04834580 Software Engineering (Honor Track) 2025-26

Software Security

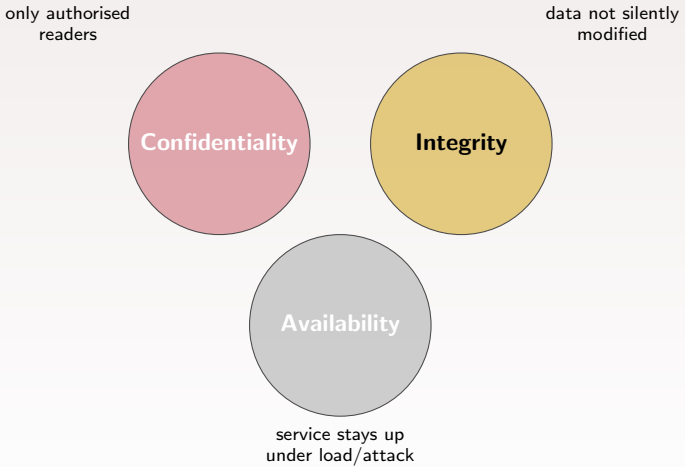
Sergey Mechtaev
mechtaev@pku.edu.cn
School of Computer Science, Peking University

Definition (Software Security)

The property that a system continues to behave as specified even in the presence of an **adversary** who may craft inputs, influence the environment, or observe the system in order to violate **confidentiality**, **integrity**, or **availability**.

Reliability asks: does it work when used *as expected*?
Security asks: does it stay correct when somebody is actively trying to break it?

The CIA Triad



Why Software Security Is Hard

- **Asymmetry.** The defender must close *every* hole; the attacker needs *one*.
- **Composition.** A safe library + a safe parser can still form an unsafe system at the seam.
- **Inputs are adversarial.** Tests cover what users *do*; attackers explore what users *would never* do.
- **Wrong defaults.** C's `gets`, MySQL's `mysql_query`, PHP's `eval` — all easy to use, all unsafe.
- **Old code is everywhere.** A vulnerability discovered in 2026 may live in a 1998 codebase that nobody dares to touch.

A Short History of Famous Incidents

- **1988 — Morris worm** [1]: a stack buffer overflow in BSD `fingerd` took down $\approx 10\%$ of the Internet of the time.
- **2014 — Heartbleed** [2]: a missing bounds check in OpenSSL's heartbeat extension leaked memory contents (including private keys) of millions of servers.
- **2017 — Equifax** [3]: an unpatched Apache Struts deserialization bug exposed personal data of ≈ 147M people.
- **Today**: Microsoft and Chromium independently report that $\approx 70\%$ of high-severity security bugs in their C/C++ codebases are *memory-safety* issues [4, 5].

Two Families of Vulnerabilities

This lecture focuses on the two largest classes:

Memory-safety bugs

buffer overflow
use-after-free
double-free
uninitialised read

root cause: language
lets you mis-use memory

Injection bugs

SQL injection
command injection
path traversal
XSS

root cause: untrusted
string used to build a program

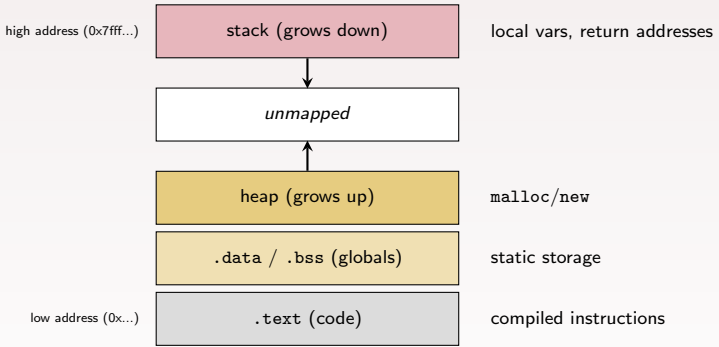
Definition (Memory Safety)

A program is memory-safe if every memory access reads or writes only objects that are:

- **Spatially valid** — within the bounds of an allocated object (no out-of-bounds), and
- **Temporally valid** — still alive at the time of the access (no use-after-free, no read-before-init).

C and C++ are *not* memory-safe by default. Java, Python, Go, C#, Swift, OCaml, and (safe) Rust are. The price paid by the latter is some combination of: a garbage collector, runtime bounds checks, or a stricter type system.

The C Process Memory Layout



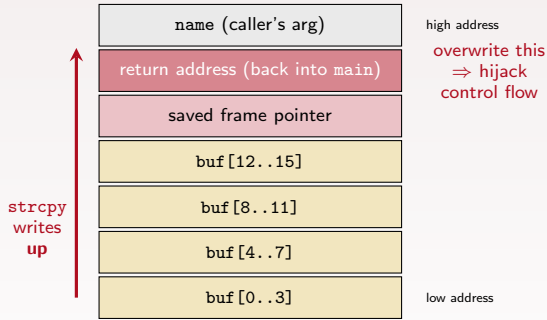
Stack Buffer Overflow

The most classic memory bug [6]:

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

void greet(char *name) {
    char buf[16];
    strcpy(buf, name);          // no length check!
    printf("Hello, %s\n", buf);
}

int main(int argc, char **argv) {
    greet(argv[1]);             // attacker controls argv[1]
}
```

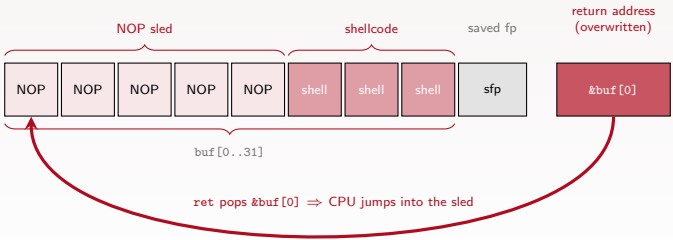
Pass a `name` longer than 16 bytes and `strcpy` happily writes past `buf`. The bytes *just after* `buf` on the stack are interesting: the saved frame pointer and the return address of `greet`.

Stack Frame Layout During `greet`



A naive overflow corrupts the return address and the program crashes when `ret` jumps to garbage. An attacker does *better*:

1. Place **shellcode** (raw machine instructions) inside `buf` itself, preceded by a **NOP sled**.
2. Overwrite the return address with (an approximation of) `&buf`.
3. When `greet` executes `ret`, the CPU loads that address, slides through the NOPs, and runs the shellcode.



Heap Buffer Overflow


```
char *user = malloc(8);
char *admin_flag = malloc(1);   // adjacent allocation
strcpy(user, "AAAAAAAAAAAA");   // 12 bytes into 8-byte buffer
// admin_flag now contains 'A' (true!)
```

Heap allocators place objects close together. An overflow on one object silently corrupts the metadata or contents of the next — often a function pointer, a vtable, or a security flag.

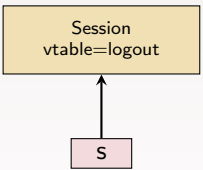
```
struct Session { void (*on_logout)(); };

Session *s = new Session{logout_handler};
delete s;                          // memory returned to allocator
// ... s is now a "dangling pointer" ...
s->on_logout();                    // executes arbitrary code if the
                                   // allocator reused that slot
```

Temporal memory error: the pointer was once valid but no longer is. Exploit pattern:

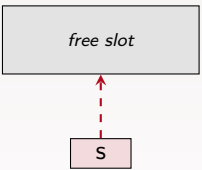
1. Free the victim object.
2. Trigger an allocation of the same size so the allocator returns the same slot, filled with attacker-controlled bytes (a fake vtable, a fake function pointer).
3. Wait for the dangling pointer to be dereferenced.

Use-After-Free Visualised

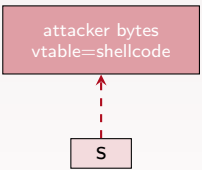
Step 1
(alive)



Step 2
(freed)



Step 3
(reused by attacker)



Double-Free

```
char *p = malloc(32);
free(p);
/* ... lots of code ... */
free(p);      // boom
```

The second `free` corrupts the allocator's internal linked lists. Classical `glibc` "`unlink`" exploits used this to make `malloc` return an arbitrary attacker-chosen address, which then overwrites a function pointer.

Doubles are easy to introduce when error-handling paths run cleanup code more than once, or when ownership of a pointer is unclear between two modules.

Heartbleed: A Real-World Memory Bug

OpenSSL 1.0.1 (2012) accepted a heartbeat message with an attacker-supplied length:

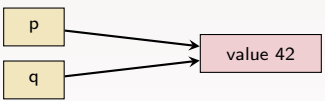
```
struct { uint8 type; uint16 length; opaque payload[length]; } HB;

/* simplified vulnerable code */
memcpy(response, request->payload, request->length);   // !!
```

The actual `payload` was usually a few bytes, but the attacker could put `length = 65535`. The server then copied 64 KB of *whatever happened to follow* the payload in memory — passwords, session cookies, even private keys — back to the attacker.

- One missing bounds check.
- Affected $\approx$ 17% of Internet-facing TLS servers.
- Cost estimated in hundreds of millions of USD.

Definition (Aliasing)

Two distinct names (variables, pointers, references) **alias** when they refer to the same memory location, so that a write through one is observable through the other.



`*p = 7;` also changes `*q`

Why Aliasing Is Dangerous

北京大学

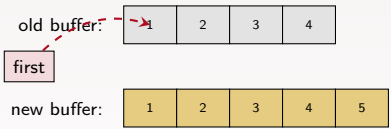
Aliasing makes it hard to reason about *any* of:

- **Correctness.** "`*p = 0; if (*q == 0) ...`" — the branch outcome depends on whether `p` and `q` alias.
- **Optimisation.** The compiler cannot keep `*q` in a register across a write through `*p`.
- **Memory safety.** If `a` owns a buffer and `b` aliases inside it, freeing through `a` leaves `b` dangling.
- **Concurrency.** Aliased writes from two threads is the textbook definition of a data race.

Iterator Invalidation: A Subtle Aliasing Bug

A canonical C++ trap:

```
std::vector<int> v = {1, 2, 3, 4};
int &first = v[0];              // reference into v's buffer
v.push_back(5);                 // may reallocate -> moves buffer
std::cout << first;             // dangling reference, UB
```



`first` was an alias of `v`'s storage; `push_back` reallocated; `first` now points into freed memory.

Aliasing also costs performance. C lets the programmer promise "no aliasing" with the `restrict` keyword:

```
void add(int *restrict a, int *restrict b, int *restrict c, int n) {
    for (int i = 0; i < n; i++) c[i] = a[i] + b[i];
}
```

Without `restrict`, the compiler must conservatively assume that a write to `c[i]` could alter `a[i+1]` (because `c` and `a` might point into the same array). This prevents vectorisation and loop reordering.

But `restrict` is a *promise*: the compiler trusts it and emits broken code if you lie. Aliasing rules in C/C++ are unchecked.

The Deep Problem

Many memory-safety bugs reduce to one underlying question:

"Who is allowed to read or write this location, and when?"

C/C++ leaves the answer in the programmer's head. Rust puts the answer **in the type system**.

Rust's Bet

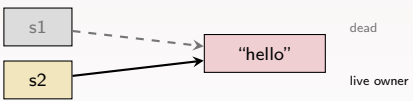
Rust [7] keeps C/C++'s performance model (no garbage collector, predictable allocations) but adds a compile-time discipline on *aliasing* and *lifetimes* that rules out, by construction, the bug classes we have just surveyed: dangling pointers, double-free, iterator invalidation, data races between threads.

Three core ideas, all enforced statically:

- **Ownership** — each value has exactly one owner.
- **Borrowing** — temporary references obey strict aliasing rules.
- **Lifetimes** — the compiler tracks how long each reference is allowed to live.

```rust
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;              // ownership MOVED from s1 to s2
    println!("{}", s1);       // compile error: s1 no longer valid
}
```

When a value is assigned, passed, or returned, its ownership *moves*. The previous binding is statically invalidated.



When the owner goes out of scope, the value is dropped exactly once. No leaks, no double-free.

Borrowing: The Aliasing Rule

北京大学

References are temporary, non-owning, and obey one rule:

At any program point, every memory location has either:

- **exactly one mutable reference** (`&mut T`), or
- **any number of shared references** (`&T`),

but never both.

```
let mut s = String::from("hi");
let r1 = &s;         // shared borrow
let r2 = &s;         // another shared borrow -- OK
let r3 = &mut s;     // ERROR: cannot also borrow mutably
```

Why the Rule Is Enough

The two cases of aliasing are precisely the two cases that cause trouble:

Pattern	In C/C++	In Rust
read + read	fine	fine (many `&T`)
write alone	fine	fine (one `&mut T`)
write + read	data race / UB	**rejected at compile time**
write + write	data race / UB	**rejected at compile time**

If you have a `&mut T` you are the sole observer of `T` for the duration of the borrow. Compiler optimisations can rely on this just like C's `restrict` — but unlike `restrict`, the compiler verifies it.

Iterator Invalidation, Revisited

The C++ bug from before, written in Rust:

```
let mut v = vec![1, 2, 3, 4];
let first = &v[0];       // shared borrow of v
v.push(5);               // wants &mut v -- ERROR
println!("{}", first);
```

The compiler refuses: `push` needs `&mut v`, but `first` is still alive as `&v`. The two cannot coexist. The category of bug is *gone*.

Lifetimes

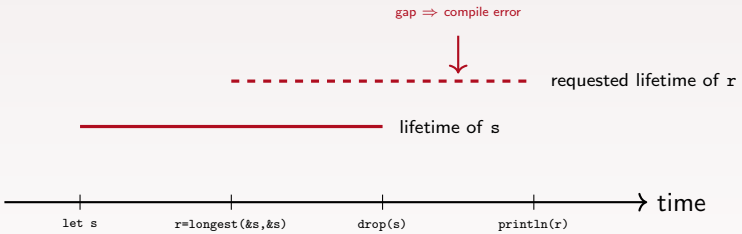
Lifetimes are the compiler's way of asking: "how long does this reference need to be valid?" Every reference has one, usually inferred.

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}
```

`'a` says: the returned reference lives as long as the shorter of `x` and `y`. The compiler then refuses any caller that would use the result after either input is gone.

```
let r;
{
    let s = String::from("hi");
    r = longest(&s, &s);     // r tied to s's lifetime
}
println!("{}", r);           // ERROR: s has been dropped
```

Lifetimes Visualised



```rust
fn dangling() -> &String {
    let s = String::from("hi");
    &s          // ERROR: s dropped at end of function;
                // reference would dangle
}
```

Rust's lifetimes turn the very *shape* of a dangling-pointer bug into a type error. There is no run-time check, and no run-time cost — the check happens at compile time.

The same machinery rules out the `double-free` pattern (because at the second free, the owner has already been moved) and the `use-after-free` pattern (because the borrow is tied to the owner's lifetime).

Cost Model: C/C++ vs Rust

	C/C++	Safe Rust
out-of-bounds access	silent UB	panic (runtime check)
use-after-free	silent UB	compile error
double-free	silent UB	compile error
null deref	silent UB	no nulls (`Option`)
uninit memory	silent UB	compile error
data race	silent UB	compile error
aliasing for optim.	`restrict` (unchecked)	enforced by borrow checker
runtime overhead	none	bounds checks only
GC	none	none

Safe Rust pays for safety at compile time (a learning curve and some refactoring) rather than at runtime.

The Escape Hatch: `unsafe`

Some low-level operations (raw FFI, custom allocators, lock-free data structures) cannot be expressed within the borrow checker. Rust does not prohibit them; it isolates them.

```
unsafe {
    let raw = ptr as *mut u32;
    *raw = 42;          // I, the programmer, promise this is fine
}
```

- `unsafe` blocks are syntactically obvious: they delineate the trusted core.
- A library can wrap `unsafe` in a safe API; users never touch the dangerous primitives. `Vec`, `Mutex`, `Rc` are all built this way.
- Auditing focuses on the small `unsafe` surface rather than the whole codebase [8].

Definition (Injection)

A vulnerability in which untrusted input is concatenated into a **program text** (SQL, shell, HTML, a path) that is later interpreted, so that part of the input is mistakenly treated as *instructions* rather than *data*.

Injection has topped the OWASP Top Ten [9] for over a decade. Variants:

- SQL injection
- OS command injection
- Path traversal
- Cross-site scripting (HTML / JS injection)
- LDAP, XPath, template, log injection ...

SQL Injection: Vulnerable Code

北京大学

```
def login(username, password):
    q = ("SELECT * FROM users "
         "WHERE name = '" + username + "' "
         "AND  pass = '" + password + "'")
    return db.execute(q).fetchone()
```

Looks fine for `username = "alice"`, `password = "hunter2"`:

```
SELECT * FROM users WHERE name = 'alice' AND pass = 'hunter2'
```

But what if `username = "alice' --"`?

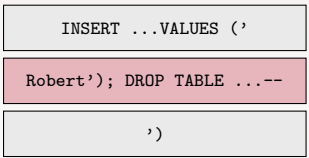
```
SELECT * FROM users WHERE name = 'alice' --' AND pass = '...'
```

Everything after `--` is a comment. Password check skipped, attacker logs in as `alice`.

Bobby Tables (xkcd #327)

The canonical illustration [10]:

```
INSERT INTO students (name) VALUES ('Robert'); DROP TABLE students;--'
```

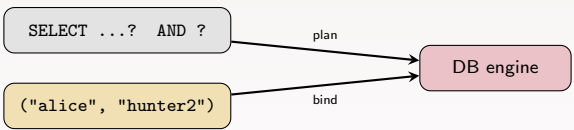


trusted code *plus*
untrusted input
⇒ becomes more code

Root cause: building a program (SQL) by string concatenation with untrusted input.

```
def login(username, password):
    q = "SELECT * FROM users WHERE name = ? AND pass = ?"
    return db.execute(q, (username, password)).fetchone()
```

The driver sends *query* and *parameters* to the database on **separate channels**. The DB plans the query first, then binds the values — they are forever data, never code.



This is also faster: the DB caches the plan and reuses it across calls with different parameters.

OS Command Injection

北京大学

```
def make_thumbnail(filename):
    os.system("convert " + filename + " thumb.png")

make_thumbnail("photo.jpg ; rm -rf /")
```

`os.system` runs the string through the shell. `; rm -rf /` is a perfectly legal shell sequence.

The fix mirrors SQL: keep argument vector and program name on separate channels.

```
import subprocess
subprocess.run(["convert", filename, "thumb.png"], check=True)
```

No shell involved; `filename` is an argv element, not code.

```
def download(name):
    path = "/var/www/files/" + name
    return open(path).read()

download("../../etc/passwd")    # escapes the directory
```

Symptom: untrusted user input enters a privileged file API. Defence: `realpath` + check that the result is still under the intended base directory; reject `..` components; use OS primitives like `openat` with `O_NOFOLLOW`.

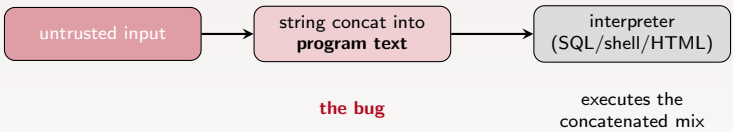
```python
@app.route("/search")
def search():
    q = request.args["q"]
    return "<p>You searched for: " + q + "</p>"
```

Browser later receives:

```
<p>You searched for: <script>steal(document.cookie)</script></p>
```

The attacker has injected JavaScript that runs in any visitor's browser. The fix: **HTML-escape** untrusted strings (or use a template engine that does it by default — Jinja2, React, Vue all escape by default).

Every injection bug fits the same template:

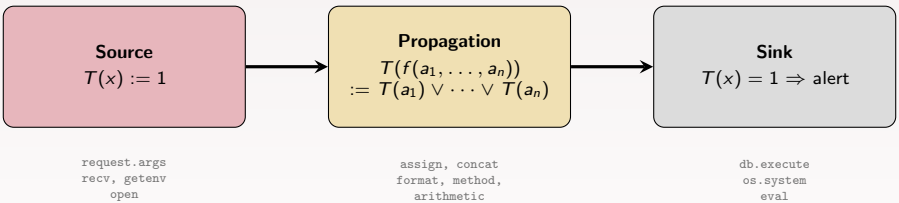


The cure is always the same: separate *code* from *data* through structured APIs (prepared statements, argv arrays, DOM manipulation, escaping templates) instead of string concatenation.

We want a tool that flags any program where untrusted input may reach a dangerous interpreter. This is exactly the source–sink problem solved by **taint analysis** [11].

- **Source.** Anything the attacker controls: HTTP parameters, request headers, cookies, file uploads, DB rows containing earlier user input.
- **Sink.** Anything that interprets a string as code: `db.execute`, `os.system`, `eval`, HTML response body.
- **Sanitiser.** A function that scrubs taint: `html_escape`, `parameterise()`, `shlex.quote`.

The instrumented runtime keeps a **shadow store**: every value carries a 1-bit *taint tag*, updated on every operation.



A Worked Example: Vulnerable Search Handler

```
def search(req):
    q = req.args["q"]                                  # (1) SOURCE
    needle = q.strip()                                 # (2) propagation
    sql = "SELECT * FROM t WHERE n='" + needle + "'"   # (3) propagation
    return db.execute(sql)                             # (4) SINK
```

We will trace the shadow store line by line. Assume the attacker sends the query parameter `q = "x' OR 1=1--"` — a classic SQL-injection payload.

Initially the shadow store is empty; no value is tainted.

Step 1 — Source: HTTP Parameter Read


```
q = req.args["q"]
```

The runtime intercepts the read from `req.args` (a registered source) and tags the result as tainted: $T(q) := 1$.

variable	value	T
q	"x' OR 1=1--"	1

```
needle = q.strip()
```

`str.strip` reads from the tainted receiver `q` and produces a new string.
Propagation rule: $T(\text{needle}) := T(q) = 1$.

variable	value	T
`q`	`"x' OR 1=1--"`	1
`needle`	`"x' OR 1=1--"`	1

```
sql = "SELECT * FROM t WHERE n='" + needle + "'"
```

+ on strings is a binary operator with two operands. At least one (needle) is tainted, so: $T(\text{sql}) := T(\text{"..."}) \vee T(\text{needle}) \vee T(\text{"'"}) = 1$.

variable	value	T
q	"x' OR 1=1--"	1
needle	"x' OR 1=1--"	1
sql	"SELECT * FROM t WHERE n='x' OR 1=1--'"	1

```
db.execute(sql)
```

`db.execute` is a registered sink. Before forwarding the call to the SQL engine, the runtime inspects $T(\text{sql})$. It is $1 \Rightarrow$ **taint alert** (and, depending on policy, abort or just log).

```
[TAINT ALERT] tainted value reached sink db.execute
  sink:    search.py:5
  value:   "SELECT * FROM t WHERE n='x' OR 1=1--'"
  origin:  req.args["q"] at search.py:2
  trail:   q → needle (str.strip) → sql (concat)
```

References I

[1] Wikipedia contributors.
Morris worm.
https://en.wikipedia.org/wiki/Morris_worm, 2025.
[Online; accessed 18-May-2026].

[2] Wikipedia contributors.
Heartbleed.
https://en.wikipedia.org/wiki/Heartbleed, 2025.
[Online; accessed 27-Jan-2025].

[3] Wikipedia contributors.
2017 Equifax data breach.
https://en.wikipedia.org/wiki/2017_Equifax_data_breach, 2025.
[Online; accessed 18-May-2026].

References II

[4] Matt Miller.
Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape.
BlueHat IL 2019 Conference, 2019.

[5] The Chromium Project.
Memory safety.
https://www.chromium.org/Home/chromium-security/memory-safety/, 2024.

[6] Aleph One.
Smashing the stack for fun and profit.
Phrack Magazine, 7(49), 1996.

[7] Steve Klabnik and Carol Nichols.
The Rust Programming Language.
No Starch Press, 2nd edition, 2023.

References III

[8] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer.
RustBelt: Securing the foundations of the rust programming language.
In *Proceedings of the ACM on Programming Languages (POPL)*, volume 2, pages 1–34, 2018.

[9] OWASP Foundation.
OWASP Top 10.
https://owasp.org/www-project-top-ten/, 2021.
[Online; accessed 18-May-2026].

[10] Randall Munroe.
Exploits of a mom (xkcd #327).
https://xkcd.com/327/, 2007.
[Online; accessed 18-May-2026].

References IV

[11] Edward J Schwartz, Thanassis Avgerinos, and David Brumley.
All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask).
In *IEEE Symposium on Security and Privacy (S&P)*, pages 317–331, 2010.