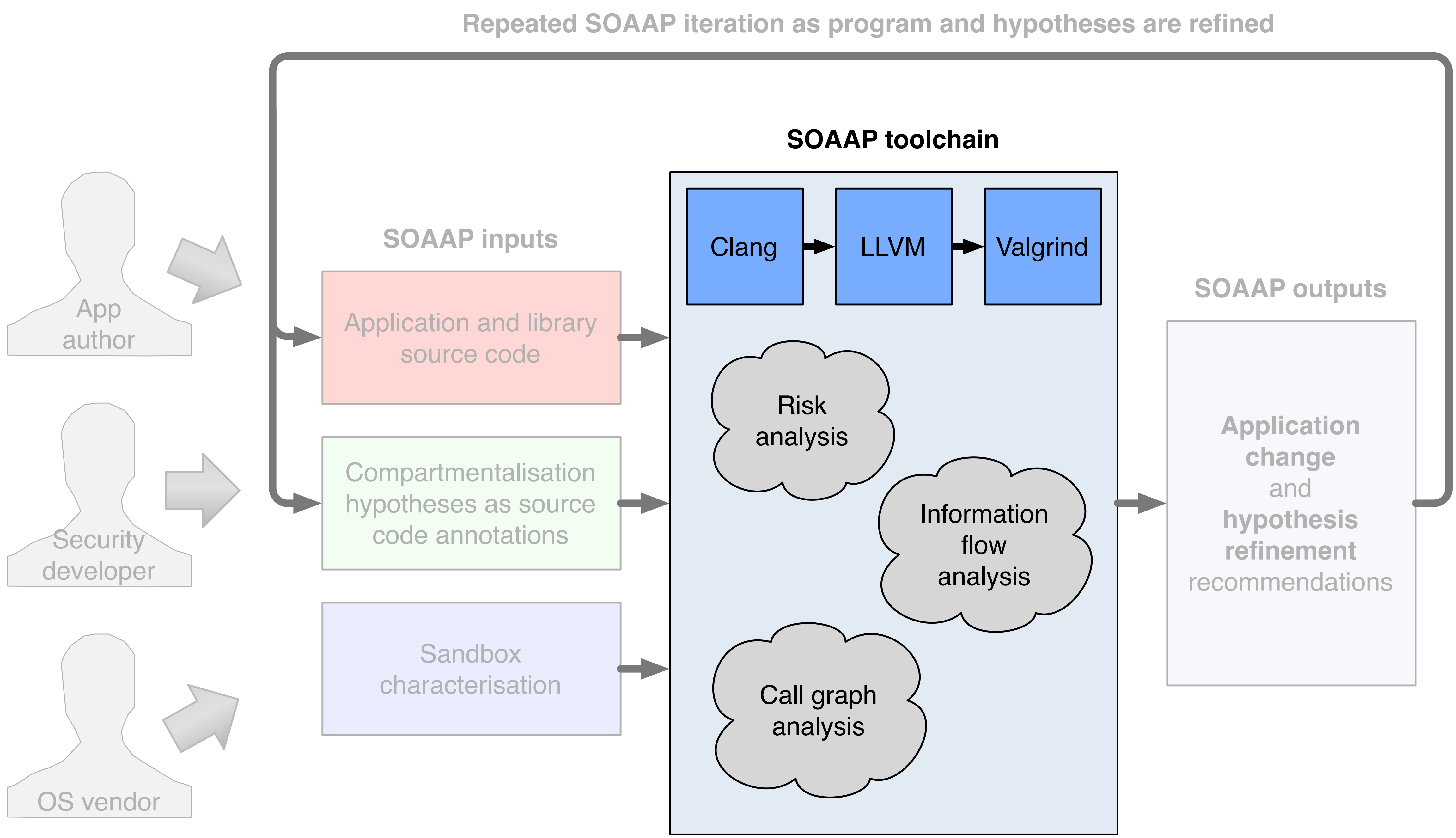
Security-oriented analysis of application programs (SOAAP)



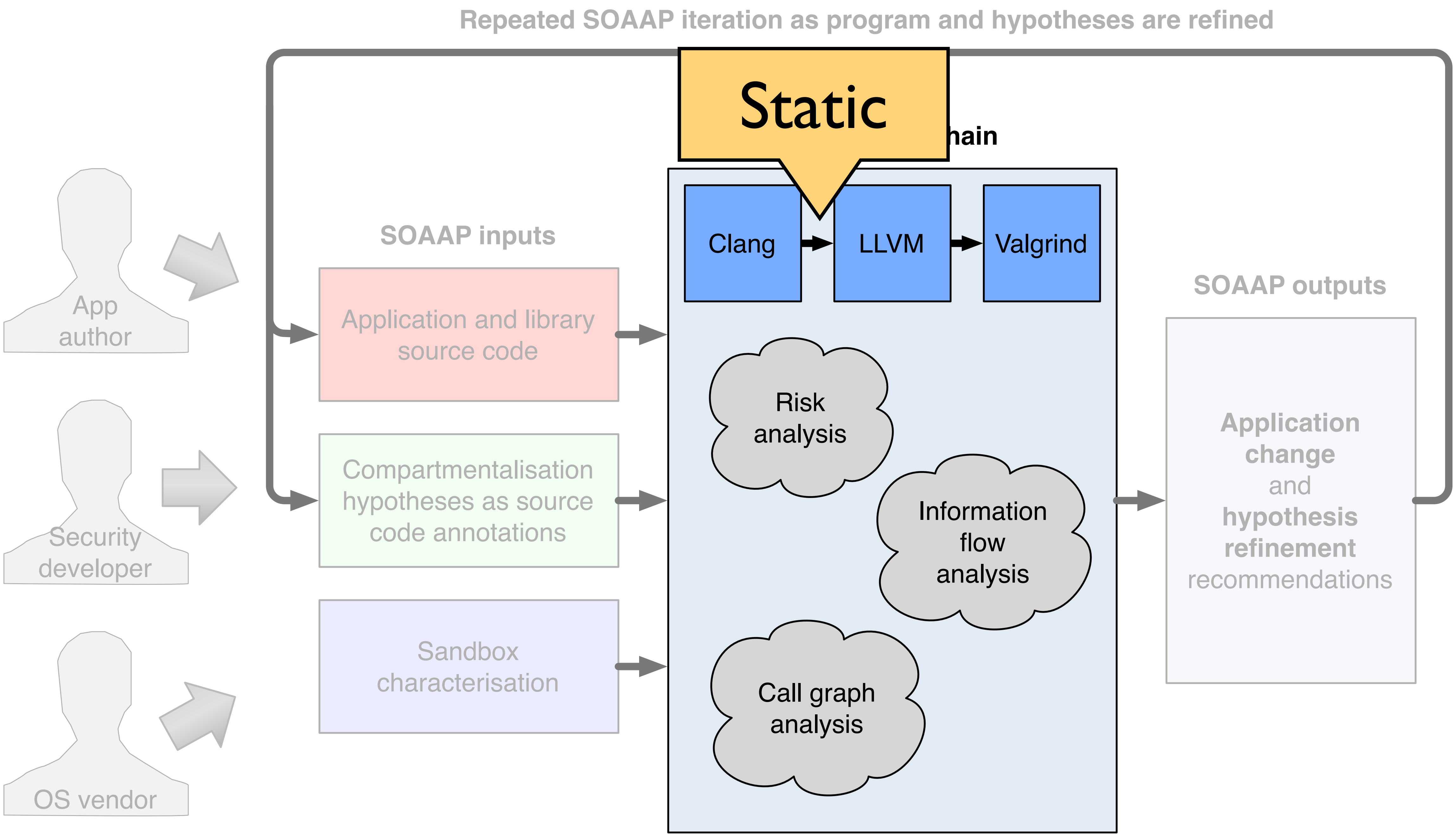
Security-oriented analysis of application programs (SOAAP)



Security-oriented analysis of application programs (SOAAP)

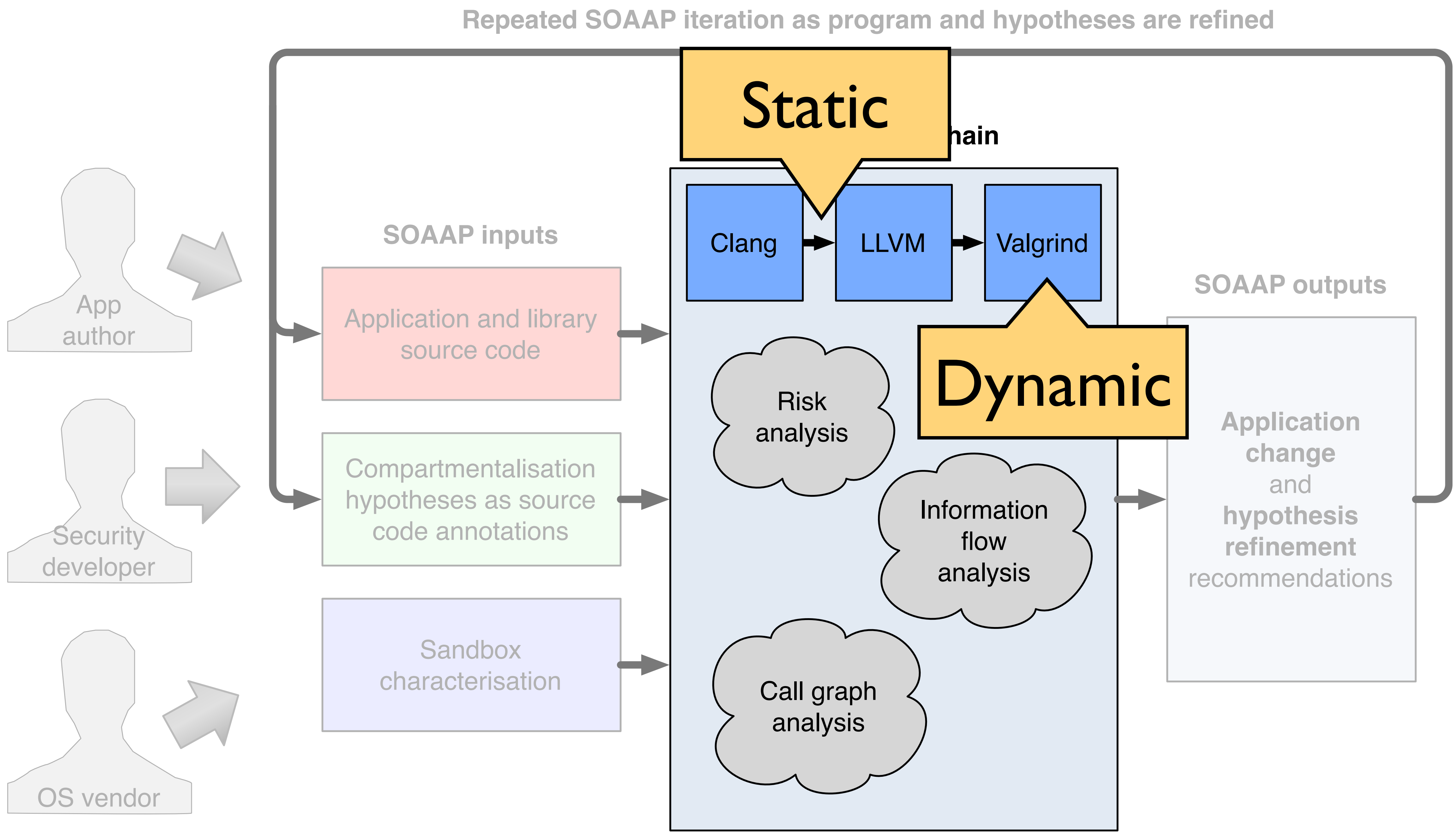


Security-oriented analysis of application programs (SOAAP)

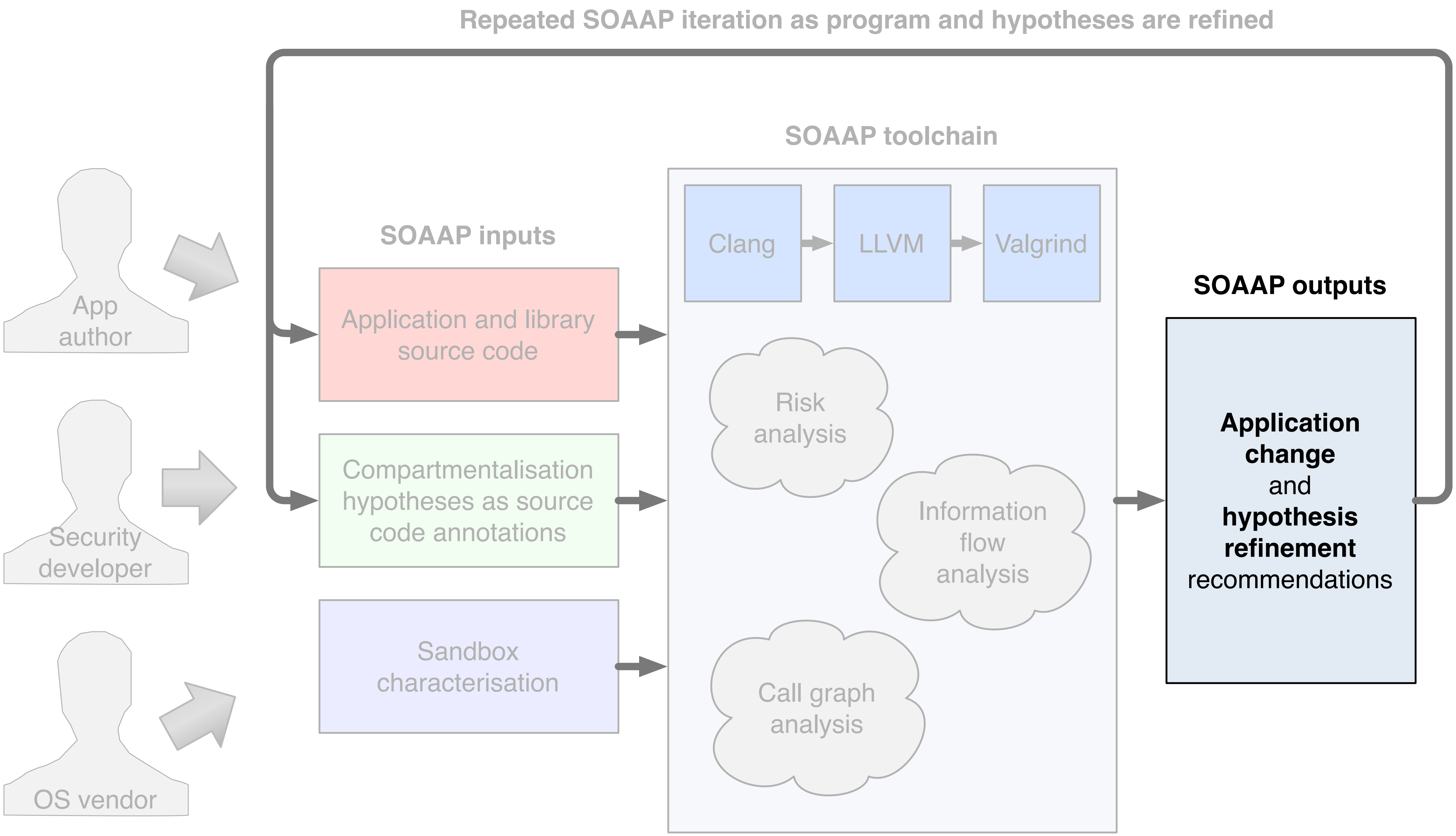


UNIVERSITY OF CAMBRIDGE

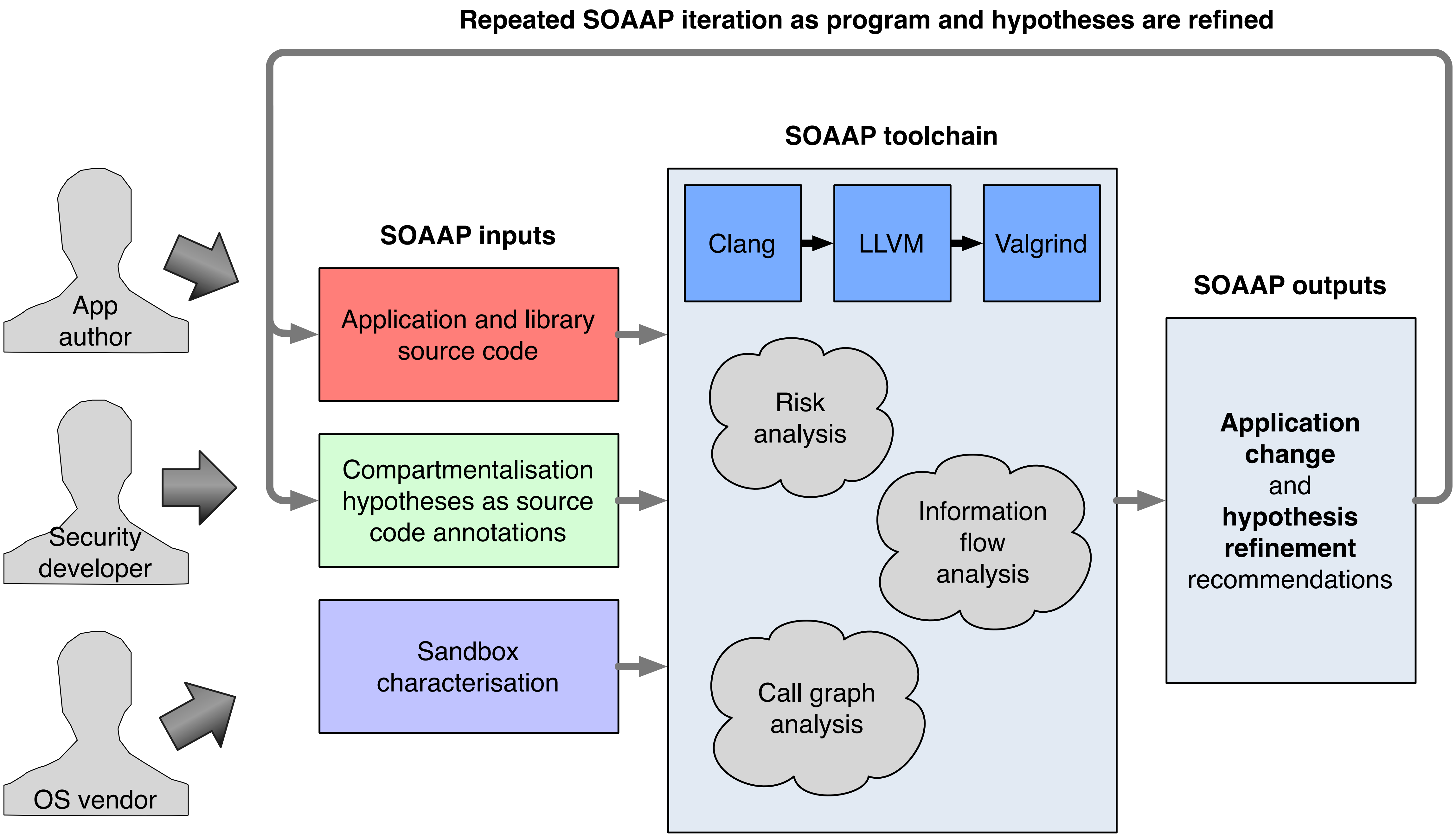
Security-oriented analysis of application programs (SOAAP)



Security-oriented analysis of application programs (SOAAP)



Security-oriented analysis of application programs (SOAAP)



Example

- Let's compartmentalise FreeBSD's gzip
- As a first pass, hypothetically sandbox the gz_compress() function
- Compile with our modified Clang/LLVM and run with our modified Valgrind

Example

```c
/* compress input to output. Return bytes read, -1 on error */
__sandbox_persistent
static off_t
gz_compress(int in, int out, off_t *gsizep, const char *origname, uint32_t mtime)
{
    z_stream z;
    char *outbufp, *inbufp;
    off_t in_tot = 0, out_tot = 0;
    ssize_t in_size;
```

Example

"What would happen if I were to sandbox gz_compress()?"

```
/* compress input to output. Return bytes read, -1 on error */
__sandbox_persistent
static off_t
gz_compress(int in, int out, off_t *gsizep, const char *origname, uint32_t mtime)
{
	z_stream z;
	char *outbufp, *inbufp;
	off_t in_tot = 0, out_tot = 0;
	ssize_t in_size;
```

Example

Sandbox read global variable "numflag" in method gz_compress, but it is not allowed to.
==9336==   at 0x804C27D: gz_compress (gzip.c:581)
==9336==   by 0x804B5D6: handle_file (gzip.c:1283)
==9336==   by 0x804A279: main (gzip.c:1817)

Example

Unspecified global variable access may lead to a bug

Sandbox read global variable "numflag" in method gz_compress, but it is not allowed to.
==9336==   at 0x804C27D: gz_compress (gzip.c:581)
==9336==   by 0x804B5D6: handle_file (gzip.c:1283)
==9336==   by 0x804A279: main (gzip.c:1817)

Example

Unspecified global variable access may lead to a bug

Sandbox read global variable "numflag" in method gz_compress, but it is not allowed to.

==9336== at 0x804C27D: gz_compress (gzip.c:581)

==9336== by 0x804B5D6: handle_file (gzip.c:1283)

==9336== by 0x804A279: main (gzip.c:1817)

Stack trace so programmer can pinpoint

Example

```c
static  int dflag;              /* decompress mode */
static  int lflag;              /* list mode */
static  int __var_read numflag = 6;     /* gzip -1..-9 value */

#ifndef SMALL
```

Example

Annotate **numflag** as
readable from sandboxes

```
static  int dflag;              /* decompress mode */
static  int lflag;              /* list mode */
static  int __var_read numflag = 6;     /* gzip -1..-9 value */

#ifndef SMALL
```


UNIVERSITY OF
CAMBRIDGE

Example

Global variable "numflag" is being written to in method main after a sandbox has been created and so the sandbox will not see this new value.

==9381==   at 0x8049F83: main (gzip.c:329)

Example

Write to global variable outside the sandbox will not be propagated

Global variable "numflag" is being written to in method main after a sandbox has been created and so the sandbox will not see this new value.
==9381==   at 0x8049F83: main (gzip.c:329)

Example

Write to global variable outside the sandbox will not be propagated

Global variable "numflag" is being written to in method main after a sandbox has been created and so the sandbox will not see this new value.

==9381== at 0x8049F83: main (gzip.c:329)

```
324     while ((ch = getopt_long(argc, argv, OPT_LIST, longopts, NULL)) != -1) {
325         switch (ch) {
326         case '1': case '2': case '3':
327         case '4': case '5': case '6':
328         case '7': case '8': case '9':
329             numflag = ch - '0';
330             break;
```

Example

Write to global variable outside the sandbox will not be propagated

Global variable "numflag" is being written to in method main after a sandbox has been created and so the sandbox will not see this new value.
==9381== at 0x8049F83: main (gzip.c:329)

```
324     while ((ch = getopt_long(argc, argv, OPT_LIST, longopts, NULL)) != -1) {
325         switch (ch) {
326         case '1': case '2': case '3':
327         case '4': case '5': case '6':
328         case '7': case '8': case '9':
329             numflag = ch - '0';
330             break;
```

Write to **numflag**

Example

Sandbox read from /usr/home/khilan/nfs/bsd_src/usr.bin/gzip/test.txt (fd: 3) in method __sys_read, but it is not allowed to.

```
==9381==    at 0x172D03: __sys_read (in /lib/libc.so.7)
==9381==    by 0x804B5E6: handle_file (gzip.c:1283)
==9381==    by 0x804A289: main (gzip.c:1817)
```

Sandbox wrote to /usr/home/khilan/nfs/bsd_src/usr.bin/gzip/test.txt.gz (fd: 4) in method __sys_write, but it is not allowed to.

```
==9381==    at 0x172CE3: __sys_write (in /lib/libc.so.7)
==9381==    by 0x804B5E6: handle_file (gzip.c:1283)
==9381==    by 0x804A289: main (gzip.c:1817)
```

Unspecified file read. Sandbox is accessing a resource it does not have permission to.

Example

Sandbox read from /usr/home/khilan/nfs/bsd_src/usr.bin/gzip/test.txt (fd: 3) in method __sys_read, but it is not allowed to.

```
==9381==    at 0x172D03: __sys_read (in /lib/libc.so.7)
==9381==    by 0x804B5E6: handle_file (gzip.c:1283)
==9381==    by 0x804A289: main (gzip.c:1817)
```

Sandbox wrote to /usr/home/khilan/nfs/bsd_src/usr.bin/gzip/test.txt.gz (fd: 4) in method __sys_write, but it is not allowed to.

```
==9381==    at 0x172CE3: __sys_write (in /lib/libc.so.7)
==9381==    by 0x804B5E6: handle_file (gzip.c:1283)
==9381==    by 0x804A289: main (gzip.c:1817)
```

Unspecified file read. Sandbox is accessing a resource it does not have permission to.

Example

Sandbox read from /usr/home/khilan/nfs/bsd_src/usr.bin/gzip/test.txt (fd: 3) in method __sys_read, but it is not allowed to.

```
==9381==    at 0x172D03: __sys_read (in /lib/libc.so.7)
==9381==    by 0x804B5E6: handle_file (gzip.c:1283)
==9381==    by 0x804A289: main (gzip.c:1817)
```

Sandbox wrote to /usr/home/khilan/nfs/bsd_src/usr.bin/gzip/test.txt.gz (fd: 4) in method __sys_write, but it is not allowed to.

```
==9381==    at 0x172CE3: __sys_write (in /lib/libc.so.7)
==9381==    by 0x804B5E6: handle_file (gzip.c:1283)
==9381==    by 0x804A289: main (gzip.c:1817)
```

Unspecified file write

Example

```
/* compress input to output. Return bytes read, -1 on error */
__sandbox_persistent
static off_t
gz_compress(int __fd_read in, int __fd_write out, off_t *gsizep, const char *origname, uint
{
    z_stream z;
    char *outbufp, *inbufp;
    off_t in_tot = 0, out_tot = 0;
```

Example

“Can read from file descriptor **in**”

```
/* compress input to output. Return bytes read, -1 on error */
__sandbox_persistent
static off_t
gz_compress(int __fd_read in, int __fd_write out, off_t *gsizep, const char *origname, uint
{
    z_stream z;
    char *outbufp, *inbufp;
    off_t in_tot = 0, out_tot = 0;
```

Example

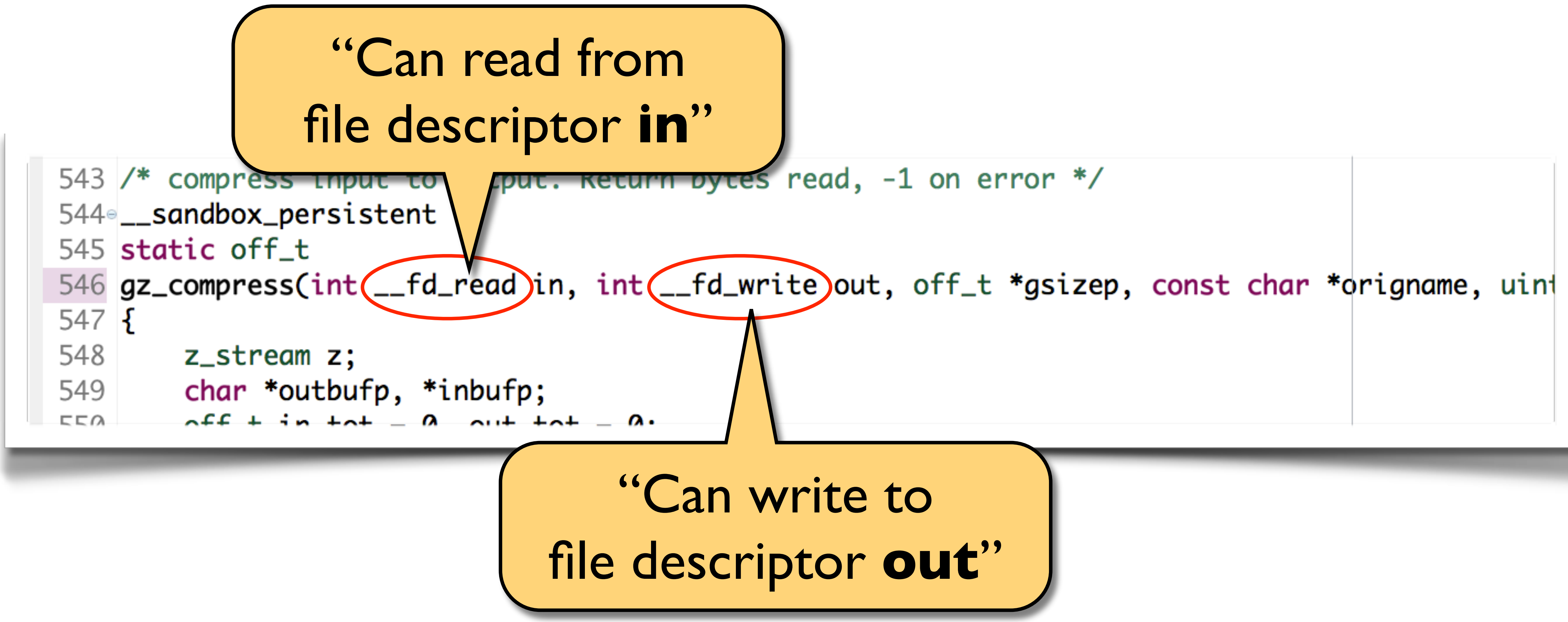
“Can read from file descriptor **in**”

```
/* compress input to output. Return bytes read, -1 on error */
__sandbox_persistent
static off_t
gz_compress(int __fd_read in, int __fd_write out, off_t *gsizep, const char *origname, uint
{
    z_stream z;
    char *outbufp, *inbufp;
    off_t in_tot = 0, out_tot = 0;
```

“Can write to file descriptor **out**”

UNIVERSITY OF CAMBRIDGE

Example



We are effectively annotating the program to use Capsicum: *sandboxes*, *delegated rights*, *call gates*, etc.

UNIVERSITY OF CAMBRIDGE

Advantages of SOAAP

- Validate functional correctness

Advantages of SOAAP

- Validate functional correctness
- Validate security requirements

Advantages of SOAAP

- Validate functional correctness
- Validate security requirements
  - Modulo different sandboxing technologies (e.g. Capsicum, seccomp, SELinux, chroot/setuid).

	OS	Sandbox	LoC	FS	IPC	NET	S≠S′	Priv
DAC	Windows	DAC ACLs	22,350	⚠	⚠	✘	✘	✔
	Linux	chroot()	600	✔	✘	✘	✔	✘
MAC	Mac OS X	Sandbox	560	✔	⚠	✔	✔	✔
	Linux	SELinux	200	✔	⚠	✔	✘	✘
Cap	Linux	seccomp	11,300	⚠	✔	✔	✔	✔
	FreeBSD	Capsicum	100	✔	✔	✔	✔	✔

Advantages of SOAAP

- Validate functional correctness
- Validate security requirements
  - Modulo different sandboxing technologies (e.g. Capsicum, seccomp, SELinux, chroot/setuid).
- Incremental sandboxing and testing

UNIVERSITY OF CAMBRIDGE

Advantages of SOAAP

- Validate functional correctness
- Validate security requirements
  - Modulo different sandboxing technologies (e.g. Capsicum, seccomp, SELinux, chroot/setuid).
- Incremental sandboxing and testing
- Trade-off exploration

Advantages of SOAAP

- Validate functional correctness
- Validate security requirements
  - Modulo different sandboxing technologies (e.g. Capsicum, seccomp, SELinux, chroot/setuid).
- Incremental sandboxing and testing
- Trade-off exploration
- Can also validate the correctness/security of already compartmentalised programs

Future plans

- **Confidentiality** - employ information flow analyses to validate flows for *sensitive* data
- **Risk** - automate the classification of risky code, e.g. machine learning, fuzzbuster
- **Sandbox characterisations**
- Apply SOAAP annotations to already-compartmentalised software

Proposed Evaluation

- How do false positive and negative rates arising out of the unsoundness of C-language program analysis affect the user experience?

Proposed Evaluation

- How do false positive and negative rates arising out of the unsoundness of C-language program analysis affect the user experience?
- When applied to a back catalogue of known compartmentalisation bugs, are all found, and if not, why not?

Proposed Evaluation

- How do false positive and negative rates arising out of the unsoundness of C-language program analysis affect the user experience?
- When applied to a back catalogue of known compartmentalisation bugs, are all found, and if not, why not?
- Are new bugs found in previously compartmentalised programs, illustrating the benefits of this approach?

Proposed Evaluation

- Once a viable and desirable compartmentalisation is identified, and then implemented by the programmer, are there other problems that arise?

Proposed Evaluation

- Once a viable and desirable compartmentalisation is identified, and then implemented by the programmer, are there other problems that arise?
- Do performance predictions made by SOAAP prove accurate?

Proposed Evaluation

- Once a viable and desirable compartmentalisation is identified, and then implemented by the programmer, are there other problems that arise?
- Do performance predictions made by SOAAP prove accurate?
- Can we scale up SOAAP-based exploration to both very large collections of programs, such as the footprint of a complete UNIX system, or individually large (monolithic) applications such as web browsers and mail clients?

Acknowledgements

- This work was sponsored by DARPA and Google
- Builds on taintgrind tool by Wei Ming Khoo
  - Originally designed for malware analysis...

UNIVERSITY OF CAMBRIDGE

Closing remarks

- Goal is to release the SOAAP tools as open source: http://github.com/CTSRD-SOAAP/
- Talk was well received at the recent FreeBSD developer summit held in Cambridge
  - Lots of interest from Capsicum developers
  - Already downloading and using SOAAP