Collapsing Towers for Side-Channel Security (Short Paper)

Cameron Wong
cwong@g.harvard.edu
Harvard University
USA

Muhammad Abdullah
abd880@mit.edu
MIT
USA

Yuheng Yang
yuhengy@mit.edu
MIT
USA

Mengjia Yan
mengjiay@mit.edu
MIT
USA

Adam Chlipala
adamc@csail.mit.edu
MIT
USA

Nada Amin
namin@seas.harvard.edu
Harvard University
USA

Abstract

We propose a new technique for ahead-of-time detection of side channel information leaks through the use of staged, instrumented interpreters. It is well-known that a staged interpreter acts as a compiler. By equipping such an interpreter with a runtime step counter, the timing behavior of the target program becomes a first-class value in the residue, to be analyzed by any number of classical methods. Finally, we demonstrate the power of our technique against naive, cache-based and speculation-based timing attacks using an off-the-shelf bounded model checker.

Keywords: staging, interpreters, side-channel security, Scala

ACM Reference Format:
Cameron Wong, Muhammad Abdullah, Yuheng Yang, Mengjia Yan, Adam Chlipala, and Nada Amin. 2025. Collapsing Towers for Side-Channel Security (Short Paper). In . ACM, New York, NY, USA, 6 pages.

1 Introduction

Side channel attacks are a persistent thorn in the side of the computer security community, with new methods of leaking information through increasingly esoteric attack vectors published every year. These attacks exploit unintended information channels such as power consumption, electromagnetic radiation, acoustic emanations and so on. Side channel leakages are especially dangerous because they ignore traditional software security measures and leave little trace when utilized.

Given the vast diversity of environments in which programs are executed, accounting for all potential side channels through traditional testing or mitigation techniques quickly becomes infeasible. Formal methods offer a way out by ruling out entire classes of vulnerabilities in a systematic and verifiable way. Unfortunately, manually developing formal security models for modern systems is challenging and error-prone, as contemporary program stacks are designed to abstract away the very low-level details that give rise to side channels in the first place.

Recent work in hardware-aware specification has introduced *software-hardware contracts* [11] as a way to define the responsibilities of each side of the software-hardware divide. Specifically, a software-hardware contract asserts that, so long as the software obeys a given set of constraints, the hardware mechanism in question will not induce any further information leakage. However, hardware mechanisms tend to be complex and detail-oriented, requiring many iterations of candidate contracts before the correct contract is discovered. Without a lightweight means of verifying a given design, it may become too costly to check every combination.

In this paper, we propose the use of *Collapsing Towers of Interpreters* [1] (henceforth just "collapsing towers") as a means to cheaply verify whether a given software-hardware design satisfies a given security property. Collapsing towers is a technique utilizing the Futamura projections [8, 9] to compile a program running atop a stack of interpreters into a single program exhibiting the semantics of the entire system. For our purposes, the interpreter stack in question is a surface program expressed via a processor's ISA (Instruction Set Architecture), which is in turn running on some complex hardware pipeline. The collapsed output (the *residue*) is a C program with all microarchitectural effects surfaced, which can then be analyzed with any off-the-shelf C analysis. Importantly, any verification annotations can be added through the compilation process – the developer does not need to adjust the original source at all, beyond isolating the critical section.

Our contributions are as follows:

- We describe a novel methodology for side-channel analysis via collapsing towers to combine high-level evaluation semantics with low-level timing minutiae to construct timing-aware models.
- Starting from a simple assembly language, we show how a naive interpreter can be staged and instrumented such that the timing behavior of the surface program is reflected as first-order C code to be analyzed by an off-the-shelf bounded model checker.
- We demonstrate how to extend the base interpreter such that the same analysis can be performed against

classical, cache-based and speculation-based timing attacks.

The full version of our implementation can be found in our repository.

2 Invisible Semantics

Ahead-of-time detection of timing leaks is particularly tricky because the minutiae of modern hardware is deliberately kept hidden from the software running on it. Rather than manipulate the hardware's capabilities directly, programmers interact with hardware through an abstracted ISA that presents the illusion of linear instruction execution. In other words, the timing behavior of the overall software-hardware system is defined by both the *visible* semantics of the ISA and the *invisible* semantics of the underlying hardware state.

In the rest of this section, we review relevant background of microarchitecture-induced timing side channels and discuss how they arise from gaps between visible and invisible semantics. We then give an overview of staging and outline our approach to using interpreter specialization as a means to bring invisible semantics to the forefront.

2.1 Side channels arise from invisible semantics

As previously mentioned, today's processors only present the illusion of well-ordered instruction execution. Modern processors will hide latency induced by busy or slow execution units by looking ahead of the logical program counter for instructions that don't depend on operations currently in-flight so that multiple instructions can be executed concurrently.

Of particular importance for timing-based side channels is *branch prediction*, in which a processor will guess the direction of a conditional branch to continue lookahead processing without needing to wait for the true result. If the prediction turns out to be wrong, any in-flight speculative instructions need to be flushed (instead of being committed to the ISA state), causing measurable slowdown.

Consider the program in Fig. 1, adapted from Cauligi et al. [4]. According to the software (visible) semantics, this program is certainly secure – the private memory is not accessed at all! On a machine with speculative execution and caching, however, the branch may be speculatively taken even though the guard `x1 < 8` is false, causing a speculative access to the out-of-bounds cell `pub[9]`. If the program memory layout happens to be such that the address of `pub[9]` falls within the `priv` array, the phantom read will be reading a private value! The hardware cache will then obligingly store the private phantom value, allowing it to be exfiltrated by a clever attacker.

2.2 Seeing the invisible via interpreter specialization

How might we have caught this? The private memory access occurs through not just an out-of-bounds read, but a *phantom* out-of-bounds read, rendering it undetectable by many

```
1 int pub[8];
2 int priv[8];
3 int x1 = 9;
4 if (x1 < 8) {
5   int x2 = pub[x1];
6   int x3 = pub[x2];
7 }
```

Figure 1. A sample SPECTRE vulnerability.

constant-time disciplines such as FaCT [5]. One approach would be to design an analysis to be aware of the hardware (invisible) semantics directly, by hand-constructing some abstract machine with an explicit model of each invisible feature as desired [4, 10].

We view the problem differently. Instead of designing new analyses for each invisible feature, we would like to make invisible semantics visible to existing analyses by transforming the input program to include the desired hardware details.

Consider a typical interpreter taking a program and an environment. If the interpreter is then specialized to a particular input program, it becomes a compiler. This is known as the first Futamura projection [8, 9]. Amin and Rompf [1] observed that, if an interpreter is instrumented with additional runtime metadata (such as, say, a timer and a simulated memory cache), that instrumentation will be reflected into the residue as flat, first-order code, where it can be independently analyzed with classical techniques [2].

This suggests another path forward: Write an interpreter according to the invisible hardware semantics, then stage it, forming a "hardware semantic compiler". Running this compiler on an input program will collapse the software-hardware tower, producing a *residual* program (the *residue*) describing the exact behavior of the hardware when running the program in question. Now, the software-hardware contract can be expressed purely in terms of visible objects in the residue, so we can enforce it with any appropriate off-the-shelf tool.

An additional benefit of using a collapsing-towers approach is that interpreters are generally easier to write than compilers or static analyzers, so *composing* interpreter features should also be easier than designing the corresponding combined analysis from scratch. Indeed, in the next section, we show how caching and speculation can be added to the staged interpreter without needing to explicitly specify how the two features interact.

The rest of this paper describes how we build the staged hardware interpreter to be easily extensible and such that the residue is amenable to analysis.

3 Proof of Concept: Collapsing the software-hardware tower

For our demonstrations, we use a subset of RISC-V as our surface language (which we dub nanoRISC) featuring only arithmetic, memory and branching.

The Scala definitions for nanoRISC (which will also serve to define the visible semantics) can be found in Fig. 2, where `get_reg`, (etc) perform the usual lookups against the members of `StateT`. These are left abstract (rather than accessing `s.regs` directly) so they can be overridden by interpreter variants that wish to implement a more involved semantics.

```
case object Eq extends Cmp // ...
case object Add extends Op // ...
case class Reg(i: Int) extends Operand
case class Immediate(i: Int) extends Operand
abstract sealed class Instr
case class Mov(dst:Reg, src:Operand)
  extends Instr
case class Binop(op:Op,dst:Reg,s1:Reg,s2:Operand)
  extends Instr
case class Load(dst:Reg, src:Reg, offs:Operand)
  extends Instr
case class Store(dst:Reg, src:Reg, offs:Operand)
  extends Instr
case class B(cmp:Option[(Cmp,Reg,Operand)],t:Addr)
  extends Instr
val prog: Vector[Instr]
def get_reg(s: StateT, r: Reg): Int = // ...
def get_mem(s: StateT, i: Int): Int = // ...
def eval_op(op:Op,i1:Int,i2:Int): Int = // ...
def execute(i: Int, s: StateT) =
  prog(i) match {
  case Mov(dst, src) => {
    set_reg(s, dst, src)
    execute(i+1, s) }
  case Binop(op, dst, src1, src2) => {
    set_reg(s, dst, eval_op(s,op,src1,src2))
    execute(i+1, s) }
  case Load(dst, src, offs) => {
    set_reg(s, dst, get_mem(s,get_reg(s,src)+offs)
    )
    execute(i+1, s) }
  case Store(dst, src, offs) => {
    set_mem(s, get_reg(s,src)+offs, get_reg(s,dst)
    )
    execute(i+1, s) }
  case B(None, tgt) => execute(tgt, s)
  case B(Some((cmp, src1, src2)), tgt) =>
    if (eval_cmp(s, cmp, src1, src2)) {
      execute(tgt, s)
    } else {
      execute(i+1, s) } }
```

Figure 2. A definitional interpreter for nanoRISC

3.1 The naive staged interpreter

Our implementation uses the Lightweight Modular Staging (LMS) [13] framework for staging. LMS-based staged programs specify binding times via type annotations: the type `Rep[T]` denotes a value of type `T` only known in the residue (dynamic), while a bare type `T` denotes a value known during staging (static). The staged version of this interpreter adds the annotation `Rep` to the state parameter `s` and the result type of `execute`, marking that they are only known to the residue runtime (Fig. 3). We also replace the recursive calls to `execute` with an additional function `call` (elided) tracking which instructions have already been emitted to reduce duplication in the residue.

```
def execute(i: Int, s: Rep[StateT]): Rep[Unit] = {
  s.timer += 1 // *
  prog(i) match {
    // ...
    case Binop(op, dst, src1, src2) => {
      set_reg(s, dst, eval_op(s,op,src1,src2))
      call(i+1, s) // execute changed to call here
    }
    // ...
} }
```

Figure 3. The nanoRISC interpreter, staged

Consider a sample surface program (Fig. 4a) and its residue (Fig. 4b), the output of calling the staged interpreter. As our end-level analysis, we use the off-the-shelf C Bounded Model Checker (CBMC) [12] to enforce a noninterference property between the program's running time and any secret memory. Notably, CBMC is not actually a timing-aware model checker, which is precisely the point of collapsing towers – the residue makes time explicit as an ordinary C value, so CBMC can routinely reason about it. Concretely, we equip the `StateT` struct with a cycle counter that is incremented in the main interpreter loop (the marked line in Fig. 3), which will then be reflected as first-order code in the residue (marked lines in Fig. 4b).

Finally, our compiler wraps the generated code with a hand-written driver that uses CBMC primitives to check noninterference (Fig. 5).

This is already sufficient to expose basic timing information to CBMC. However, we aren't yet ready to find the timing leak in Fig. 4b, which requires both caching and speculation, which we detail next.

3.2 Caching and Speculation

We earlier claimed that a major advantage of using collapsing towers is that it is easy to amend the analysis to use a more complex model. In Fig. 6, we equip the naive interpreter with a two-entry LRU cache keyed by memory address.

```
mov r3, #pub
mov r0, #priv-pub
bge r0, #priv-pub, rest // x5
ldr r1, [r3, r0] // x3
ldr r2, [r1]
rest:
```

Figure 4a. The nanoRISC source

```
// ...
struct StateT x3(struct StateT x4) {
  x4.timer = x4.timer + 1; // *
  x4.regs[1] = x4.mem[x4.regs[3] + x4.regs[0]];
  return x1(x4); }
struct StateT x5(struct StateT x6) {
  x6.timer = x6.timer + 1; // *
  return x6.regs[0] >= 8 ? x6 : x3(x6); }
// ...
struct StateT Snippet(struct StateT x0) {
  return x9(x0); }
```

Figure 4b. Residue via the staged interpreter of Fig. 3

```
int main(int argc, char *argv[]) {
  struct StateT state1, state2;
  initialize(&state1);
  initialize(&state2);
  for (int i=PRIVATE_MEM_START; i<MEM_MAX; i++) {
    state1.memory[i] = bounded(0,20);
    state2.memory[i] = bounded(0,20); }
  state1 = Snippet(state1);
  state2 = Snippet(state2);
  __CPROVER_assert(
    state1.timer == state2.timer,
    "no timing leak");
  return 0; }
```

Figure 5. Enforcing noninterference in the residue

```
class StateT {
  regs, memory, ...,
  cache_keys, cache_values}
override def get_mem(s: Rep[StateT], addr: Rep[Int
    ]):
    Rep[Int] = {
  if (s.cache_keys(0) == addr) {
    // ... address is in cache, return value ...
    return s.cache_vals(0)
  } else if (s.cache_keys(1) == addr) {
    // ... swap entries ...
  } else {
    // address not in cache
    // ... evict oldest ...
    // Reading from MMU is expensive, increase
    // timer accordingly
    s.timer += 100
} }
```

Figure 6. Adjusting the interpreter to model the cache

Here, we take advantage of the fact that the base interpreter (Fig. 2) uses abstract `get_mem` and `set_mem` operations. Importantly, by hijacking just the memory lookup operations, we ensure that non-memory operations can fall back to the base timing model, only further adjusting the timer on cache miss. With this change (and the corresponding change to `set_mem`), our residue now models enough of the cache to detect the classic Evict + Time attack on AES[3, 15].

Because speculation changes control flow, we must instead override the `execute` function to intercept branch instructions (Fig. 7), falling back to `super.execute` appropriately.

The speculation driver in Fig. 7 also demonstrates the use of stage-time state to simplify the residue. The top-level interpreter knows whether the current instruction is being executed speculatively (via the stage-time `inBranch` variable), and instead simply emits code corresponding to a rollback at the appropriate point (line 25 in Fig. 8). Performing checks at staging time when possible means fewer additional variables in the residue, which limits the potential for path explosion beyond paths present in the original input.

Specializing the new interpreter to Fig. 4a gives the residue in Fig. 8. Here we can see the cache logic and a rollback from a mis-predicted branch both represented. Running CBMC against the residue in Fig. 8 now rejects, as desired.

3.3 Discussion

We have shown the end-to-end construction of a staged nanoRISC processor for microarchitectural side-channel analysis. Our approach starts from the obvious, naive interpreter and extends it to include caching and speculation. We again highlight the separation of concerns present in this implementation – the caching and speculation implementations are orthogonal and do not need to be explicitly aware of each other, yet the final result exhibits the desired vulnerability that arises from the interaction of the two.

4 Related Work

Staged programming. The use of staged interpreters to ease analysis has been explored in a series of prior work [7, 14, 17] also using the LMS framework. A common thread in existing work is that many properties of interest are more naturally expressed at runtime, and the Futamura projections allow us to model those properties at compile-time "for free". Our work applies the same methodology to locating side

```
// No further change to StateT!
var inBranch: Option[Instr]
override def execute(i: Int, s: Rep[StateT]):
    Rep[Unit] = {
  s.timer += 1
  inBranch match {
    case None =>
      // not currently speculating
      (i < prog.length, prog(i)) match {
        case (true, B(Some(cnd), tgt))
          // current instruction is branch, guess
          // that branch is not taken and execute
          // next instruction
          if tgt.unAddr > i => {
            inBranch = Some(B(Some(cnd), tgt))
            call(i+1, s) }
        case _ => super.execute(i, s) }
    case Some(Branch(rs, target)) => {
      // speculation in progress ...
      if (i == target) {
        inBranch = None
        // If branch was taken, rollback
        if (eval_cmp(s,cmp,src1,src2)) {
          rollback(s)
          // resume execution from rolled-back
          // state
          call(tgt.unAddr, s)
        }
        // ...
      } else if (i < prog.length) {
        prog(i) match {
          case Load(rd, im, rs) if rd != rs => {
            saveForRollback(s, Load(rd,im,r))
            super.execute(i,s)
          }
          // ...
} } } } }
```

Figure 7. Adjusting the interpreter for speculation

channels, using LMS to model a runtime dynamic property (namely, running time) to make it accessible to off-the-shelf static analysis tools.

Spectre-aware side channel analysis. Existing approaches to ahead-of-time discovery of microarchitectural vulnerabilities broadly fall into two categories – static analysis of some surface-level code using a formalization of the channel in question, or statistical analysis through fuzzing or machine learning.

We have already discussed Spectre-aware side channel analyses [4, 10] in Section 2.2, which fall into the former category. We generalize the idea approaches by effectively using the hardware interpreter itself to construct a new model by fusing the hardware semantics into the target program before analysis. Other work in this area includes CacheAudit [6],

```
// ...
int x8 = x2.cache_keys[0] == x7 ?
  x2.cache_vals[0] : (x2.cache_keys[1] == x7 ? ({
  // Push memory access to top of LRU
  int x9 = x2.cache_vals[1];
  x2.cache_keys[1] = x2.cache_keys[0];
  x2.cache_vals[1] = x2.cache_vals[0];
  x2.cache_keys[0] = x7;
  x2.cache_vals[0] = x9;
  x2.timer = x2.timer + 1;
  x9;
}) : ({
  int x10 = x2.mem[x7];
  x2.mem[x2.cache_keys[1]] = x2.cache_vals[1];
  x2.cache_keys[1] = x2.cache_keys[0];
  x2.cache_vals[1] = x2.cache_vals[0];
  x2.cache_keys[0] = x7;
  x2.cache_vals[0] = x10;
  // Reading from memory is slow
  x2.timer = x2.timer + 100;
  x10; }));
x2.regs[2] = x8;
// if branch was mis-predicted as not taken,
// perform rollback
if (x2.regs[0] >= PRIVATE_MEM_START) {
  x2.timer = x2.timer + 15;
  x2.regs[2] = x2.saved_regs[2];
  x2.regs[1] = x2.saved_regs[1]; }
return x2;
```

Figure 8. Residue C program of the source nanoRISC program of Fig. 4a with caching and staging.

which performs abstract interpretation of the input against a parameterized abstract model of cache states to obtain an overapproximation of all cache-based leakages for a given abstract memory, cache and event model.

Osiris [16] performs hardware fuzzing to identify potential side-channel gadgets by observing that almost all microarchitecture attacks follow the pattern of prime-trigger-measure, and thus generates instruction sequences that may allow known microarchitectural components to leak memory through side channels. Fuzzing results can be used to direct a collapsing-towers analysis by isolating possibly-vulnerable code sections to avoid performing an expensive whole-program analysis.

5 Conclusion and Future Work

This short paper presents our preliminary work towards lightweight evaluation of software-hardware contracts through specialization of a given hardware interpreter. We show how a naive ISA interpreter can be staged to make timing information explicit, allowing the software-hardware contract to be stated purely in terms of first-order code. We also demonstrate how the base interpreter can be extended to

compositionally support microarchitecture features with relatively simple modifications. We use each staged processor to collapse a sample source assembly program into a residue C program that reifies time as an ordinary C variable, so that an off-the-shelf C bounded model checker can reason about timing attacks. We highlight a sample source assembly program tried on each processor variant, identifying timing vulnerabilities in the cache and speculation variants but not the base, as expected.

We claimed that the use of collapsing towers exposes the invisible semantics of hardware to off-the-shelf tools. Thus far, however, we have only achieved satisfactory results through enforcing noninterference via bounded model checking. We intend to explore other directions or even design our own novel analyses (specialized to work on generated residues). The most promising techniques in our sights are *taint analysis* and *program slicing* to detect whether the timer depends on any secret value. A primary obstacle in implementing such analyses on the residue immediately is that the timer is never directly written to by a value within the program. Instead, the timer only becomes tainted by *implicit flows* induced by branching on a secret value.

Acknowledgments

We thank Thomas Bourgeat for initial guidance, brainstorming and prototyping. Cameron Wong and Nada Amin were partially supported by the Harvard Dean's Competitive Fund for Promising Scholarship and the National Science Foundation under Award No. 2303983.

References

[1] Nada Amin and Tiark Rompf. 2017. Collapsing towers of interpreters. *Proc. ACM Program. Lang.* 2, POPL, Article 52 (dec 2017), 33 pages. https://doi.org/10.1145/3158140

[2] Nada Amin and Tiark Rompf. 2017. LMS-Verify: abstraction without regret for verified systems programming. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 859–873. https://doi.org/10.1145/3009837.3009867

[3] Daniel J Bernstein. 2005. *Cache-Timing Attacks on AES.* https://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[4] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-Time Foundations for the New Spectre Era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 913–926. https://doi.org/10.1145/3385412.3385970

[5] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. Fact: a DSL for timing-sensitive computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 174–189.

[6] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2015. Cacheaudit: A tool for the static analysis of cache side channels. *ACM Transactions on information and system security (TISSEC)* 18, 1 (2015), 1–32.

[7] Grégory M. Essertel, Guannan Wei, and Tiark Rompf. 2019. Precise reasoning with structured time, structured heaps, and collective operations. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 157 (oct 2019), 30 pages. https://doi.org/10.1145/3360583

[8] Yoshihiko Futamura. 1971. Partial evaluation of computation process-an approach to a compilercompiler. *Systems, computers, controls* 2, 5 (1971), 45–50.

[9] Yoshihiko Futamura. 1999. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation* 12, 4 (1999), 377–380.

[10] Roberto Guanciale, Musard Balliu, and Mads Dam. 2020. InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1853–1869. https://doi.org/10.1145/3372297.3417246

[11] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-software contracts for secure speculation. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1868–1883.

[12] Daniel Kroening. [n. d.]. *The C Bounded Model Checker.* https://www.cprover.org/cbmc/

[13] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the ninth international conference on Generative programming and component engineering*. 127–136.

[14] Shangyin Tan, Guannan Wei, and Tiark Rompf. [n. d.]. Towards Partially Evaluating Symbolic Interpreters for All (Short Paper). ([n. d.]).

[15] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology* 23 (2010), 37–71.

[16] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. 2021. Osiris: Automated discovery of microarchitectural side channels. In *30th USENIX Security Symposium (USENIX Security 21)*. 1415–1432.

[17] Guannan Wei, James Decker, and Tiark Rompf. 2018. Refunctionalization of abstract abstract machines: bridging the gap between abstract abstract machines and abstract definitional interpreters (functional pearl). *Proc. ACM Program. Lang.* 2, ICFP, Article 105 (jul 2018), 28 pages. https://doi.org/10.1145/3236800