Constantine

Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization

CCS 2021

Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, Cristiano Giuffrida

Sapienza University of Rome - Vrije Universiteit Amsterdam

November 16, 2021

Constantine

Automatically harden programs against side-channel attacks

Low overhead

- carefully designed optimizations
- 16% over real-world benchmarks

Strong security guarantees

completely linearize instructions and memory accesses

High compatibility

- loops, raw pointers, arbitrary types
- handle real-world crypto libraries

Open source

https://github.com/pietroborrello/constantine

Software Side Channel Attacks

Allow attackers to leak information from victim execution by observing changes in the microarchitectural state.

e.g.

- Time taken for a program to execute
- Instruction count
- Cache effects (FLUSH+RELOAD, PRIME+PROBE)
- Port Contention

Do not consider CPU bugs & Transient Execution Attacks [36]

Constant Time Programming

Eliminate any secret dependent computation:

- Secret dependent branches (Control Flow)
- Secret dependent memory accesses, data operand-dependent latencies (Data Flow)

→ Any observable computation of the program does not depend on secret data

Daunting and error prone task if done manually

Automatically transform programs into their constant-time equivalents

Existing Solutions

Control Flow:

- Branch Balancing [45]
- Transactional Execution [53]
- Predicated Execution [20]

→ Violate data flow invariants

→ **Control Flow Linearization**

Data Flow:

- Cache-line preloading [62, 80]
- Oblivious RAM [53]

→ Do not consider active attackers

→ High overhead

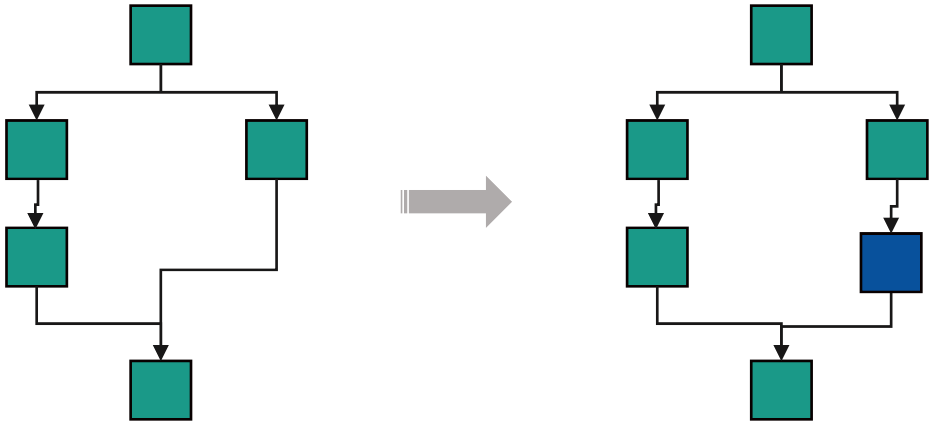
→ **Data Flow Linearization**

Existing Solutions

Control Flow:

- **Branch Balancing** [45]
- Transactional Execution [53]
- Predicated Execution [20]

→ Violate data flow invariants



- instruction cache
- port contention

Data Flow:

- Cache-line preloading [62, 80]
- Oblivious RAM [53]

→ Do not consider active attackers

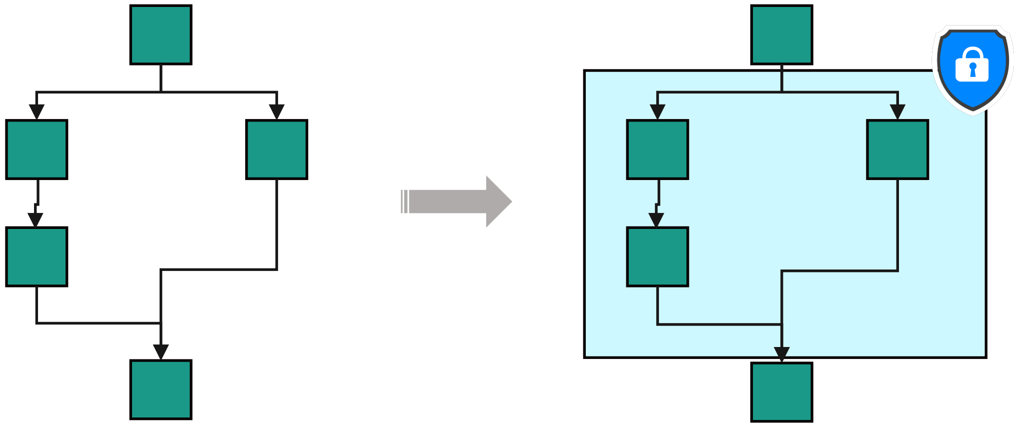
→ High overhead

Existing Solutions

Control Flow:

- Branch Balancing [45]
- **Transactional Execution** [53]
- **Predicated Execution** [20]

→ Violate data flow invariants



- may crash the program
- may break invariants on decoy paths
- is it enough to unroll loops?

Data Flow:

- Cache-line preloading [62, 80]
- Oblivious RAM [53]

→ Do not consider active attackers

→ High overhead

Existing Solutions

Control Flow:

- Branch Balancing [45]
- Transactional Execution [53]
- Predicated Execution [20]

→ Violate data flow invariants

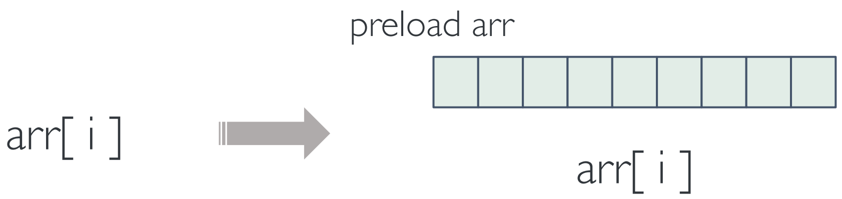
Data Flow:

- **Cache-line preloading** [62, 80]
- Oblivious RAM [53]

→ Do not consider active attackers

→ High overhead

- FLUSH+RELOAD
- bigger objects may not fit
- what about arbitrary pointers?



Existing Solutions

Control Flow:

- Branch Balancing [45]
- Transactional Execution [53]
- Predicated Execution [20]

→ Violate data flow invariants

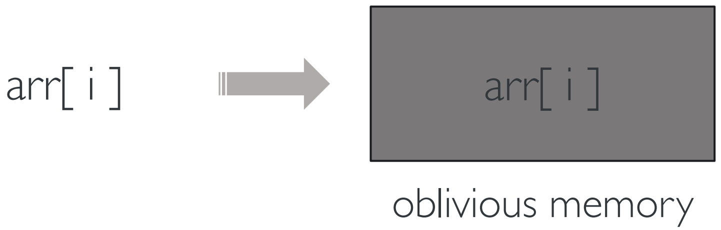
- 16x overhead (up to 1000x)
- may crash on arbitrary pointers

Data Flow:

- Cache-line preloading [62, 80]
- **Oblivious RAM** [53]

→ Do not consider active attackers

→ High overhead



Constantine

Automatically harden programs against microarchitectural side channels during compilation

→ Introduce radical abstractions around value computations

Control Flow Linearization

Yield secret invariant instruction traces

Data Flow Linearization

Touch all the possible locations an instruction may reference

Constantine

Control Flow Linearization

Yield secret invariant instruction traces

- how to prevent crashing?
- how to minimize the overhead?
- how to avoid state explosion?

Data Flow Linearization

Touch all the possible locations an instruction may reference

Constantine

- how to prevent crashing?
- how to minimize the overhead?
- how to avoid state explosion?

Control Flow Linearization

Yield secret invariant instruction traces

- which branch to linearize?
- what about loops?

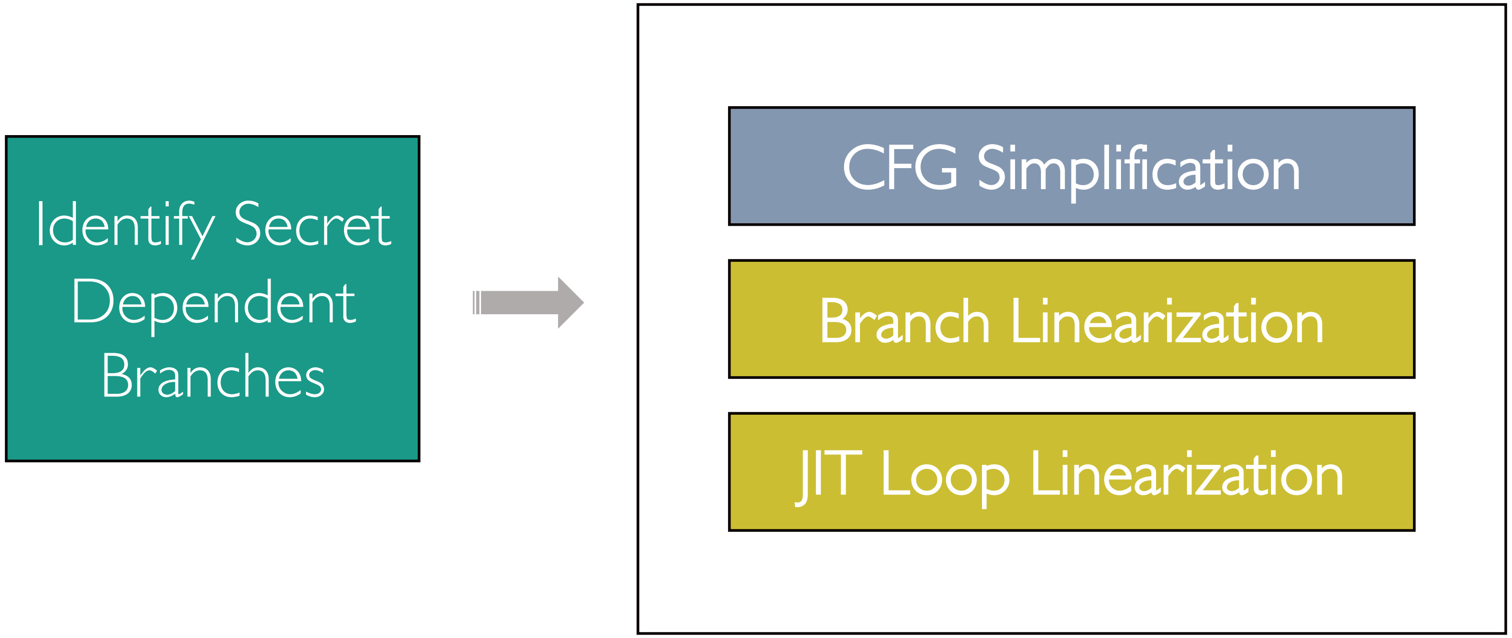
Data Flow Linearization

Touch all the possible locations an instruction may reference

- how to find such locations?
- how to protect from active attackers?

Control Flow Linearization

The sequence of secret-dependent instructions that the CPU executes is constant for any initial input (*Program Counter Security*) and secret data do not affect the latency of each such instruction.



Identify Secret Dependent Branches

Dynamic Taint Analysis (DFSan) provides information on sensitive program portions that depend on inputs:

Taint Sources:

- input
- tainted variables

Taint Sinks:

- branches
- memory accesses

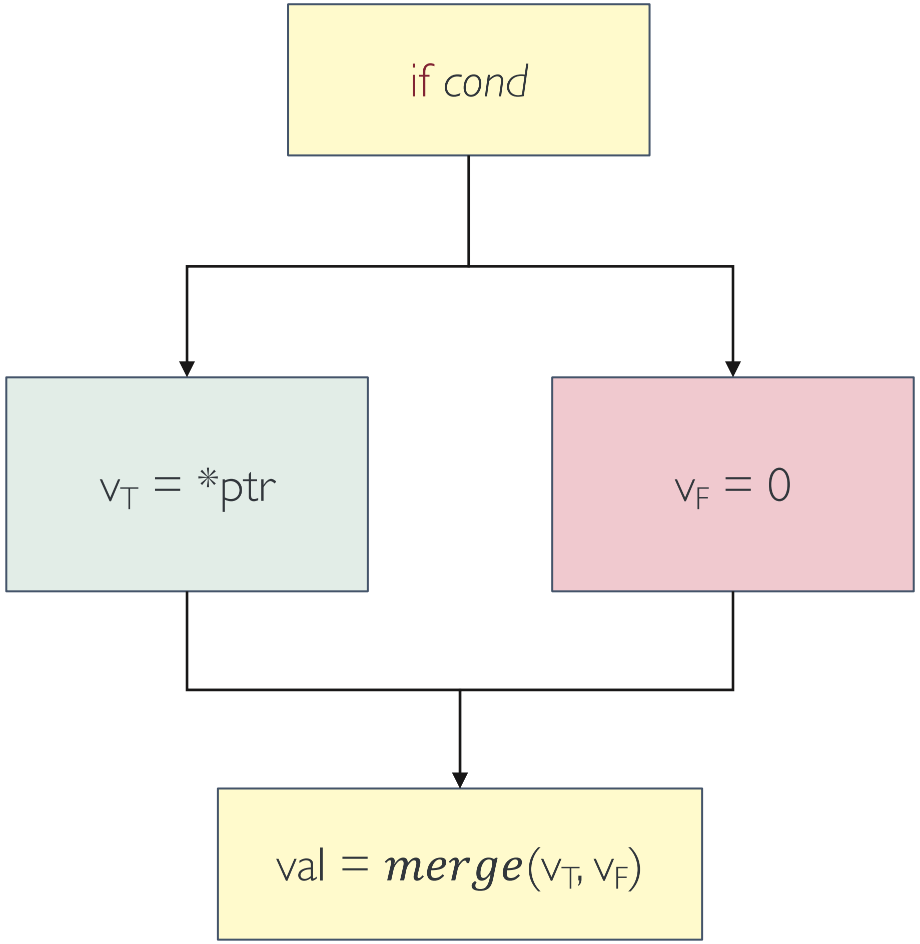
```
int tls1_cbc_remove_padding(const SSL *s,
                            SSL3_RECORD *rec,
                            unsigned bs, unsigned mac_size)
{
  int ii, i, j;
  int l = rec->length;
  ii=i=rec->data[l-1]; /* padding_length */
  i++;
  if (s->options&SSL_OP_TLS_BLOCK_PADDING_BUG)
    {
      /* First packet is even in size, so check */
      if ((*(unsigned long*)s->s3->read_sequence == 0) && !(ii & 1))
        s->s3->flags|=TLS1_FLAGS_TLS_PADDING_BUG;
      if (s->s3->flags & TLS1_FLAGS_TLS_PADDING_BUG)
        i--;
    }
```

Leverage a profiling phase over the test suite of the original program to gather taint information

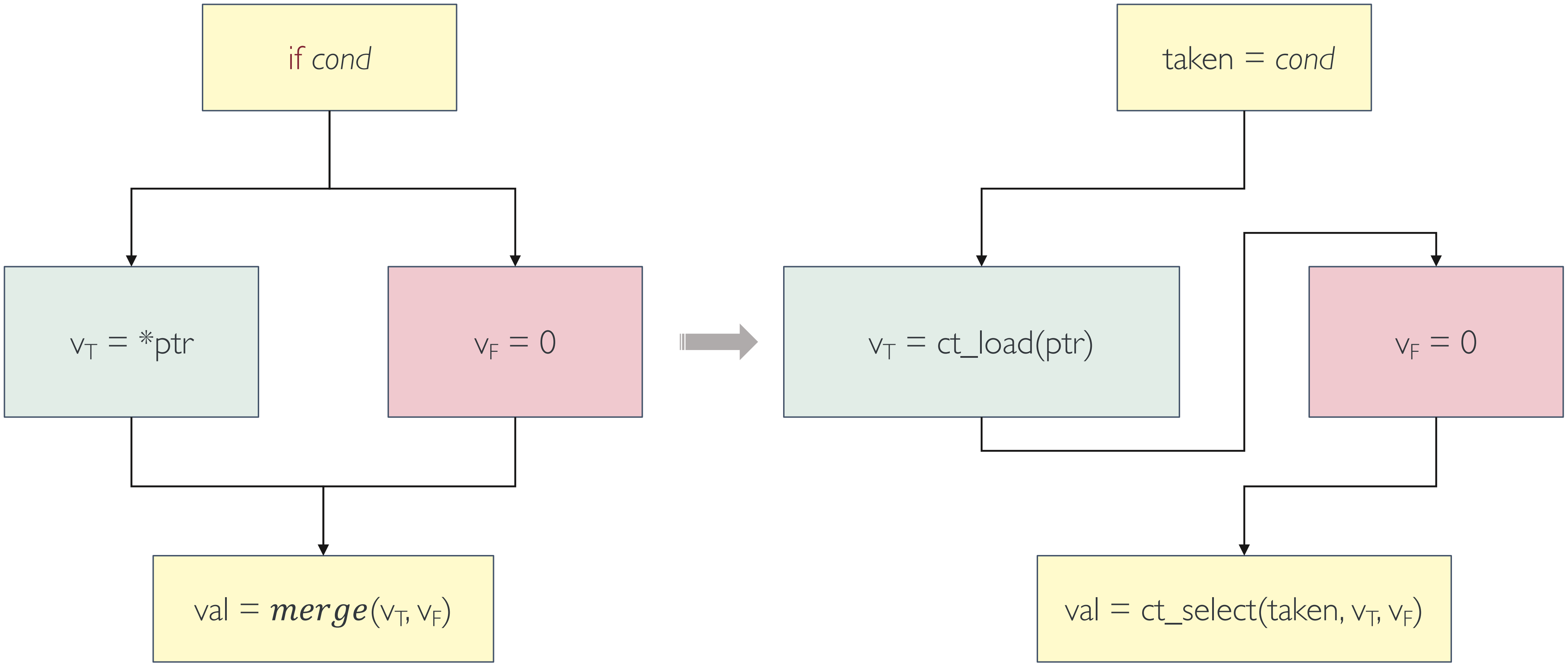
Branch Linearization

if $cond$:
    $val = *\,ptr$
else:
    $val = 0$

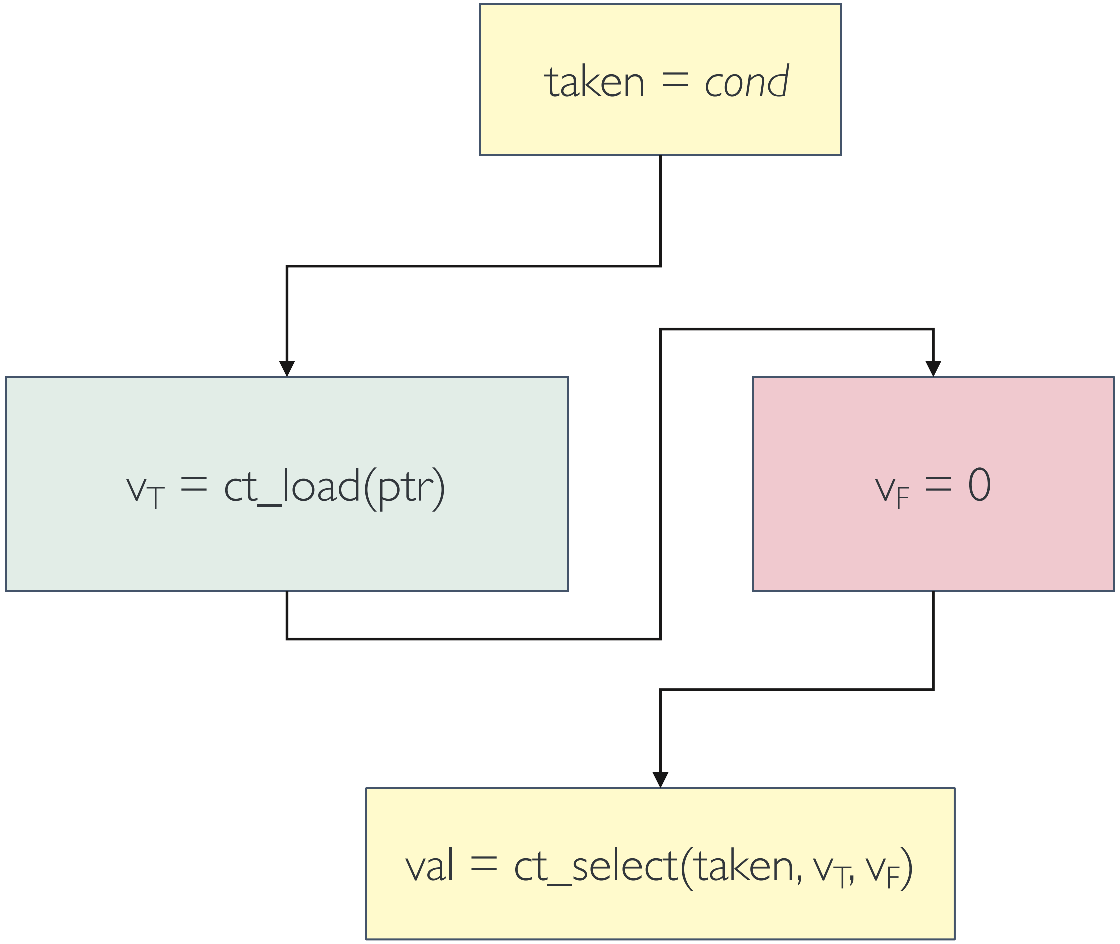
Branch Linearization



Branch Linearization



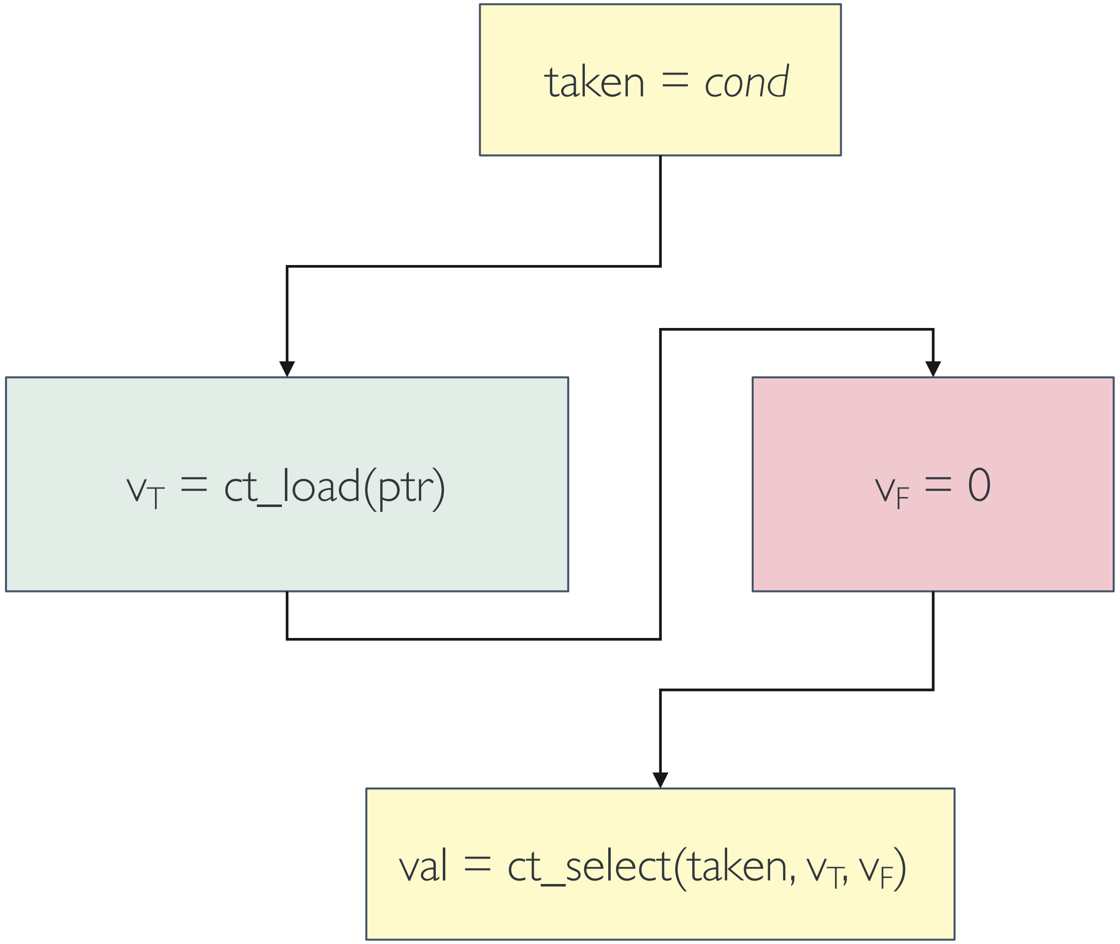
Branch Linearization



ct_select: constant time selection primitive

- cmov [80]
- LLVM select
- multiplication
- bit operations [80]

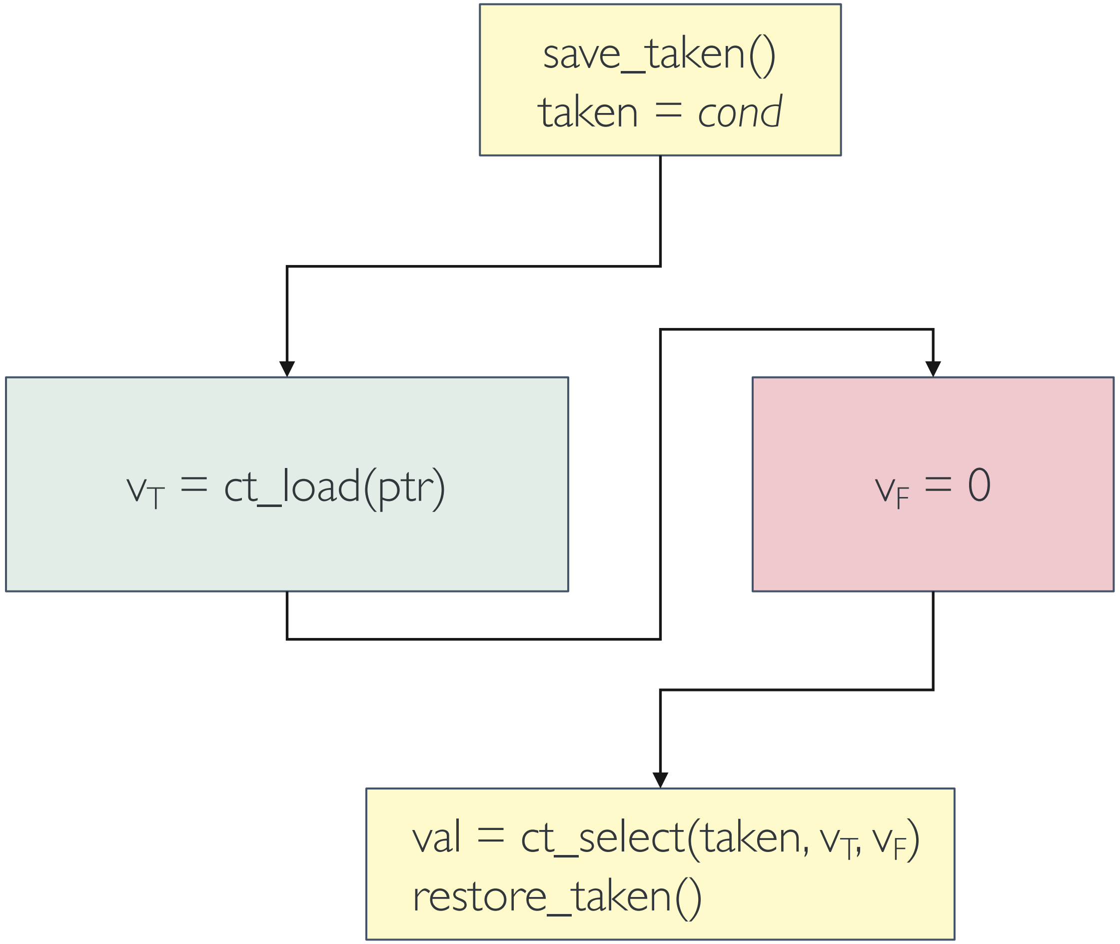
Branch Linearization



ct_select: constant time selection primitive

- cmov [80]
- **LLVM select**
- **multiplication**
- bit operations [80]

Branch Linearization



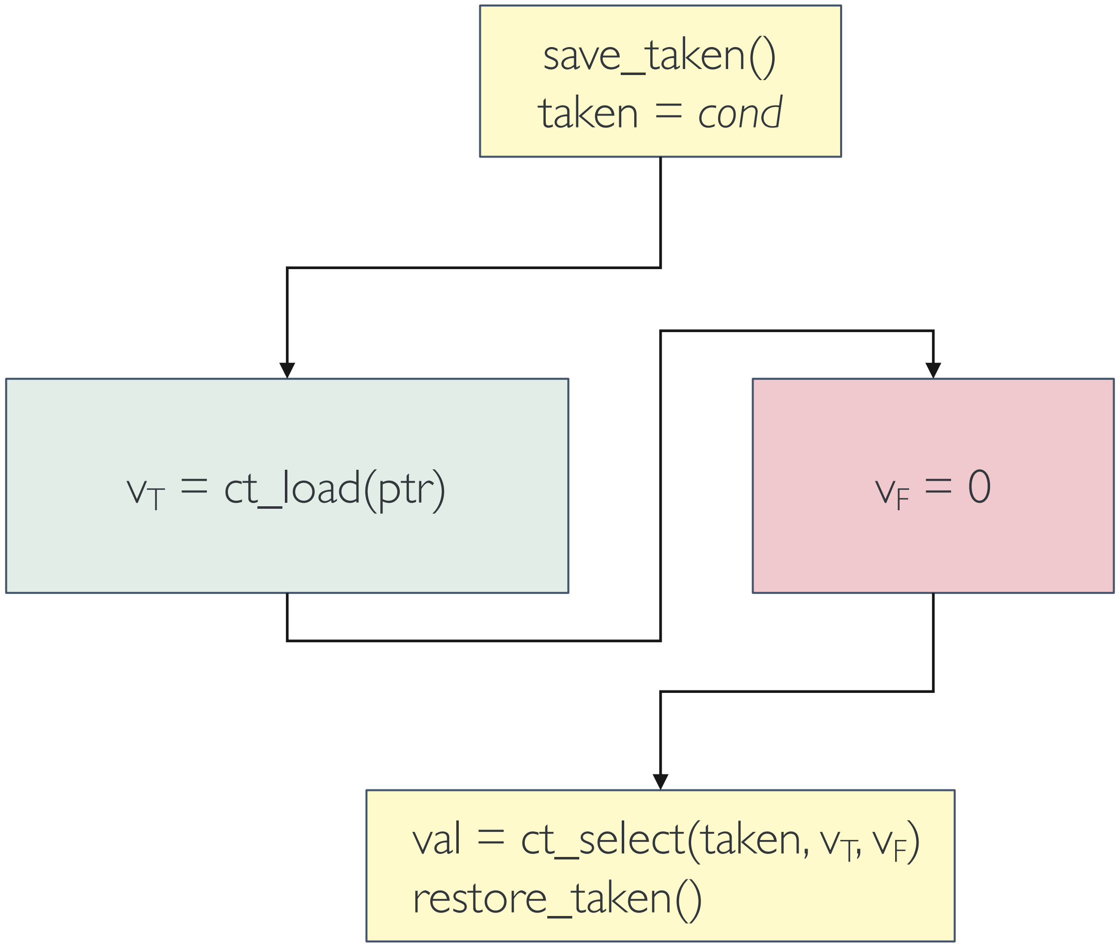
ct_select: constant time selection primitive

- cmov [80]
- **LLVM select**
- **multiplication**
- bit operations [80]

Branch Linearization

Using Branch Linearization:

- branches never depend on secret conditions
- allow **decoy paths** to perform local computations, without being globally visible
- defer memory access linearization to DFL
- the design poses no restriction on code optimizations



Loop Linearization

What to do with loops?

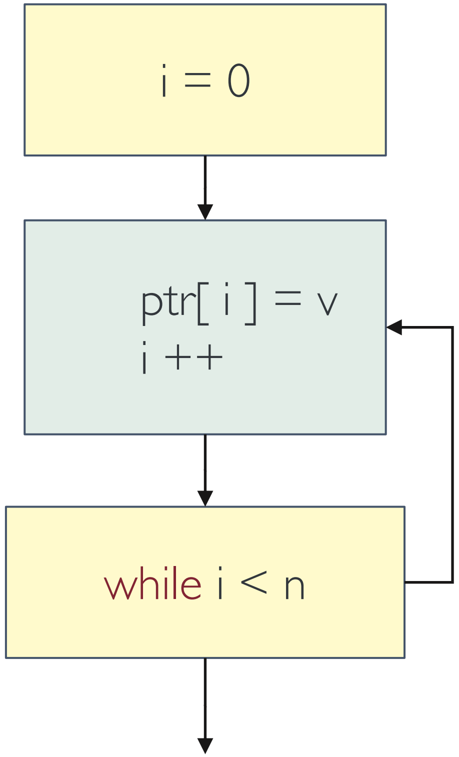
- may not possible to unroll (statically unknown number of iterations)
- may be too costly to unroll (number of iterations too big)

But the number of times a loop executes may depend on secret values!

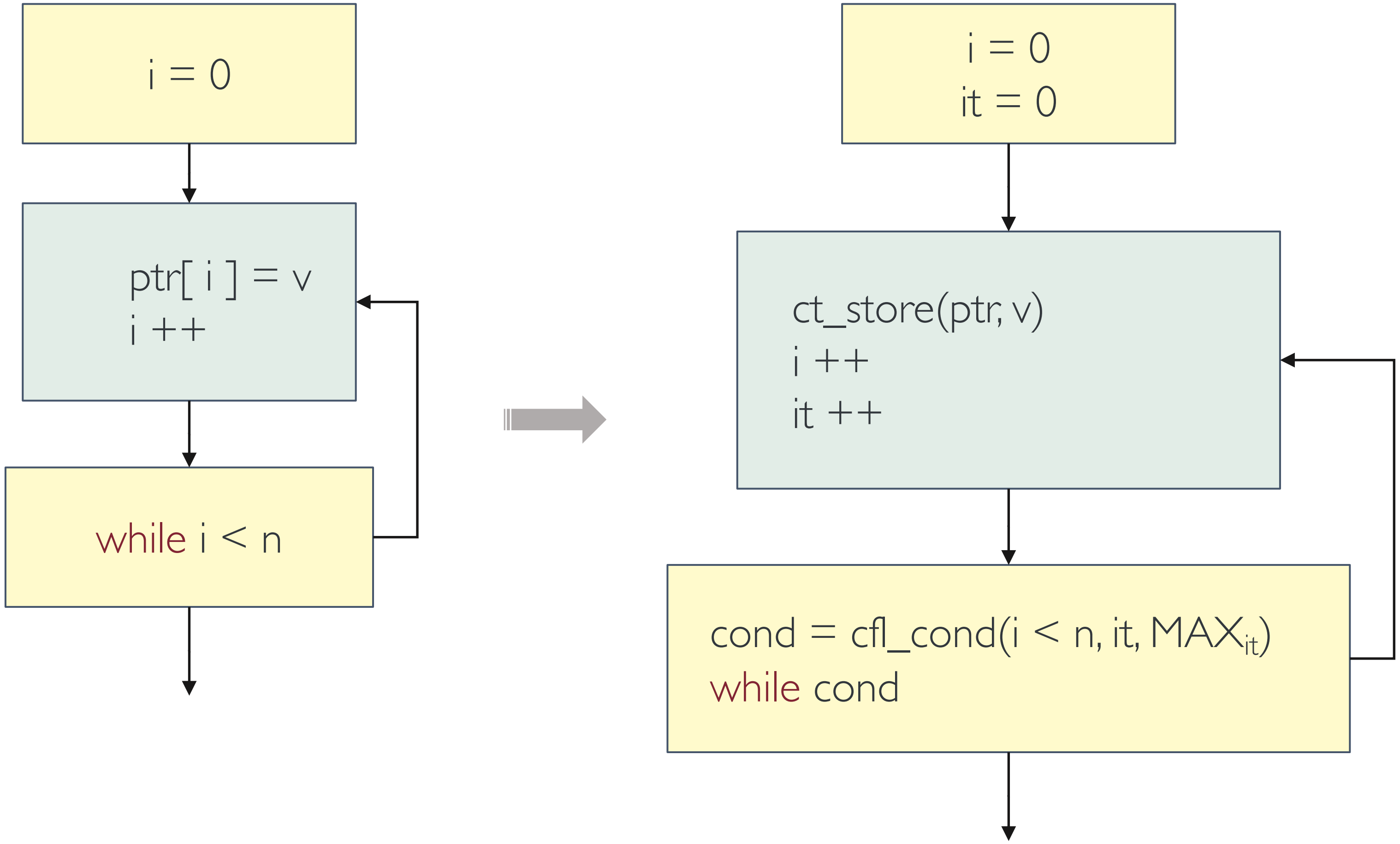
```
while (i > secret):
   do_stuff()
   i ++
```

```
while (i > 0x10000000):
   if secret_condition:
       break
   i ++
```

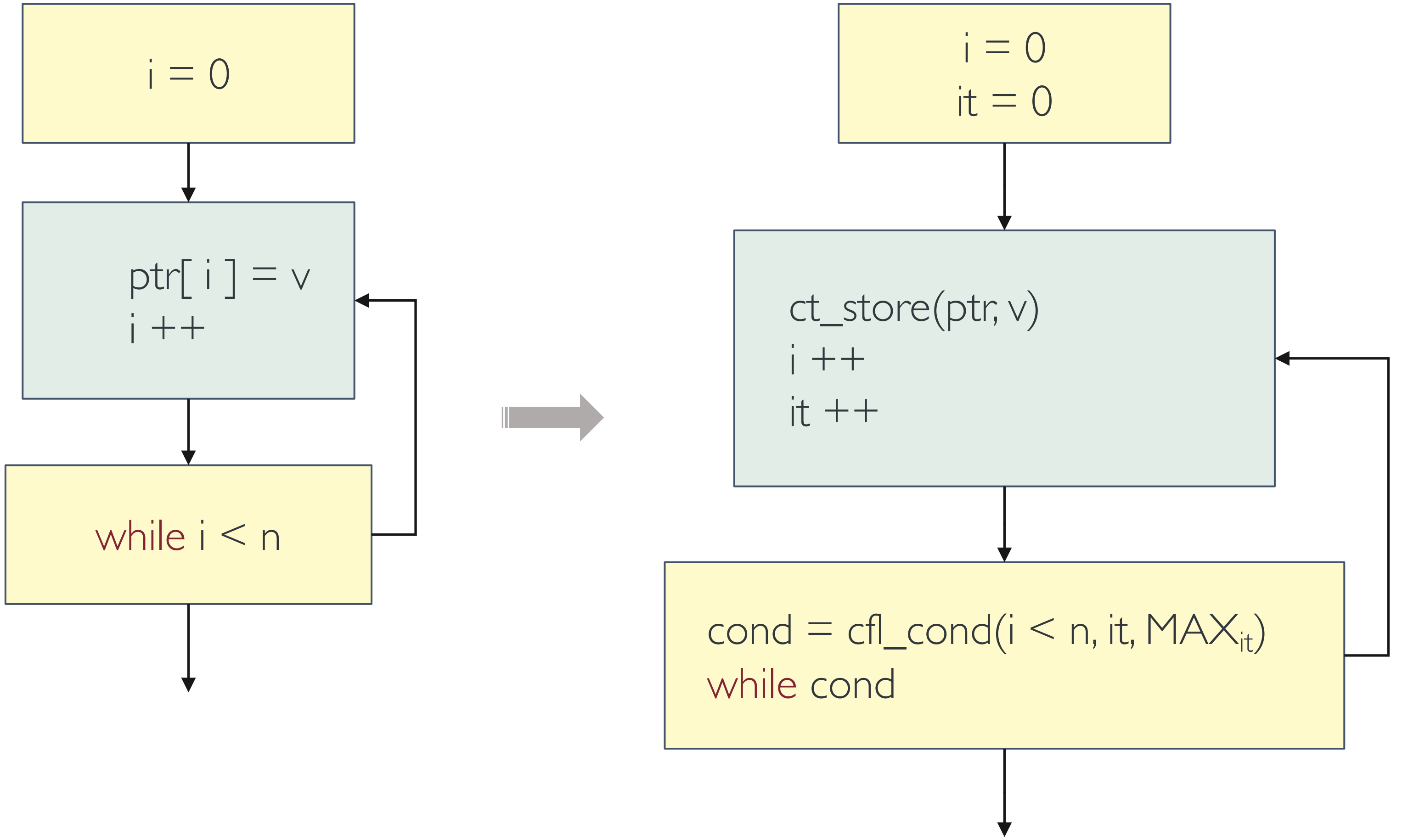
Just in Time Loop Linearization



Just in Time Loop Linearization



Just in Time Loop Linearization



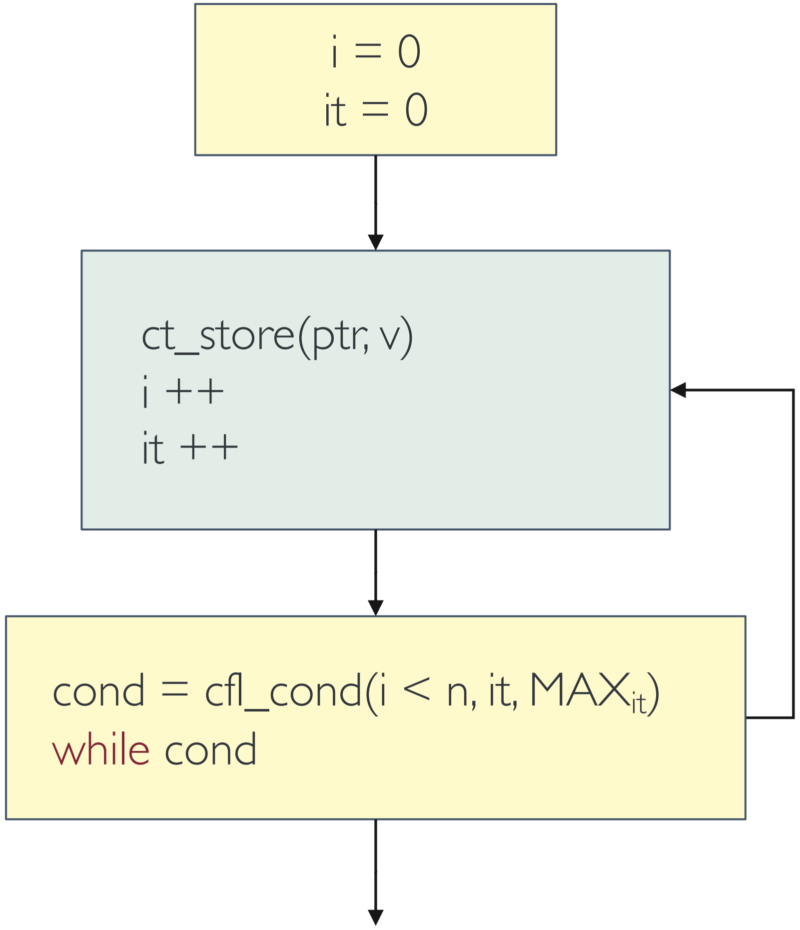
cfl_cond(cond, it, MAX):

Always continue up to MAX$_{it}$, inserting padding iterations and transitioning as a **decoy path** if program execution would exit

Just in Time Loop Linearization

Using JIT Loop Linearization:

- replace the loop trip count with a custom induction variable
- avoid explosion problems
- minimize overhead
- initialize MAX_{it} values during the profiling phase to avoid initialization leaks



Data Flow Linearization

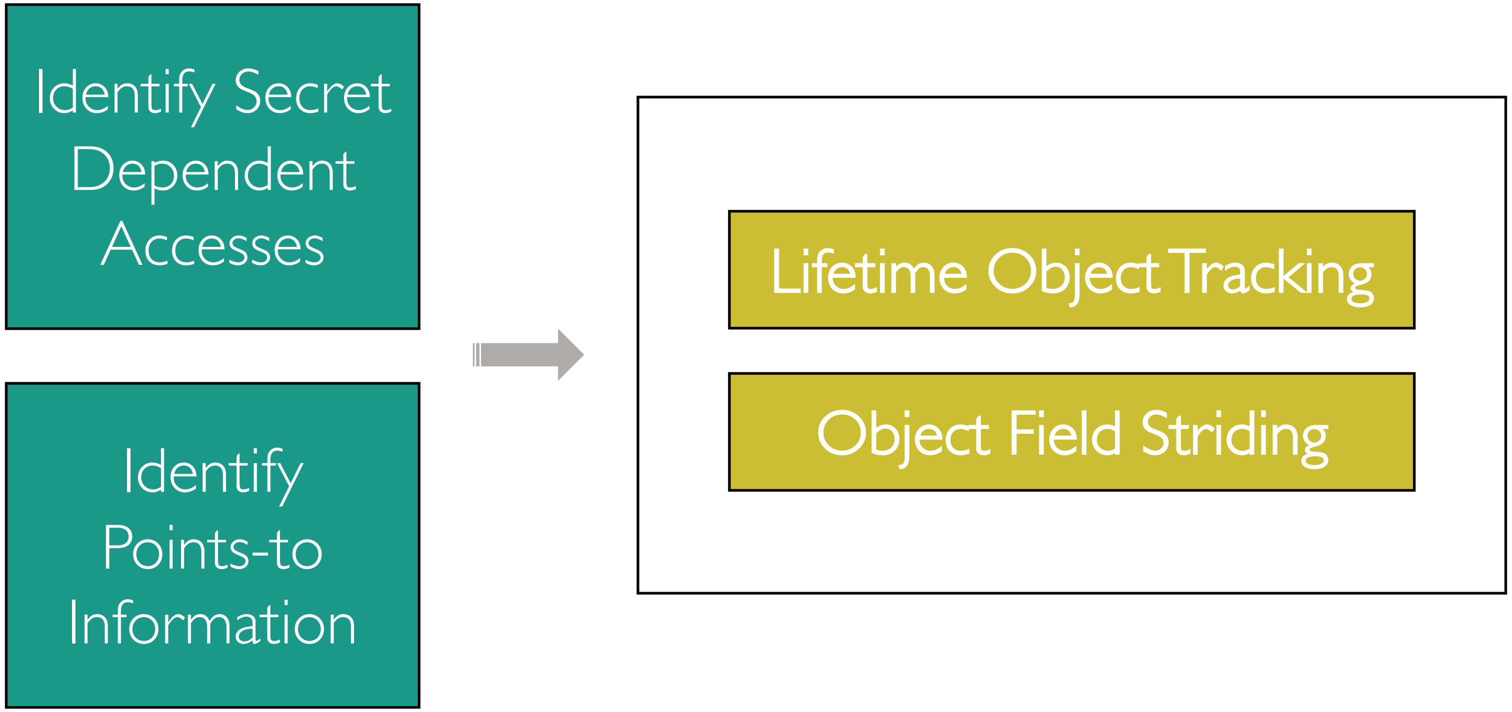
For each secret sensitive memory access, obliviously accesses all the locations that the original program can possibly reference for any initial input

- Do not use shadow locations for decoy paths!
  → Accessing them would reveal the nature of the path

- Do not let dummy values concur into pointer accesses
  → Using them may break some program invariants

Data Flow Linearization

For each secret sensitive memory access, obliviously accesses all the locations that the original program can possibly reference for any initial input



Identify Points-to information

Use pointer analysis (SVF framework) to collect all objects and fields a memory access may touch

Object:

- *global*
- *local*
- *dynamically allocated*

Identify Points-to information

Use pointer analysis (SVF framework) to collect all objects and fields a memory access may touch

Object:

- *global*
- *local*
- *dynamically allocated*

```
access(void* ptr)
    *ptr

func1()
    access(obj1)

func2()
    access(obj2)
```

ptr → {obj1, obj2}

Identify Points-to information

Use pointer analysis (SVF framework) to collect all objects and fields a memory access may touch

Object:

- *global*
- *local*
- *dynamically allocated*

```
access(void* ptr)
    *ptr

func1()
    access(obj1)

func2()
    access(obj2)
```

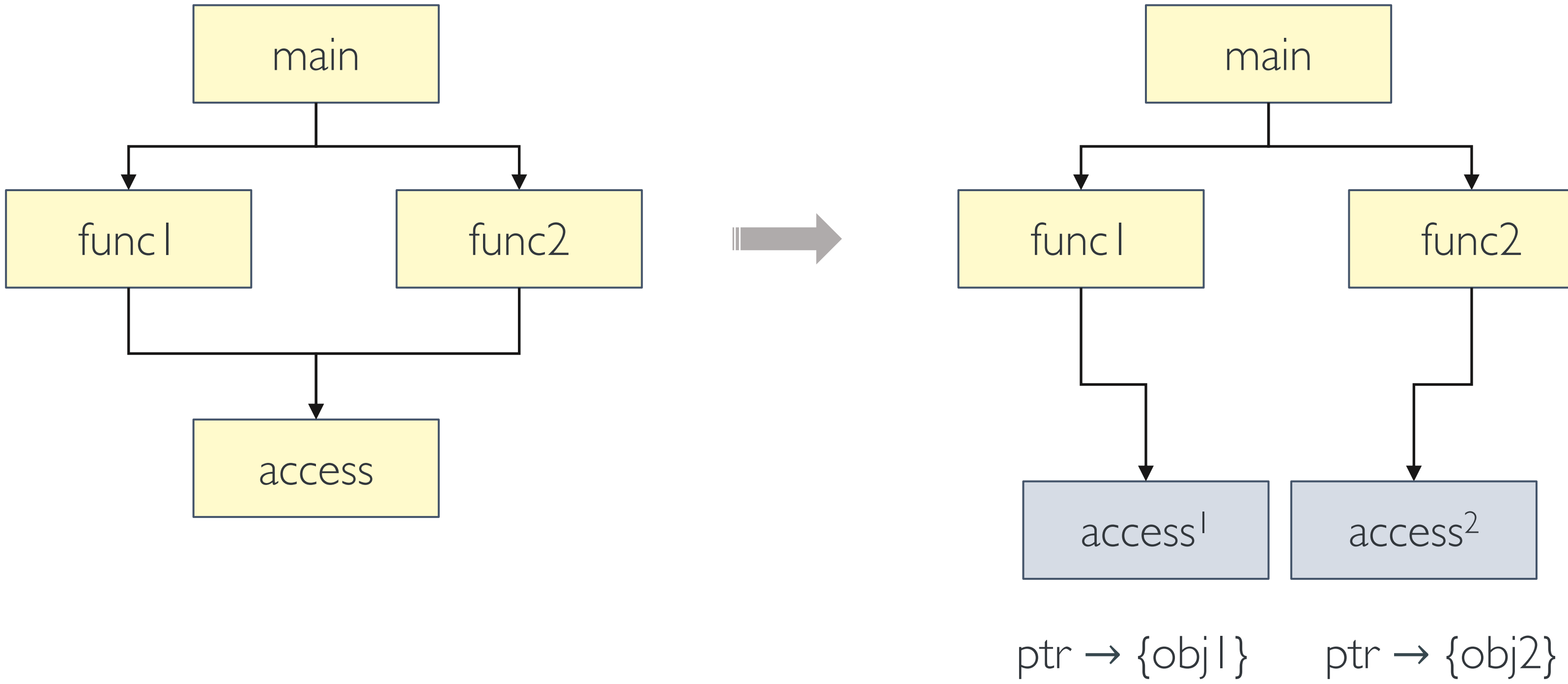
ptr → {obj1, obj2}

no context info!

→ Without context information any memory access will depend on any possible caller, and the points-to sets explode

Aggressive Function Cloning

Add context information by creating a function clone for every different calling context encountered during secret dependent execution

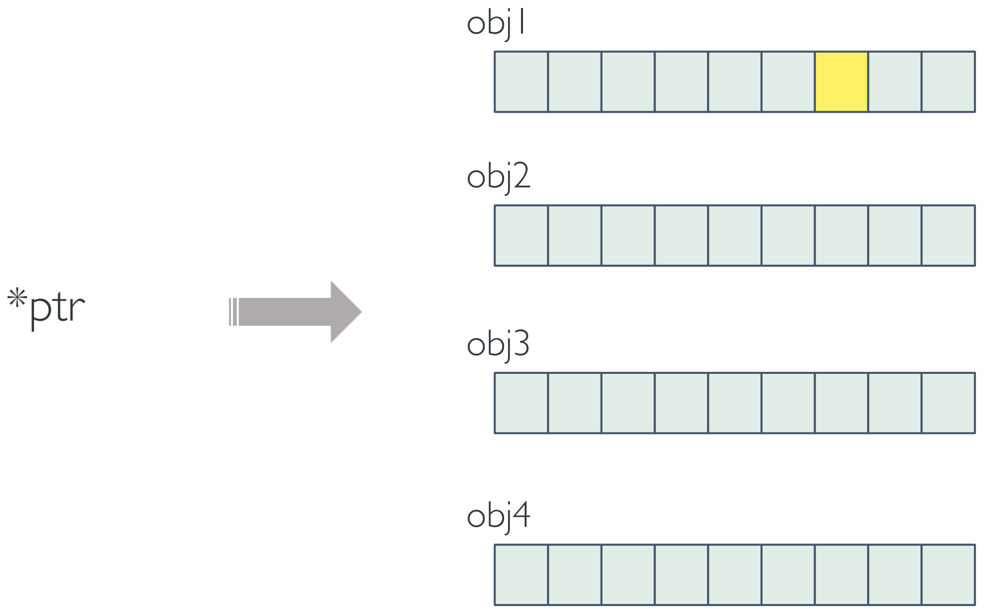


- Context sensitive pointer analysis only where required
- Precise points-to information

Linearizing Memory Accesses

For each secret sensitive memory access, touch all the possible objects the instruction may refer to, but only return/update the correct location

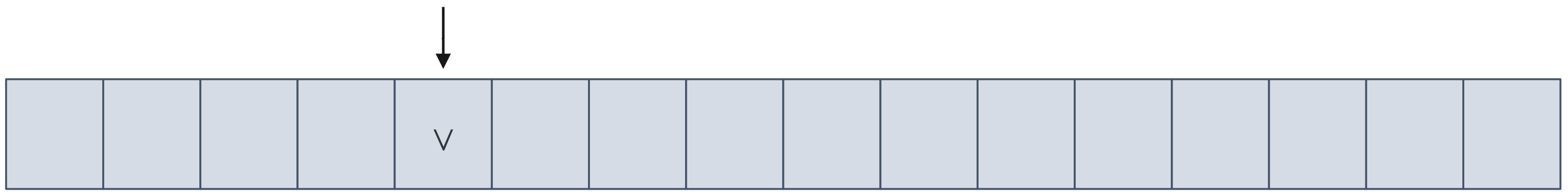
→ protect against active attackers



- need to track dynamic objects
- need a fast way to touch all locations and merge results

Object Field Striding

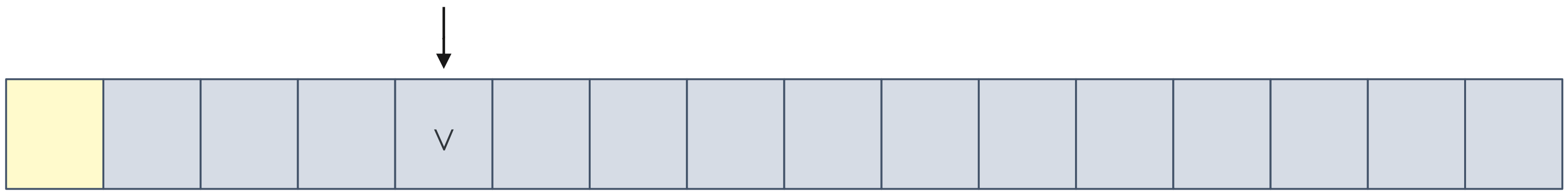
For each secret dependent memory access, stride each portion of each object the memory may touch, at λ granularity (e.g. cache line granularity)



ret = $\perp$

Object Field Striding

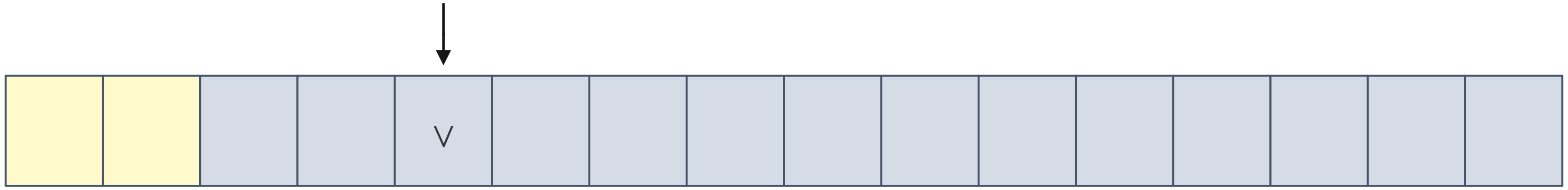
For each secret dependent memory access, stride each portion of each object the memory may touch, at λ granularity (e.g. cache line granularity)



ret = $\perp$

Object Field Striding

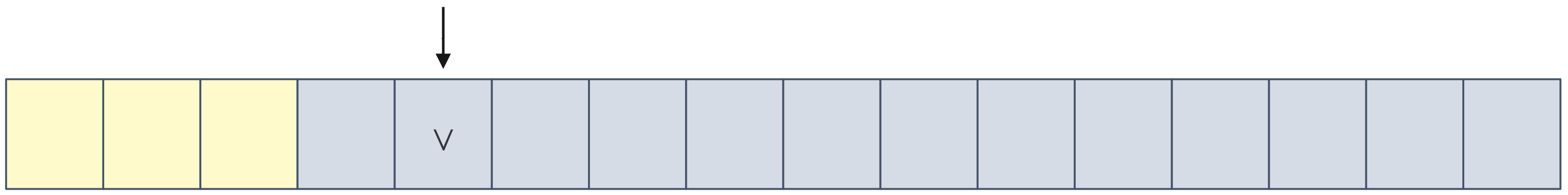
For each secret dependent memory access, stride each portion of each object the memory may touch, at λ granularity (e.g. cache line granularity)



ret = $\perp$

Object Field Striding

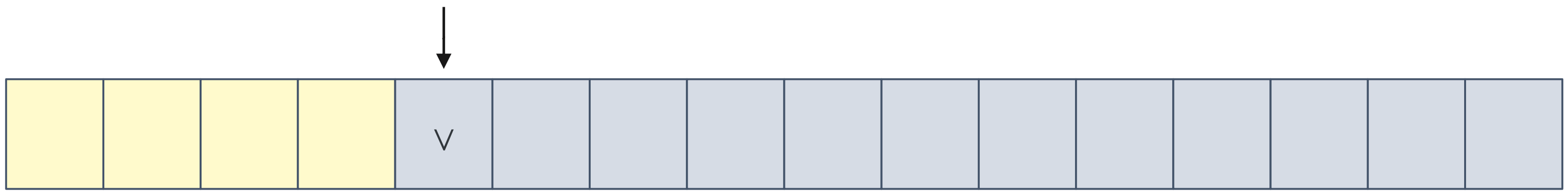
For each secret dependent memory access, stride each portion of each object the memory may touch, at λ granularity (e.g. cache line granularity)



ret = $\perp$

Object Field Striding

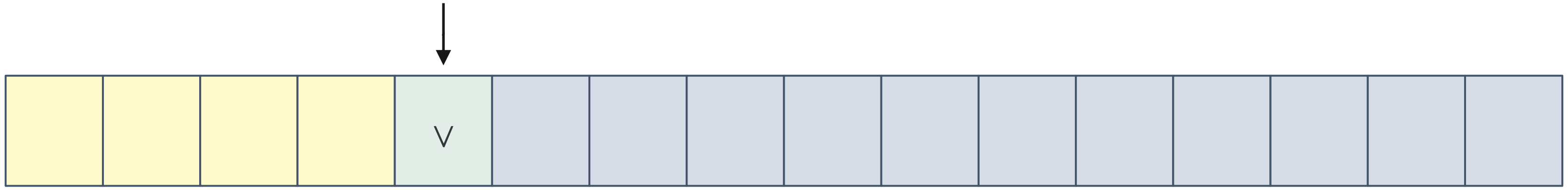
For each secret dependent memory access, stride each portion of each object the memory may touch, at λ granularity (e.g. cache line granularity)



ret = $\perp$

Object Field Striding

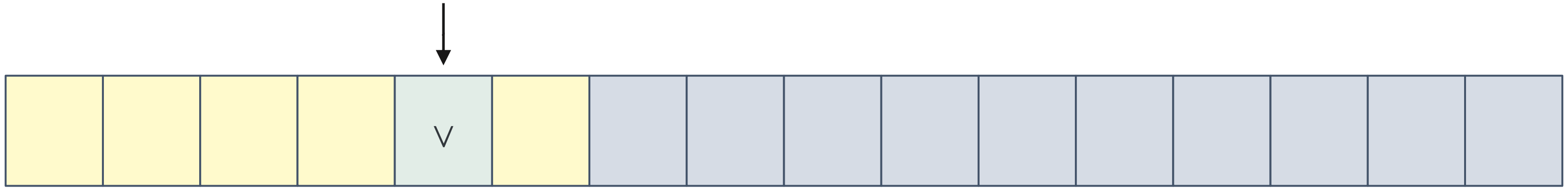
For each secret dependent memory access, stride each portion of each object the memory may touch, at λ granularity (e.g. cache line granularity)



ret = v

Object Field Striding

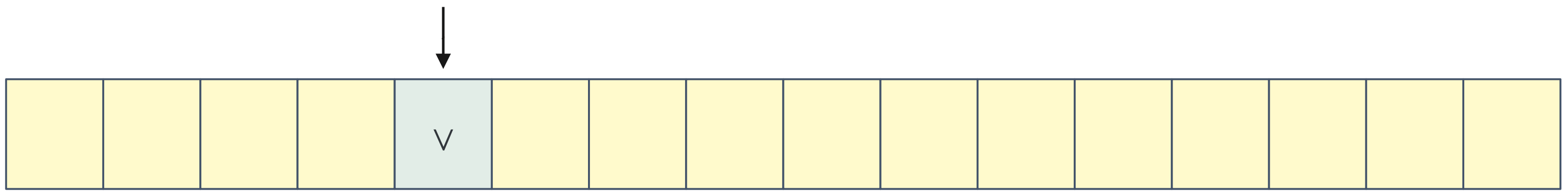
For each secret dependent memory access, stride each portion of each object the memory may touch, at λ granularity (e.g. cache line granularity)



ret = v

Object Field Striding

For each secret dependent memory access, stride each portion of each object the memory may touch, at λ granularity (e.g. cache line granularity)

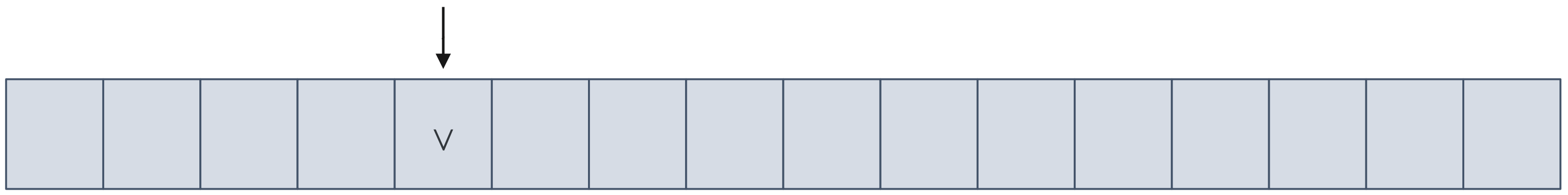


ret = v

Object Field Striding - AVX

For each secret dependent memory access, stride each portion of each object the memory may touch, at λ granularity (e.g. cache line granularity)

Use AVX scatter & gather on bigger objects to load multiple $\lambda - \mathbf{sized}$ regions

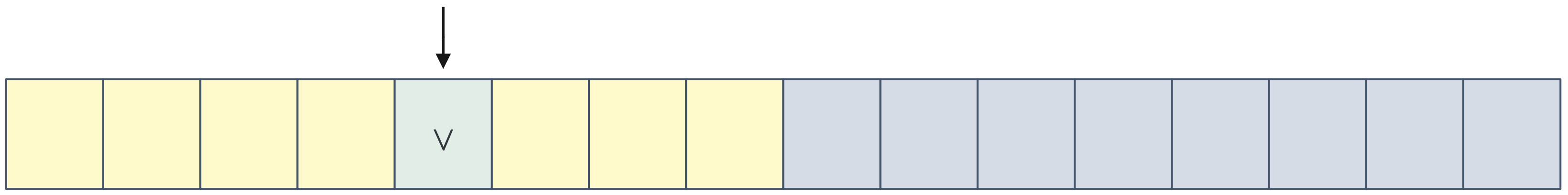


ret = $\perp$

Object Field Striding - AVX

For each secret dependent memory access, stride each portion of each object the memory may touch, at λ granularity (e.g. cache line granularity)

Use AVX scatter & gather on bigger objects to load multiple $\lambda - \mathbf{sized}$ regions

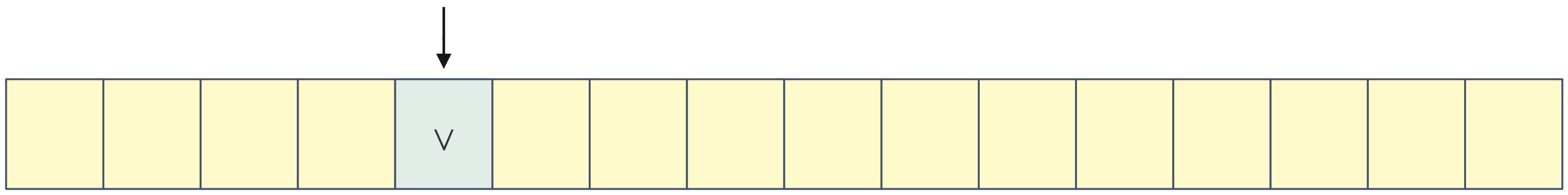


ret = v

Object Field Striding - AVX

For each secret dependent memory access, stride each portion of each object the memory may touch, at λ granularity (e.g. cache line granularity)

Use AVX scatter & gather on bigger objects to load multiple $\lambda - \mathbf{sized}$ regions

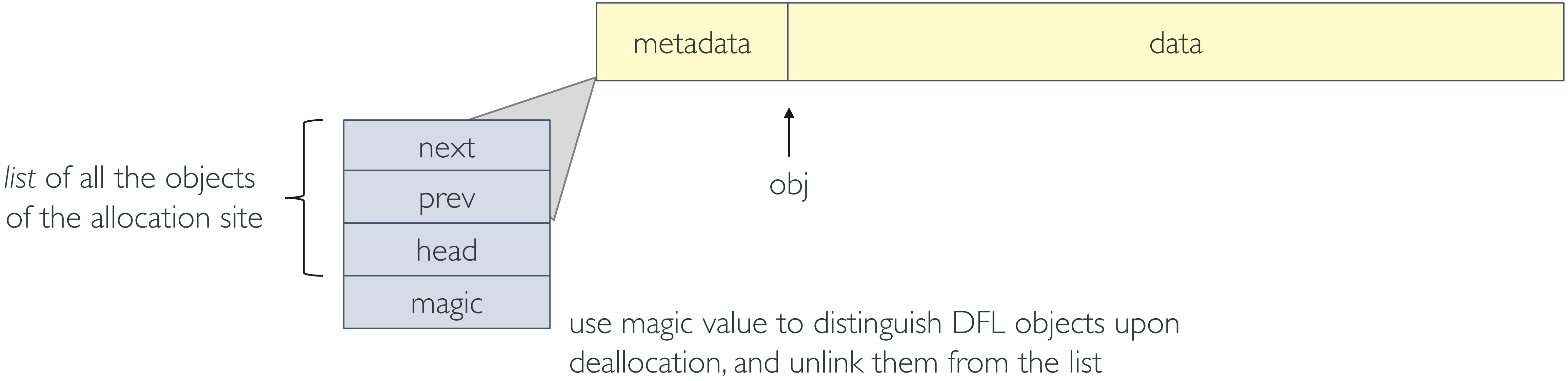


ret = v

Runtime Object Tracking

Need to track lifetime for dynamically allocated + stack objects that concur in sensitive accesses, to protect them

→ Enlarge objects allocations to insert *in-band* metadata



Runtime Object Tracking

Need to track lifetime for dynamically allocated + stack objects that concur in sensitive accesses , to protect them

→ Enlarge objects allocations to insert *in-band* metadata

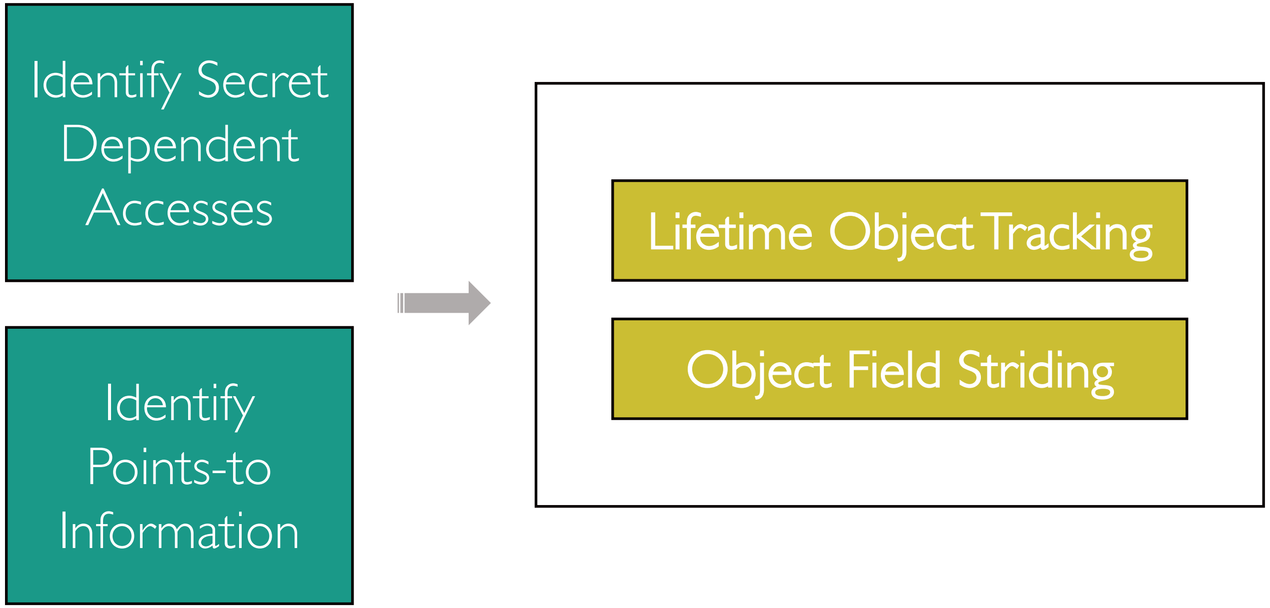
Stack promotion: Global values do not need runtime tracking

→ Promote local variables into globals for every *non-recursive* sensitive function

Data Flow Linearization

Using Data Flow Linearization:

- remove leaks from secret dependant memory accesses
- touch a minimal set of possible object thanks to **function cloning** and **SVF optimizations**
- stride over fields using **AVX** operations
- ensure security and memory safety by tracking dynamic objects



Runtime Overhead

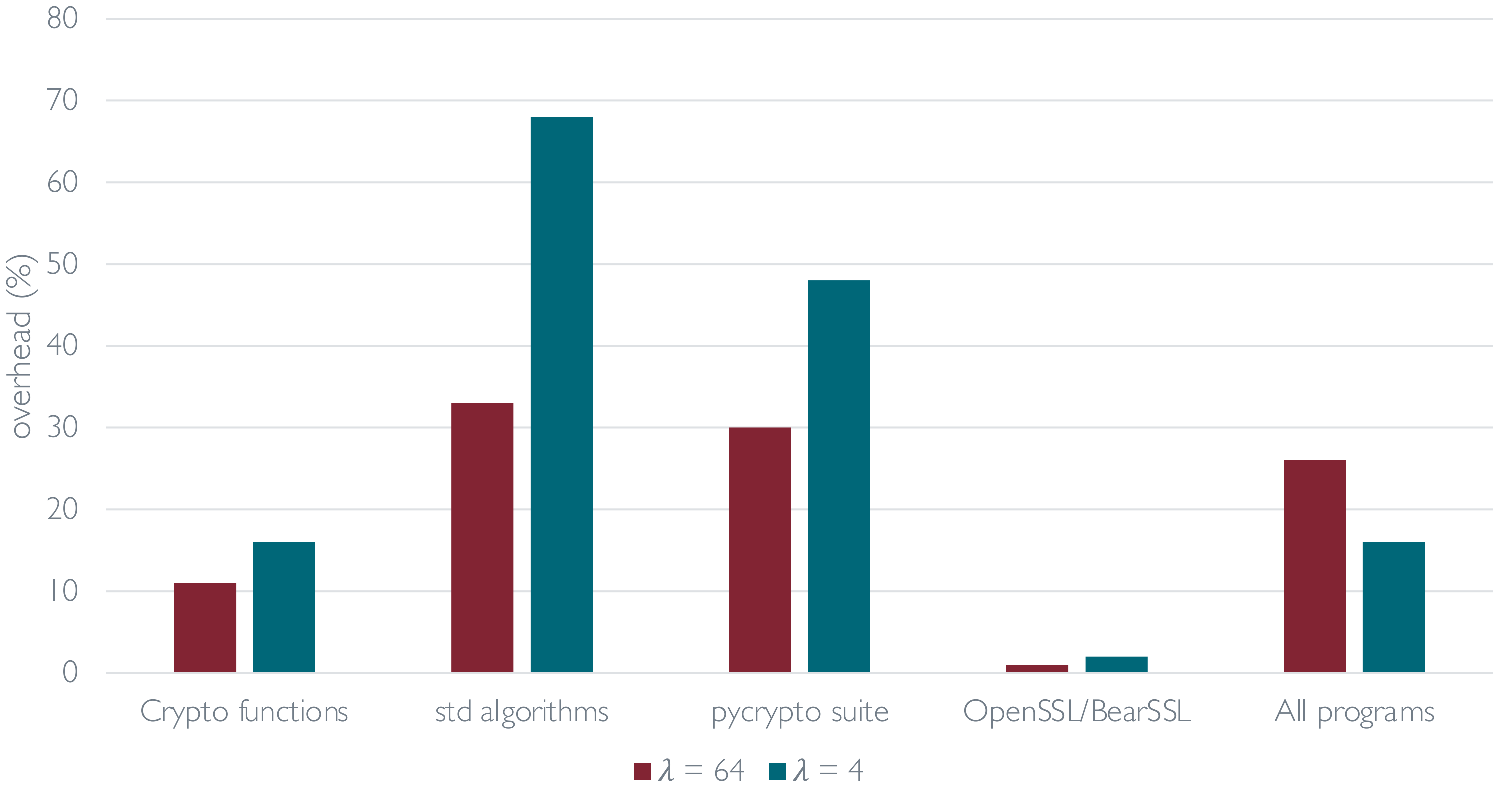
Protect cryptographic implementations against cache-level attacks (λ =64) and core-colocation attacks (λ =4)

- 23 cryptographic modules extracted from a 19-KLOC codebase
- 6 microbenchmarks of standard algorithms (matmul, sorting, …)
- 5 modules of the *pycrypto* suite
- 3 leaky functions from OpenSSL and BearSSL

check using:
- hw perf counters
- GEM5 simulator
- cachegrind
- PIN-tool

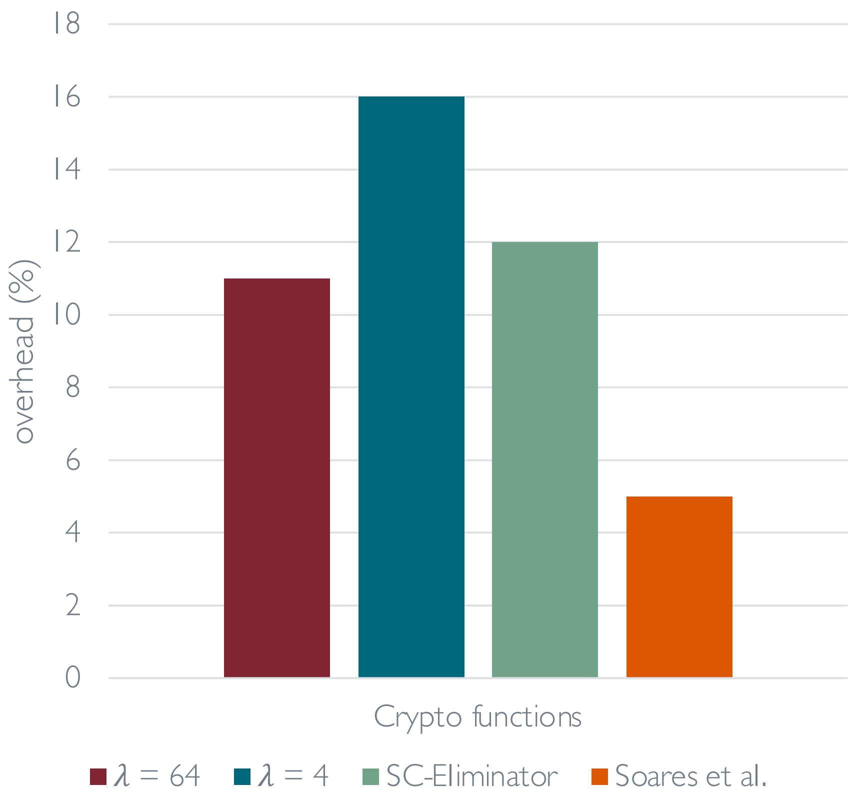
Runtime Overhead

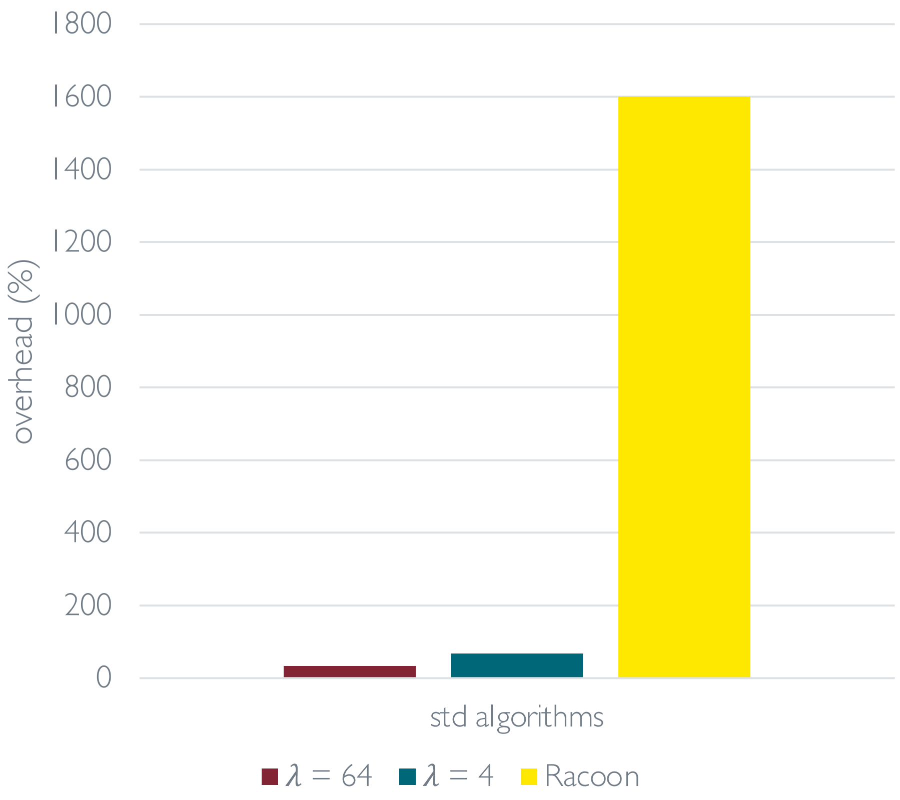
Protect cryptographic implementations against cache-level attacks (λ =64) and core-colocation attacks (λ =4)



Runtime Overhead

Protect cryptographic implementations against cache-level attacks (λ =64) and core-colocation attacks (λ =4)





Case Study

Linearize ECDSA modular multiplication of the non-CT WolfSSL implementation to compare with the hand-written CT implementation

- 84 functions
- 6.28 possible object for each memory access on average

Case Study

Linearize ECDSA modular multiplication of the non-CT WolfSSL implementation to compare with the hand-written CT implementation

→ aftern cloning

- ~~84~~ 864 functions
- ~~6.28~~ 1.08 possible object for each memory access on average

Case Study

Linearize ECDSA modular multiplication of the non-CT WolfSSL implementation to compare with the hand-written CT implementation

→ aftern cloning

- ~~84~~ 864 functions
- ~~6.28~~ 1.08 possible object for each memory access on average

11.4x overhead w.r.t. the hand-written, carefully designed, CT ECDSA implementation

→ Constantine can handle a real-world crypto library component

Conclusions

automatically transform programs into their constant-time equivalents

introduce **radical abstractions** around value computations

low overhead and **strong security guarantees**

in the future: Constantine for **speculative** constant-time guarantees

Constantine: https://github.com/pietroborrello/constantine

Questions?

thank you

contact: borrello@diag.uniroma1.it

References

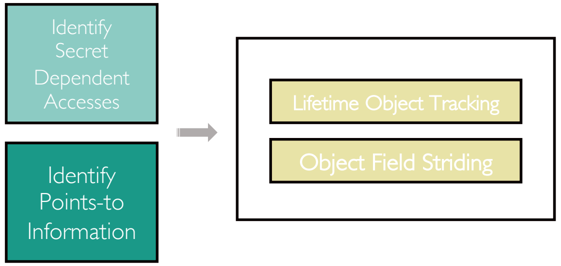
[20] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern X86 Processors. S&P 2009

[45] J. Agat. Transforming out Timing Leaks. POPL 2000

[53] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. USENIX SECURITY 2015

[62] L. Soares and F. Magno Quintao Pereira. Memory-Safe Elimination of Side Channels. CGO 2021

[80] M. Wu, Shengjian Guo, P. Schaumont, and C. Wang. Eliminating Timing Side-Channel Leaks Using Program Repair. ISSTA 2018



Refined Field Sensitivity

Distinguish the portions of an object that can be accessed

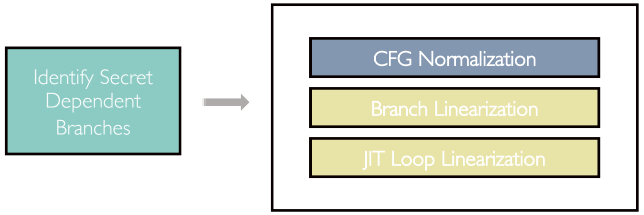
- refine SVF field sensitivity by delaying field insensitive promotion
- apply heuristics based on duck typing when SVF fails

```
short* ptr → objS.buf2

long*   ptr → objS.id
```

```
struct S {
    long  id;
    char  buf1[256];
    short buf2[256];
};
```

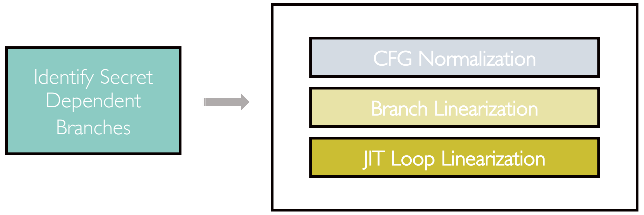
CFG Normalization

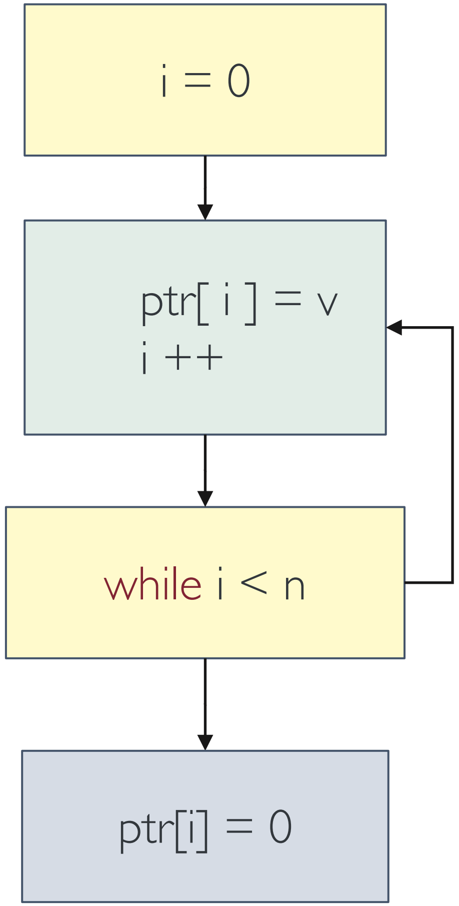


Bring the intermediate representation of a program into normal form

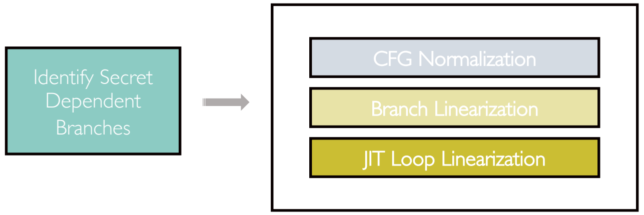
- SSA
- Single-entry, single-exit regions
- Lower switch constructs into *if-else* sequences
- Promote indirect calls into direct ones
- Normalize Loops

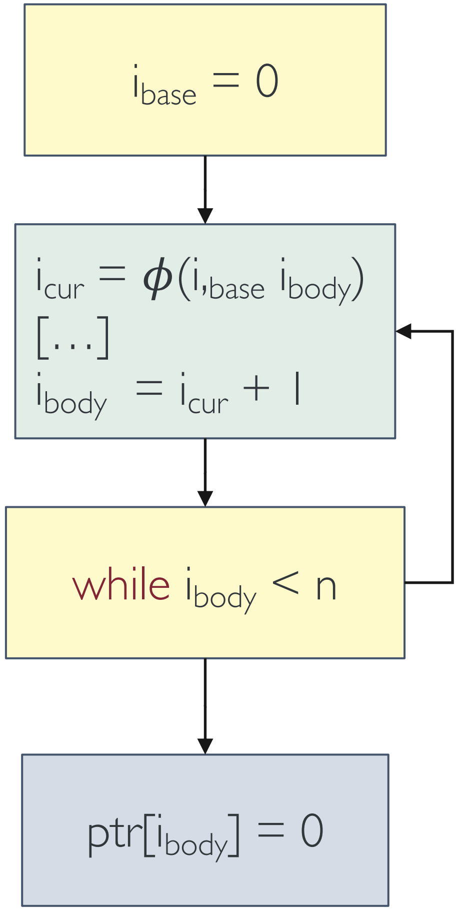
Just in Time Loop Linearization-Escaping variables





Just in Time Loop Linearization-Escaping variables





Just in Time Loop Linearization-Escaping variables