Highly Automated Verification of Security Properties for Unmodified System Software

Ganxiang Yang*
Columbia University
New York, NY, USA
ganxiang.yang@columbia.edu

Wei Qiang*
Columbia University
New York, NY, USA
wei.qiang@columbia.edu

Yi Rong*
Columbia University
New York, NY, USA
rong@cs.columbia.edu

Xuheng Li
Columbia University
New York, NY, USA
xuheng@cs.columbia.edu

Fanqi Yu
Columbia University
New York, NY, USA
fanqi@cs.columbia.edu

Jason Nieh
Columbia University
New York, NY, USA
nieh@cs.columbia.edu

Ronghui Gu
Columbia University
CertiK
New York, NY, USA
ronghui.gu@columbia.edu

Abstract

System software is often complex and hides exploitable security vulnerabilities. Formal verification promises bug-free software but comes with a prohibitive proof cost. We present Spoq2, the first verification framework to highly automate security verification of unmodified system software. Spoq2 is based on the observation that many security properties, such as noninterference, can be reduced to establishing inductive invariants on individual transitions of a transition system that models system software. However, directly verifying such invariants for real system code overwhelms existing SMT solvers. Spoq2 makes this possible by automatically reducing verification complexity. It decomposes transitions into individual execution paths, extends cone-of-influence analysis to the individual transition level, and eliminates irrelevant machine states, clauses, and control-flow paths before invoking the SMT solver. Spoq2 further optimizes how pointer operations are modeled and verified through pointer abstractions that eliminate expensive bit-wise operations from SMT queries. We demonstrate the effectiveness of Spoq2 by verifying security properties of four unmodified, real-world system codebases with minimal manual effort.

CCS Concepts: • **Software and its engineering → Formal methods**; • **Theory of computation → Automated reasoning**; • **Security and privacy → Operating systems security**.

Keywords: Formal verification, system software, automated verification, security properties, SMT solvers

ACM Reference Format:
Ganxiang Yang, Wei Qiang, Yi Rong, Xuheng Li, Fanqi Yu, Jason Nieh, and Ronghui Gu. 2026. Highly Automated Verification of Security Properties for Unmodified System Software. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3779212.3790171

*Equal contribution.

ASPLOS '26, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790171

1 Introduction

System software lies at the heart of modern computing systems and spans operating systems, hypervisors [7], and firmware. Applications inherently trust this software, yet its size and complexity often expose it to vulnerabilities, endangering the confidentiality and integrity of the entire system. Formal verification is a powerful technique for guaranteeing critical security properties, but has been notoriously difficult to apply in practice. Manually writing formal specifications, proving refinement, and verifying security properties all demand extensive expertise and effort. Previous approaches that have attempted to reduce proof burden—such as Spoq [56]—still require significant manual Coq proof work for security properties, while frameworks like Serval [64] automate security verification but do not support pointers. Moreover, the key challenge of automated verification of large systems is path explosion from considering all combinations of branch conditions and solver explosion from considering every possible state, compromising scalability and limiting their applicability to real-world system code.

We present Spoq2, a new verification framework that highly automates security verification of unmodified system software, bridging the gap between promising formal methods and the realities of large, complex codebases. It enables verification of concurrent software without requiring users to invest significant manual effort to write refinement or security proofs. Spoq2 takes as input a set of security properties, a software implementation to be verified, and a machine configuration that specifies abstract machine states, including registers and memory objects. It constructs a transition system that models the behavior of the software implementation, with its states defined by an abstract machine model generated from the machine configuration, and its transitions defined by a Coq specification generated from and refined by the implementation. Instead of trying to prove the security properties directly over the entire specification, Spoq2 reduces properties into invariants and checks if each invariant is inductive [13]—that it holds initially and is preserved for each execution path of an individual atomic transition in the transition system. By making each proof goal well-defined yet small enough, they can be verified automatically using the Z3 SMT solver [24], then composed together to verify the overall implementation.

To handle the large transition systems found in real system software, Spoq2 employs a suite of new optimization techniques grounded in classic model-checking ideas, adapted for efficient security reasoning. First, Spoq2 advances Cone-of-Influence (COI) analysis to prune irrelevant conditionals and machine states, mitigating path and solver explosion by identifying what can influence the target security property and discarding the rest. By decomposing the proof for each property down to verifying inductive invariants, Spoq2 applies COI for the first time at a fine-grained per-transition level rather than over the entire program because each transition can be verified independently against the invariants. Spoq2 extends COI's reach beyond conventional state pruning to the whole verification pipeline, reducing redundant proof goals and clauses. Second, Spoq2 introduces pointer abstractions that uniformly model complex memory address manipulations—common in system software—and translate them into simple attribute accessors, avoiding slow bit-wise arithmetic in Z3 queries. Finally, Spoq2 caches SMT formulas and Z3 query results so they can be reused across similar code paths.

We used Spoq2 to verify four unmodified system codebases: the Realm Management Monitor (TF-RMM v0.3.0) [3] for Arm's Confidential Computing Architecture (CCA) [55]; the Trusted Firmware-A (TF-A v2.13) [58] EL3 secure world software for Arm-A class CPUs; the SeKVM hypervisor [51–54] for Linux 6.1; and the Komodo security monitor [26]. Across all codebases, Spoq2 verifies system code as is, without retrofitting for verification, and automatically proves security properties with minimal manual effort. Compared to prior approaches, Spoq2 reduces manual effort by up to 80%—cutting thousands of lines of security proofs down to fewer than 200 lines of definitions—and enables verification tasks that once timed out after days to complete within minutes. Our results demonstrate Spoq2's effectiveness as the first framework that delivers highly automated security verification for unmodified system software. Spoq2 is open sourced at https://github.com/VeriGu/spoq3.

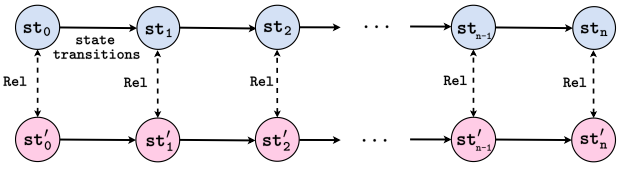
```
1 struct page_t pages[MAX_SIZE];
2 void free_page(struct page_t* p) {
3     if (p->status != FREE) {
4         clear_page(p->data);
5         p->tag = NS;
6         p->status = FREE;
7 }}
```

Figure 1. An example to free a page (locks omitted).

2 Overview

We show how Spoq2 automates security verification using the simplified C function `free_page` in Figure 1, which releases a memory page. The example assumes a confidential computing architecture with two worlds: Secure and Non-Secure (NS). Secure world hosts confidential applications, while NS world runs all other software, including untrusted system code. Memory pages originate in NS world, may be allocated to Secure world, and then become inaccessible to NS world. When no longer needed, they are freed back to NS world. Each page has a tag indicating the world to which it is allocated; a page tagged `SECURE` cannot be accessed from NS world. `free_page` checks the page status; if the page is not yet freed, it clears the page's data with `clear_page`, resets the status, and sets the tag to `NS`. Lock operations are omitted for simplicity.

Security Properties. We want to prove that the contents of a page with the `SECURE` tag are never leaked to NS world. Such properties are typically formalized as noninterference assertions [75]; if two executions of the system differ only in secure page contents, then the data observable in NS world, referred to as public data, should be the same. In the example, public data consists of the fields `status` and `tag`, and `NS`-tagged page content `data`. `SECURE`-tagged page contents are considered private. Two states that differ only in private data are indistinguishable states. Noninterference is proven by showing that if two executions of the system start from indistinguishable states, their resulting states remain indistinguishable:

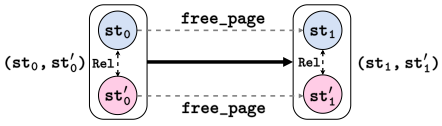


We use `Rel` to denote the indistinguishability relation between the states of the system across two executions:

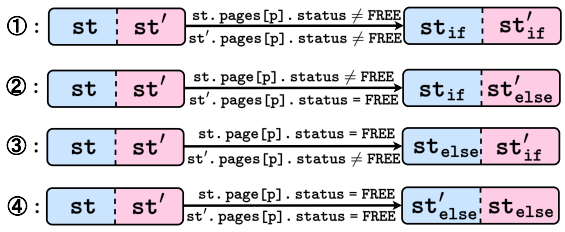
```
Relation Rel st st' := forall p,
  st.pages[p].status = st'.pages[p].status
  /\ st.pages[p].tag = st'.pages[p].tag
  /\ (st.pages[p].tag = NS /\ st'.pages[p].tag = NS
      => st.pages[p].data = st'.pages[p].data).
```

Reduction to Inductive Invariants. Noninterference proofs require showing that indistinguishability is preserved across the entire system execution—a task that is difficult to automate and costly to verify manually. For example, proving noninterference for a Linux KVM hypervisor required 4.8K lines of manual Coq proofs [53].

Instead of proving noninterference directly, Spoq2 reduces the problem to establishing an inductive invariant. An invariant is inductive if it holds in all initial states of the system and is preserved by each valid transition, ensuring it remains true in all reachable states. Because noninterference compares two executions of the system, Spoq2 constructs a *composed* transition system that merges these two executions, and proves invariants over this composed system instead of the original one. In this composed system, the global state is a pair (`st`, `st'`) of states from each original execution, and any transition is applied simultaneously to both `st` and `st'`. Thus, noninterference amounts to proving that an invariant—requiring `st` and `st'` in the composed system (`st`, `st'`) to satisfy `Rel`—holds initially and is preserved by each atomic transition (e.g., `free_page`). This composed system can be shown as:

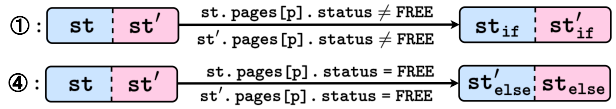


Decomposing Invariant Proofs. The benefit of this reduction is that the invariant for each atomic transition can be verified independently, without reasoning about the rest of the system's behavior. However, each atomic transition may still contain multiple execution paths, making automated verification challenging. Therefore, Spoq2 enumerates all possible execution paths for each atomic transition and generates a separate proof goal for every path. Note that in the composed system, the two states may follow different paths. Thus, if an atomic transition has n distinct paths, up to $n \times n$ decomposed goals may result. The following diagram illustrates the four decomposed goals for the `free_page` function, arising from the branch in line 3 of Figure 1:



The diagram also marks the associated path constraints, which state certain preconditions that must be true. For example, the second proof goal shows that the state ($\texttt{st}_{\text{if}}$, $\texttt{st'}_{\text{else}}$) is only reachable when `st.pages[p].status!=FREE` and `st'.pages[p].status= FREE`.

Pruning Paths. Enumerating all execution paths produces independent proof goals, but may also cause path explosion. To efficiently prune invalid paths, Spoq2 employs a data dependency analysis technique called Cone-of-Influence (COI) [13], which conservatively computes a set of variables—referred to as the COI set—that may influence the value of a given variable within a transition. In the case of noninterference, since the two executions differ only in private data initially, a variable whose COI set contains no private data must have the same value in both executions. In other words, if COI analysis determines that there is no information flow from private data to a public variable, then the value of that public variable is guaranteed to remain identical across both executions. Consider the branch condition at line 3 of Figure 1. It depends only on the field `status`, and the COI set of `status` contains no private data. Therefore, the branch condition is independent of private data, and both executions must take the same branch path. Thus, only two proof goals remain valid:



Below is the proof goal for the `if` branch. The transition is expressed as a relation between the pre-transition state `st0` and the post-transition state `st1`, where we use "`d {f: v}`" to denote updating field `f` in `d` to the value `v`:

```
(* state transition on the if branch *)
Transition FreeIf st0 st1 p:=
  st0.pages[p].status != FREE (* Path constraints *)
  /\ st1 := st0 { pages[p].data: ZERO_DATA }
                { pages[p].tag: NS }
                { pages[p].status: FREE }.

(* proof goal for the if branch *)
Goal Rel_Free_If := forall st0 st0' st1 st1' p,
  Rel st0 st0'/\ FreeIf st0 st1 p /\ FreeIf st0' st1' p
  => Rel st1 st1'.
```

Note that COI is a fast but conservative analysis. Even if private data appears in a variable's COI set, it does not necessarily mean that an actual information flow exists. While COI helps prune proof goals and reduce expensive queries, it cannot replace precise reasoning via Z3. Although COI analysis was originally developed for model checking [13], its application to security reasoning has historically been difficult. Traditional COI analysis examines the entire system at once and conservatively marks most variables as potentially influencing others, limiting its effectiveness in pruning proof goals. In contrast, Spoq2 makes COI analysis practical and effective by applying it to independent proof goals focused on individual transitions. This localized analysis produces tighter COI sets and allows Spoq2 to prune significantly more proof goals (see §5).

```
1 void free_pte(u64 pte) {
2   if (unlikely(!(pte & _PG_VALID))) return;
3   struct page_t* p =
4     (struct page_t*)((pte & _PG_MASK)- PA + (u64)&pages
      );
5   free_page(p);
6 }
```

Figure 3. Free a page with page table entry.

Optimizing Z3 Queries. For proof goals that cannot be eliminated, Spoq2 further simplifies the corresponding Z3 queries using COI analysis. In the running example, if a public variable's COI set does not include private data, its indistinguishability check in the relation `Rel` can be safely omitted from the proof goal, since its value is guaranteed to be identical in both executions. Because the COI sets of both `status` and `tag` exclude private data, their checks can be removed. After unfolding `Rel` in the check part, the resulting simplified proof goal, `Rel_Free_If`, becomes:

```
Goal Rel_Free_If' := forall st0 st0' st1 st1' p,
  Rel st0 st0'/\ FreeIf st0 st1 p /\ FreeIf st0' st1' p
  => (st1.pages[p].tag = NS /\ st1'.pages[p].tag = NS
      => st1.pages[p].data = st1'.pages[p].data).
```

Since the transition `FreeIf` updates `tag` to `NS` and `data` to `ZERO_DATA`, substituting these values yields the following simplified proof goal, which can be trivially verified by Z3:

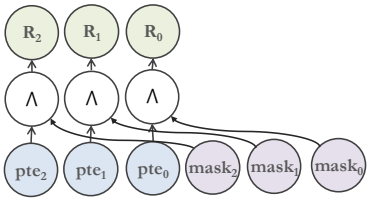
```
Goal Rel_Free_If'' := forall st0 st0' st1 st1' p,
  Rel st0 st0'/\ FreeIf st0 st1 p /\ FreeIf st0' st1' p
  => (NS = NS /\ NS = NS
      => ZERO_DATA = ZERO_DATA).
```

Identifying Security Violations. If the security property does not hold, Spoq2's Z3 queries will generate a counterexample, provided Z3 terminates in time. In our running example, if the page content is not zeroed by calling `clear_page`, noninterference is violated: once the page tag changes to `NS`, previously private page content `data` becomes observable. Thus, the two system executions become distinguishable due to differences in their private data.

Pointer Verification. Figure 3 shows an example using `free_page` to free a page table entry, which involves pointers. Handling pointers is crucial as they are extensively used in system software, often in complex, bit-level pointer operations. Modeling memory accesses via pointers is complicated because it often involves multiple levels of conditional branches to check which memory object is being referenced. Furthermore, bitwise operations appear in state updates, which then need to be encoded into Z3 queries. For instance, the branch condition "`!(pte & _PG_VALID)`" in Figure 3 is translated into a path constraint in the Z3 query:

```
(* the path constraint for !(pte & _PG_VALID)*)
cond := (bvand pte _PG_VALID = 0).
```

Handling such queries requires the use of the *bit-vector* theory, which involves reasoning over individual bits of a variable in the state-of-the-art *bit-blasting* approach. For 64-bit variables, this means solving 64 propositional constraints—one for each bit. The example below illustrates this process for a simplified 3-bit variable, showing the computation of results R_0, R_1, and R_2:



Z3 queries involving full 64-bit variables are significantly more complex and computationally expensive [9].

To address these challenges, Spoq2 introduces pointer verification optimizations. A *pointer abstraction* provides an abstract interface and operations for handling common pointer manipulations in systems code. For example, the abstract pointer interface for `pte` includes two attributes: `validity` and `addr`. The branch condition in Figure 3 is mapped to reading the `validity` attribute, while "`pte&_PG_MASK`" at line 4 is mapped to reading the `addr` attribute. This abstraction eliminates bit-level operations from the Z3 queries. For example, the path constraint for "`!(pte&_PG_VALID)`" no longer requires *bit-vector* theory:

```
(* the path constraint without bit operations *)
cond' := (pte.validity = 0).
```

Spoq2 introduces transformation rules to simplify pointer dereferencing and pointer-related expressions, applying them only when Z3 can prove the simplified expression is equivalent to the original. To avoid the overhead of repeated Z3 queries for common pointer operations, Spoq2 caches Z3 query results so they can be used (see §6).

3 Spoq2 Verification Workflow

Figure 4 shows in more detail the workflow for how Spoq2 verifies a set of security properties for a software implementation:

Step 1. Spoq2 begins by compiling the unmodified system code into LLVM IR using Clang [45].

Step 2. The user provides a *machine configuration*, which Spoq2 uses to construct an abstract machine model. The model is a sequentially consistent (SC) [44] local CPU model that includes an event log and oracle to model concurrent execution interleavings from other CPUs [34, 56]. The configuration specifies the machine states, including registers and memory objects, such as stacks and global data. The configuration for the example in Figure 1 is:

```
Record Page:= { status: Z; data: ZMap.t Z; tag: Z }.
Record State:= { regs: Registers; pages: ZMap.t Page }.
```

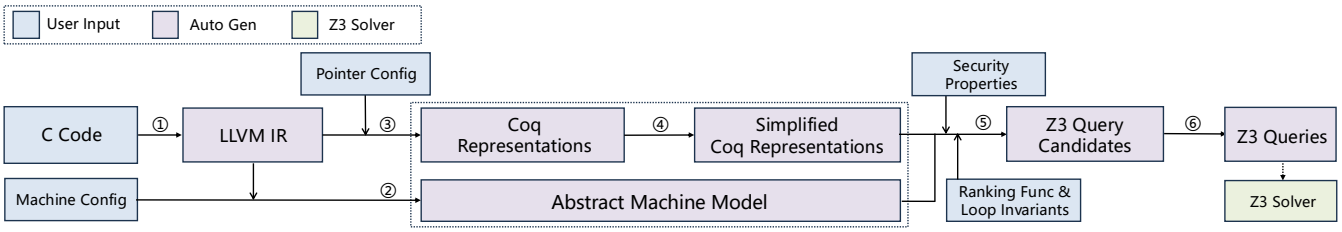


Figure 4. Spoq2 workflow.

The machine state `State` consists of CPU registers and a memory object `pages`. The `pages` array corresponds to the `struct page_t page[MAX_SIZE]` definition in the code.

Step 3. Spoq2 translates each LLVM IR function into a Coq representation. The user can provide a small *pointer configuration* for pointers overloaded with metadata that specifies the bit ranges for each attribute. For example, the pointer configuration for `pte` specifies that a one-bit valid attribute is stored in the bit range [0, 1), while the address information resides in [12, 64):

```
Config PTEPtr := {"valid": [0, 1), "addr": [12, 64)}.
```

Step 4. Spoq2 employs a set of verified transformation rules [57] to simplify the Coq representation into a self-contained form free of calls to other functions, while preserving semantic equivalence. The resulting self-contained Coq representations of the system interface functions collectively model the behavior of the system atop the abstract machine model as a transition system, where each transition is the Coq representation of an interface function. For example, Figure 5 shows the Coq representation of `free_page` from Figure 1, which takes a page pointer `p` and a state `st` as inputs and computes the resulting state after the transition.

Step 5. The user specifies the desired security properties as inductive invariants in Coq using the abstract machine states. Spoq2 automatically traverses each interface function's Coq representation to enumerate all execution paths, and prepares candidate Z3 queries for each path to prove that the invariants hold inductively.

The user also provides a ranking function [56, 84] and a loop invariant [71, 82] for each loop, which Spoq2 uses to check that the loop terminates and the loop invariant is inductive. The ranking function must be nonnegative and strictly decreasing for each loop iteration. Spoq2 checks each loop invariant and the ranking function's validity using Z3. If the check fails, Spoq2 terminates and reports failure.

Step 6. Spoq2 employs COI analysis to eliminate unnecessary Z3 queries and applies various optimization techniques to simplify and accelerate the remaining ones. It then invokes Z3 to discharge the resulting queries.

Trusted Computing Base. In Steps 2-3, Spoq2 trusts the user-provided configuration for the machine and pointers.

```
(* generated and simplifed for illustration *)
Definition free_page_coq (p: PTEPtr) (st: State) :=
   if (st.pages[p].status != FREE)
      st {pages[p].data : ZERO_DATA}
         {pages[p].tag: NS }
         {pages[p].status : FREE}
   else st. (* Remain unchanged in the else branch *)
```

Figure 5. Coq representation of `free_page` (locks omitted).

```
(* invariant on free pages, input from user *)
Invariant Inv st := forall p,
  st.pages[p].status = FREE
  => st.pages[p].data = ZERO_DATA.

(* proof goal for the initial state *)
Goal Inv_Init := Inv st_init.

(* proof goals for each transition *)
Goal Inv_Tran0 := forall st, Inv st => Inv (trans0 st).
Goal Inv_Tran1 := forall st, Inv st => Inv (trans1 st).
...
```

Figure 6. Invariant `Inv` and its proof goals.

In Step 3, Spoq2 trusts a small translator to translate LLVM IR and assembly into Coq representations. In Step 4, Spoq2 trusts the Coq proof assistant when proving the correctness of the transformation rules used. In Step 5, Spoq2 trusts the user-defined security properties. In Steps 5-6, Spoq2 trusts its own proof checker that implements its COI analysis and the Z3 SMT solver.

4 Proof Goal Construction

Spoq2 supports automated verification of properties that can be reduced to inductive invariants over a transition system. This class includes safety properties [42, 43], most information-flow security properties [22, 23, 35, 55, 67], and a wide range of k-safety hyperproperties [14]. System availability properties, such as denial-of-service [63], typically cannot be reduced to inductive invariants and are therefore beyond the scope of Spoq2.

To construct proof goals that Z3 can use to verify inductive invariants, Spoq2 ensures that each goal falls within decidable fragments of first-order logic augmented with SMT theories [62]. Specifically, Spoq2 (1) provides expression templates that restrict quantified variables to finite domains, while preserving expressiveness by issuing warnings if unrestricted quantifiers are used, and (2) formulates queries

```
Goal Inv_FreePage_If := forall st p,
  Inv st /\ (st.pages[p].status != FREE)
  => Inv (st { pages[p].data : ZERO_DATA }
             { pages[p].tag : NS }
             { pages[p].status : FREE }   ).
```

Figure 7. Proof goal of `if`-branch in `free_page` for `Inv`.

to check invariants using decidable SMT theories, including linear integer arithmetic, arrays, and bit-vectors [6].

Safety Properties. Safety properties ensure that the system does not reach bad states, and can be formulated as inductive invariants [42, 43]. For example, for the `free_page` example in Figure 1, a desired safety property is that a freed page should never contain nonzero data. This can be formulated as the invariant `Inv` shown in Figure 6. Spoq2 verifies such invariants by constructing proof goals for Z3 to check that the initial states satisfy the invariant, and that each atomic transition inductively preserves it. These proof goals are also listed in Figure 6. The proof goal for the initial state is typically straightforward to check. For each atomic transition, Spoq2 traverses all execution paths within the transition. For `free_page`, Figure 7 is the proof goal generated for the only non-trivial path, which is the branch with the path constraint `(st.pages[p].status!=FREE)`. If the execution path violates the invariant or the provided invariant is not inductive, Z3 returns a counterexample.

An important safety property useful to verify security properties for concurrent systems is data race freedom (DRF)—ensuring that locks are used properly so that access to shared objects are well synchronized. Spoq2's machine model provides locks which it uses its event oracle to check are properly acquired and released when accessing shared memory objects, otherwise the machine will halt and not be able to complete execution [34]. Using this model, we can prove DRF by constructing proof goals for Z3 that verify the inductive invariant that the machine will never halt on any transition. This can be used to simplify proofs for other security properties, which can then be conducted using sequential reasoning without further consideration of the effects of concurrent interleavings once DRF has been proven. While Spoq2 constructs proofs based on an SC model, its proofs can be extended to relaxed memory hardware by additionally verifying VRM's six weak-data-race-free invariants [79].

Safety properties may also be useful to verify other security properties that may themselves not be inductive invariants. In this case, it may be necessary to verify additional safety properties as inductive invariants such that the conjunction of such invariants with the security property of interest becomes inductive. Spoq2 relies on the user to identify such invariants, though other tools have been developed to help automate finding inductive invariants [83, 85].

Information-Flow Security. Many information-flow security properties can be reduced to invariants over a composed system, such as confidentiality [4, 28], integrity [5, 12], noninterference [60, 75], relaxed noninterference [50], and step consistency [70]. For such properties, the user provides a relation `Rel` over a state pair (`st`, `st'`) in the composed system. The proof goal is then to show that: (1) `Rel` holds on the pair of initial states, and (2) `Rel` is preserved by all pairs of execution paths for each transition in the composed system. The proof goal template for proving the relation `Rel` over the composed system is formulated in Figure 8.

The `free_page` example in §2 demonstrates how to use `Rel` to express the confidentiality property that `SECURE` page contents must not be leaked to the `NS` world, and how to generate corresponding proof goals using the proof goal template. Here, we present another `free_page` example to verify a form of *relaxed integrity* [50]: the content of a `SECURE` page must not be influenced by public data, except when the page tag is changed to `NS`. This property can be encoded as a relation `Rel` over a pair of states in the composed system:

```
Relation Rel st st' := forall p,
  (st.pages[p].tag= SECURE /\ st'.pages[p].tag= SECURE)
  => st.pages[p].data = st'.pages[p].data.
```

`Rel` specifies that the page contents in the two executions must be equal whenever both tags are `SECURE`, regardless of differences in public data. For instance, the public variable `status` used in the conditional for `free_page` in Figure 1 may differ across two executions. When the `status` values are equal, both executions follow the same path, and Spoq2 can easily verify that the invariant `Rel` holds. The non-trivial case arises when `status` values differ in two executions and `st` follows the first branch and `st'` takes the second branch. The proof goal following the template in Figure 8 is:

```
Goal Rel_FreePage_If_Else := forall st st' p,
  Rel st st' /\
  st.pages[p].status != FREE /\ (* first branch *)
  st'.pages[p].status = FREE /\ (* second branch*)
  => Rel (st { pages[p].tag : NS }
             { pages[p].data : ZERO_DATA }
             { pages[p].status : FREE }   ) st'.
```

K-Safety Hyperproperties. This proof goal template can be extended to construct proof goals for k-safety hyperproperties [14], such as *Goguen-Meseguer noninterference* [28] and *observational determinism* [86]. While information-flow security involves two executions of one system, k-safety properties are a more generalized version that can involve k executions of up to k systems. Such properties can be reduced to inductive invariants of a composed system that merges `k` executions where each execution is potentially for a different system, and `Rel` is defined over the resulting `k` states. For example, the ideal/real machine model [55] compares two executions on two systems, specifically an execution on the system being verified against an execution of an ideal machine. Other than requiring an ideal machine specification for the second system, constructing proof goals is similar to doing so with the composed system described in §2.

```
(* proof goal for two initial states *)
Goal Rel_Init := Rel st_init st_init'.

(* proof goal template for any pair of paths *)
Goal Rel_Path_ij := forall st st',
  Rel st st' /\ PathCond_i st /\ PathCond_j st'
  => Rel (path_i st) (path_j st').
```

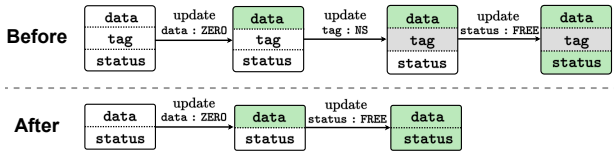
Figure 8. Proof goal templates for relation `Rel`.

5 Proof Goal Optimization

Although Spoq2 decomposes security properties into many independent proof goals over individual transitions, verifying these goals can still be computationally expensive or even infeasible for complex systems. We identify three main sources of complexity: 1) the intricate structure of the abstract machine state, 2) the large number of generated proof goals, and 3) the extensive state variable updates within each transition function. All of these contribute to a heavy reasoning burden for Z3. Spoq2 adapts COI analysis to: 1) eliminate irrelevant machine states given the desired property, 2) prune unnecessary proof goals, and 3) simplify the clauses within each individual goal.

5.1 Machine State Simplification

Spoq2 employs Cone-of-Influence (COI) analysis to prune irrelevant machine state updates for each desired property. The soundness of the COI analysis guarantees that properties verified on the simplified machine state still hold on the original system [13]. Consider the invariant proof of `Inv` in Figure 6 for the Coq representation of `free_page` in Figure 5. We observe that, regardless of how `free_page_coq` updates the value of `tag`, the preservation of `Inv` is unaffected. This is because `Inv` does not reference `tag`, and the updates to `tag` do not influence any variables involved in `Inv`. As a result, we can verify `Inv` on a simplified machine state that omits `tag` and its updates. Under this simplification, the transition for the `if`-branch path of `free_page_coq` reduces to the following form when proving `Inv`:



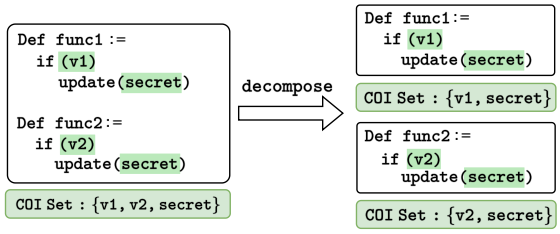
Spoq2 simplifies the machine state separately for each transition. Given a property and a transition's Coq representation, Spoq2 performs three steps. First, Spoq2 computes the COI set of variables in the transition that may influence the desired property. Second, it removes all irrelevant variables outside this COI set from the machine state, along with their corresponding accesses in the transition. Third, it traverses the simplified Coq representation to generate proof goals.

Step 1. Spoq2 determines a set of state variables that may influence the desired property in the transition by computing the COI set defined as follows:

Definition 1 (COI Set)**.** The COI set C for a set V of variables is defined as the minimal set of variables satisfying the following conditions:
1) C includes all variables in V.
2) If $s \in C$ and the update of s in the transition depends on a variable v, then $v \in C$.
3) If $s \in C$ and s is updated within a branch, then each variable v referenced in the branch condition must also be in C.
4) If $s \in C$ and s is updated within a loop body, then each variable v referenced in the loop condition must also be in C.

Given a desired property, the COI set is computed for V as the set of all variables referenced in the desired property. For example, consider the desired property `Inv` in Figure 6, which references the variables $V = \{\texttt{status}, \texttt{data}\}$. Applying the COI computation algorithm to the `free_page_coq` in Figure 5 yields the COI set $C = V = \{\texttt{status}, \texttt{data}\}$.

When constructing a COI set for security properties over the entire system, the result often includes most of the machine state, making it ineffective for simplification. By contrast, Spoq2 decomposes the global proof into sound, independent invariant proofs for individual transitions, and computes a separate COI set for each transition. This localized analysis dramatically improves simplification precision and effectiveness at the transition level, as shown in the following example:



Per-transition COI analysis is sound for inductive proofs, a well-established fact in model checking [13].

Step 2. Given the COI set, Spoq2 simplifies the transition's Coq representation by removing state updates to variables outside the COI set for all expressions. For example, since `tag` is not in the COI set C, the simplified Coq representation of `free_page_coq` becomes:

```
Definition free_page_coq' (p: PTEPtr) (st: State):=
   if (st.pages[p].status != FREE)
      st {pages[p].data : ZERO_DATA}
         {pages[p].status : FREE}
   else st. (* st_else*)
```

Step 3. Spoq2 symbolically traverses the simplified Coq representation that excludes irrelevant state updates to generate proof goals for each execution path. Instead of the original proof goal shown in Figure 7, the proof goal for the `if`-branch can be simplified to:

```
Goal Inv_FreePage_If' := forall st p,
  Inv st /\ st.pages[p].status != FREE
  => Inv (st { pages[p].data : ZERO_DATA }
             { pages[p].status: FREE }    ).
```

5.2 Reducing the Number of Proof Goals

Spoq2 reduces the number of proof goals that need to be checked in two ways. One is that, after applying state simplification in §5.1, it merges execution paths that yield identical state transitions, simplifying the control-flow graph of the Coq representation and reducing the number of proof goals. Another is that it also leverages COI analysis to eliminate proof goals whose path constraints are contradictory. Given a path constraint, the COI set is computed for V as the set of all variables referenced in the path constraint.

For example, consider the noninterference proof for `free_page` in §2 in which two executions only differ in private data, along the proof goal template shown in Figure 8. One candidate goal arises when the first execution takes the `if`-branch, while the second execution takes the `else`-branch. Assuming the states at the branch condition are `st` and `st'`, the path constraints are:

```
st.pages[p].status != FREE    (*run 1; if-branch*)
/\ st'.pages[p].status = FREE. (*run 2; else-branch*)
```

Since the path constraints depend only on `status`, Spoq2 computes the COI set for $V = \{\texttt{status}\}$ over the Coq representation `free_page_coq'`, yielding $C = \{\texttt{status}\}$. This indicates that no other variable influences `status` in this transition. Because the two executions differ only in private data and `status`'s COI set contains no private data, `status` must therefore take the same value in both executions:

```
st.pages[p].status = st'.pages[p].status.
```

This contradicts the assumed path constraints, allowing Spoq2 to soundly eliminate this proof goal.

5.3 Reducing Clauses Within Proof Goals

Spoq2 further reduces the clauses inside each proof goal. This is done when proving security properties using a composed transition system, such as information-flow security. Verifying such properties against real-world systems places a heavy burden on SMT solvers due to the complicated definition of the relation required for security proofs.

For example, for information-flow security, Spoq2 leverages COI analysis as a sound data dependency tool to track influences from private to public data variables, or vice versa, and skip proving trivial clauses in the relation. If a public data variable `v` is not influenced by any private data variable, then its equality clause `st1.v = st2.v` does not need to be checked after transition and can be removed from the final Z3 query, because if `v` only depends on public data variables, which are assumed to be identical in two executions before transition, then the equality clause must hold after transition. When COI analysis detects that a private data variable may influence a public data variable, it does not necessarily imply a secret leak. Since COI analysis is an over-approximation, Spoq2 must still pass the corresponding proof goals to Z3.

5.4 Caching Z3 Queries

During Coq generation, Spoq2 applies transformation rules and uses Z3 to eliminate branches whose conditions can be statically determined. However, for unresolved branch conditions that appear repeatedly, Z3 reattempts to resolve them each time—even when the result is unknown. Similarly, during property verification, the same expressions may occur multiple times. Each time an expression is encoded for Z3, a fresh expression node is constructed by traversing the Coq expression. As a result, the Z3 encoding process becomes heavily duplicated and inefficient for large systems. Spoq2 introduces two Z3 caches that store hashes of expressions and queries, respectively. When a Z3 expression or query appears again, the cached result is returned directly, avoiding redundant interaction with the Z3 SMT solver.

6 Pointer Verification

Pointer Usage. System software frequently deals with pointers—variables whose values are memory addresses—and overloads their representations in memory with additional metadata such as error codes, permission bits, and valid flags, which are often manipulated using bitwise operations. Because C permits casting arbitrary values to pointer types, any variable may be interpreted as a pointer and used for memory access, not just those that are declared as pointer types. We observe several common pointer uses in system software:

Example 1. Pointer dereference:

```
(a) ptr = &pages; val = *ptr;
(b) ptr = &pages; *ptr = val;
```

A pointer contains address information and is dereferenced to load or store values. The pattern `(a)` loads a value from a location in the memory object `pages` the pointer refers to, and the pattern `(b)` stores a value to that location.

Example 2. Tagged pointers:

```
(c) u8 tag = (u64)(addr) >> 56;
(d) u8 addr_tag = (addr & ~TAG_MASK) | (tag << 56);
```

A tagged pointer stores metadata directly as a tag in its highest bits to improve performance [10] and memory safety [29]. The pattern `(c)` retrieves the tag from its highest eight bits. The pattern `(d)` yields `addr_tag` with the address of `addr` and a replacement tag `tag`.

Example 3. Page table entries:

```
(e) pte = addr | L_RDONLY_FLAG
(f) addr = pte & _PAGE_MASK;
```

A page table entry stores address information and other metadata. The pattern `(e)` constructs a page table entry by attaching the read-only permission bit to the non-overlapping bits of an aligned physical address. The pattern `(f)` reads and masks the page table entry to get the physical address.

```
1 void free_pte(u64 pte) {
2   ...
3   struct page_t* p =
4    (struct page_t*)((pte & _PG_MASK)- PA + (u64)&pages
       );
```

Figure 9. An example of computing a linearly mapped pointer from a page table entry.

Example 4. Physical address translation:

```
(g) va = (u64*)(addr - PHYS_OFFSET + PAGE_OFFSET)
```

The pattern `(g)` maps a physical address `addr` to its corresponding kernel virtual address `va` assuming the kernel uses a linear map of RAM mapped to its address space. `PHYS_OFFSET` is the physical address at which system RAM starts. `PAGE_OFFSET` is the virtual address where the kernel's linear map of physical memory starts.

Example 5. Error pointers:

```
(h) #define ERR_PTR(err) ((void *)((long)(err)))
```

In Linux, runtime errors are often encoded into pointer values when propagating across function calls [69]. The pattern `(h)` is a macro used to wrap an error code into a pointer.

Such common pointer operations pose a significant challenge for verification, primarily because of the presence of metadata overloaded into the pointer representation that requires bitwise operations on the pointer. Verification frameworks generally either avoid them entirely [8, 64], or treat arithmetic and bit operations on pointers as standard integer operations [9, 48, 68]. As discussed in §2, the latter relies on bit-vector theory, which is computationally expensive and remains a major bottleneck for SMT-based verification [37]; state-of-the-art *bit-blasting* fails to scale even for 32-bit vectors [66]. As a result, it is important for Spoq2 to avoid Z3 queries that require checking complex bitwise operations on 64-bit pointers to make its SMT-based verification practical for large systems.

Pointer Abstraction. Spoq2 addresses this problem by introducing a pointer abstraction that constructs abstract interfaces and operations for pointer representations overloaded with metadata, commonly used in system software. This can eliminate complex bitwise operations involving such pointers from Coq representations, which also enables the transformation rules used by Spoq2 to simplify such representations to be more effective. Using these simplified Coq representations avoids the need for Z3 queries to check proof goals that contain complex bitwise pointer operations, enabling the queries to be handled efficiently.

The pointer abstraction requires the user to provide a configuration file that specifies three types of information, a list of storage formats, a list of hints, and a layout. A storage format defines the bit ranges for the address and associated metadata. A hint indicates where and how a storage format is used, specifically in what function it is used and whether it is used as an argument or return value. Each pointer of interest will have a hint, but multiple hints may use the same storage format. A layout encodes low-level memory layout assumptions that are typically not directly defined by the source code. Every configuration file must have at least a storage format and hint, but the layout is optional if there are no required assumptions defined outside of the memory objects in the source code. For example, the pointer configuration for Figure 9 is:

```
(* User-specific configuration for tagged pointer *)
Config PTEPtr := {"addr": [12, 64), "valid": [0, 1) }.
Config Hint :=   {"free_pte" : ["arg_0", "PTEPtr"] }.
Config Layout := {"start" : PA, "object": pages }.
```

The `PTEPtr` format states the bit range of address information and the valid bit in the `pte`. The hint indicates that the first argument of `free_pte` should use the format `PTEPtr`. The layout indicates that the integer address constant `PA` is the start of the memory object `pages`, which is necessary in this example because the code makes assumptions regarding how physical memory is organized that are not defined explicitly by the memory objects in the code.

Spoq2 supports its pointer abstraction by extending the CompCert [49] memory model to represent a pointer as a Coq record based on the storage format. This record consists of address information as well as other attributes. The address consists of a pair of fields, an identifier `id` and an offset `ofs`. The identifier `id` refers to the start of a memory object, and the offset `ofs` refers to the offset from the start of the memory object that the address specifically references. For overloaded pointer representations, Spoq2 uses one integer field for each additional attribute. For example, the Coq record for the page table entry in Figure 9 is:

```
(* Generated Coq record for page table entry *)
Record PTEPtr := {id: Z, ofs: Z, valid: Z}.
```

Spoq2 identifies the code pattern in the LLVM IR and maps the pointer operations to the corresponding record fields. In standard C, pointer operations consist of addition, subtraction, comparison, integer/pointer cast, and dereference. Spoq2 maps addition and subtraction to the field `ofs`, and comparison between two pointers of the same memory object to their `ofs`. Spoq2 models integer/pointer cast with two uninterpreted abstract functions, `ptr_to_int` and `int_to_ptr`. Round-trip casts, where a pointer is cast to an integer and then back to the pointer or vice versa, are treated as no-ops. Spoq2 models dereference by calling the `load` and `store` primitives provided by the abstract machine model in which out-of-bound accesses are checked. The `load` and `store` primitives are defined as a case analysis `if-then-else` statement that compares the pointer identifier `id` to every memory object in the abstract memory model and maps the `load` or `store` to the matched memory object. Spoq2 automatically applies transformation rules to simplify the representation by resolving it to a single branch matching

the pointer identifier. If there are low-level memory layout assumptions, Spoq2 leverages the `Layout` configuration to identify the memory object even in the presence of complex address calculations involving physical addresses.

For bitwise operations, Spoq2 additionally maps them to the corresponding attributes within the storage format provided as a pointer abstraction. Once mapped, these bitwise operations are eliminated from the generated Coq representation. For example, the Coq representation of lines 3-4 in Figure 9 is:

```
p := Ptr {id: pages, ofs: pte.ofs}.
```

This simplified expression in which `pte.ofs` directly refers to the `addr` attribute of the pointer abstraction yields a much simpler Z3 query, without any bitwise operations, utilizing faster linear integer arithmetic theory and avoiding costly bit-vector theory.

Spoq2 can support all five examples shown earlier using its pointer abstraction. For Examples 1 and 5, no pointer configuration is needed as neither involves bitwise operations or low-level memory layout assumptions. For Examples 2, 3, and 4, pointer configurations are helpful. For Example 2, it is useful to define where the metadata is stored. The configuration is:

```
Config TagPtr := {"addr":[0, 55), "tag":[55, 64)}.
```

For Example 3, it is useful to define the page table entry format, specifically where the one-bit valid, the one-bit read-only attribute, and the address are stored. The configuration is:

```
Config PTEPtr := {"valid": [0, 1), "rd_only": [1, 2),
                  "addr": [12, 64) }.
```

For Example 4, it is useful to state the low-level memory layout assumption about the two integer address constants `PHYS_OFFSET` and `PAGE_OFFSET`. The configuration is:

```
Config Layout:= [{"start" : PHYS_OFFSET, "object": pa},
                 {"start" : PAGE_OFFSET, "object": va}].
```

In practice, Spoq2 incrementally guides users to manually constructing pointer configurations according to verification needs, without requiring configurations for all pointers, or for configurations to be written all at once. It begins the verification attempt with an empty configuration. When the target property cannot be proved because certain pointer operations are not abstracted, Spoq2 reports overloaded pointers and asks the user to specify their configurations one by one. The user can learn each pointer's bit range from its definition in the source code and then write the configuration. §7 shows that the manual effort required for constructing pointer configurations is modest. Once a pointer is configured, all its occurrences in the system are automatically resolved. For soundness, Spoq2 trusts the user-specified memory layout, but not the storage format. If the user provides a configuration with an incorrect storage format, Spoq2 will skip this pointer abstraction because the format does not match the bit ranges used.

7 Evaluation

Our evaluation addresses three key questions: (1) Does Spoq2 scale to verifying security properties of unmodified, real-world system software? (§7.1); (2) How much manual effort does Spoq2 require compared to prior verification frameworks? (§7.2); and (3) How effective are Spoq2's proof optimizations, both individually and in combination? (§7.3)

Environment Setup. All experiments were conducted on a commodity workstation with a 24-core Intel i9-13900KF processor and 64 GB of RAM, running Ubuntu 22.04 with Linux kernel 6.5. Spoq2 relies on LLVM 14.0.0 and Z3 4.12.5, though other versions of these tools can also be used. This setup reflects a realistic developer environment rather than a specialized cluster.

Prototype Implementation. The Spoq2 prototype is implemented in 13.9K lines of code (LoC) in C++, consisting of an LLVM-to-Coq translator, an assembly-to-Coq translator, and an automated proof checker. The LLVM-to-Coq translator is 4K LoC; it follows the LLVM IR semantics of Spoq [56] and applies pointer abstractions to generate Coq representations. The assembly-to-Coq translator is 0.9K LoC; it translates assembly code into lists of assembly instructions as Coq representations. Our current implementation supports only ARMv8 assembly. The proof checker is 9K LoC; it constructs proof goals and incorporates all optimizations described in §5.

7.1 Verification Targets

We evaluated Spoq2 on four substantial system software codebases: (1) the Realm Management Monitor (RMM) v0.3.0, the core TCB of Arm CCA [3, 55]; (2) Trusted Firmware-A (TF-A) v2.13, Arm's EL3 firmware [58]; (3) the latest SeKVM hypervisor [53] based on Linux 6.1 LTS; and (4) Komodo [26], a TrustZone enclave monitor.

These targets span a broad spectrum of verification difficulty. Early RMM prototypes have been manually verified, but not the released TF-RMM. TF-A has never been verified. SeKVM prototypes have been previously verified semi-automatically, but not the Linux 6.1 version. Komodo has been automatically verified, though, unlike the other systems, it does not support multiprocessor execution. Together they demonstrate that Spoq2 can verify diverse, unmodified, real-world systems.

RMM. Arm CCA [55] is a state-of-the-art confidential computing architecture [2, 15, 38, 55], introducing the Realm abstraction to protect sensitive data in confidential virtual machines (VMs). Realms run in a separate Realm world isolated from Non-secure (NS) world. Its security relies on RMM,

which enforces isolation between Realms and untrusted system software running in NS world. Untrusted software interacts only through the RMM interface and has no direct access to Realm memory or CPU state.

We verified the unmodified open-source RMM v0.3.0 (TF-RMM) [3], the first such verification. By contrast, prior efforts were entirely manual: one verified only safety properties of an early prototype [27], while the other verified security properties of a different early prototype [55]. Like prior efforts, we omit proofs for functionality to allow resetting Realm IPA state, typically not used for CVMs. TF-RMM comprises 5.5K LoC in C and assembly, exposes 32 interfaces, and includes complex features such as dynamic memory management [55].

Properties Verified. We first verified eight safety properties as inductive invariants, which serve as lemmas for proving security properties. These inductive invariants guarantee that: (1) all `delegated` pages—unused pages that have been allocated to Realm world and therefore are not accessible from NS world—are zeroed, ensuring they contain no malicious content; (2) each Realm's mapped pages are always in the `data` state—each page has been delegated to Realm world and allocated for storing Realm content; and (3) each Realm's page table and CPU context stored in memory are isolated from other Realms.

Building on these invariants, we then verified RMM guarantees two information-flow security properties: confidentiality and integrity of Realm memory and CPU state. Confidentiality requires that a Realm's private state remain unobservable by other Realms or untrusted software. Integrity requires that a Realm never observe changes to its private state that it did not make itself. We formalized both properties as indistinguishability simulation relations on a composed system as discussed in §4. Confidentiality requires that two related RMM executions agree on public data (other Realms, RMM, and hypervisor state) while each Realm's private data (memory and CPU state) remains unconstrained. Integrity requires that two related RMM executions agree on each Realm's private data, while public data remains unconstrained. Spoq2 verifies these properties by proving that the simulation relations are inductively preserved. General purpose registers (GPRs) are explicitly declassified to allow legitimate parameter passing and return values.

TF-A. TF-A [58] is the open-source reference implementation of Arm Trusted Firmware for Armv8-A. TF-A's EL3 runtime services mediate access to Realm world, handling secure world switches between Realm and NS world, including saving and restoring Realm CPU state during world switches. Untrusted system software does not interact with TF-A directly.

While TF-A includes various boot stages (e.g., BL1, BL2) for hardware initialization, these components exit after booting. Our verification targets the persistent EL3 runtime firmware (BL31). We verified the unmodified v2.13 implementation, which comprises 10.0K LoC of C and assembly. This scope covers the 24 standardized service interfaces and the context-switching machinery itself.

Properties Verified. For TF-A, we verified the confidentiality of Realm CPU state: no service call leaks private Realm state to the untrusted NS world, which is a core security guarantee of TF-A [58]. We formalized this as an indistinguishability simulation relation on a composed system as discussed in §4, requiring that two TF-A executions agree on all public data while differing only in private data (Realm CPU state and CPU context stored in memory). GPRs are explicitly declassified to allow legitimate parameter passing and return values. Spoq2 verifies confidentiality by proving that the simulation relation is inductively preserved.

SeKVM. SeKVM is a Linux KVM/Arm hypervisor [18–21] that enforces per-VM isolation. While an earlier version of SeKVM was semi-automatically verified [53], we verified the latest Linux 6.1 LTS version, a production-level port that has been adopted as part of the DARPA V-SPELLS [25] program. The core of SeKVM consists of 4.0K LoC in C and assembly, implementing 19 hypercalls.

Properties Verified. We verified VM confidentiality and integrity, the core guarantees of SeKVM's VM isolation [53, 56]. Confidentiality requires that a VM's memory and CPU state remain hidden from the host and other VMs. Integrity requires that a VM only observe changes it induces itself. We formalized confidentiality as a simulation relation on a composed system requiring that two SeKVM executions agree on public data while keeping each VM's private data unconstrained. Integrity was formalized as an inductive invariant: a VM's secrets remain unchanged across all hypercalls, except for `GRANT`, which explicitly allows page sharing.

Komodo. Komodo [26] is a TrustZone enclave monitor. We targeted the version previously verified by Serval [64], consisting of 1.5K LoC, in order to compare against Serval, the only prior automated verification framework for security properties of system software known to us.

Properties Verified. We ported Komodo's Racket definitions from Serval into Coq, covering five security and 16 safety properties across 19 interfaces. Together, they enforce noninterference isolation of enclaves, ensuring that the OS neither influences enclave behavior nor learns from it.

Summary of Verification Results. We verified the unmodified codebases of all four target systems using Spoq2, without retrofitting for verification, demonstrating that Spoq2 scales to verifying security properties of real-world system software. §7.2 discusses manual proof effort. §7.3 quantifies verification performance. Spoq2 uncovered two previously unknown bugs in the SeKVM Linux 6.1 port, both confirmed by its developers. The first was an inconsistent use of vCPU identifiers intended to denote the same logical core. The second was a missing page-table lock acquisition: the `GRANT`

	RMM	TF–A	SeKVM	Komodo
Machine Configuration	1,450	1,389	938	875
Pointer Configuration	193	41	192	0
Loop Invariant, Ranking Function	82	80	26	10
Security Property Definition	156	74	165	510
Spoq2 Total (LoC)	2.0K	1.6K	1.3K	1.4K
Baseline (LoC)	10.3K	-	5.8K	1.6K
Manual Effort Reduction (%)	**80%**	-	**78%**	**13%**

Table 1. Manual efforts breakdown in lines of code (LoC).

hypercall accessed guest page tables without holding the required lock. Both bugs were introduced in the Linux 6.1 port, uncovered by Spoq2 during verification, and then fixed.

7.2 Manual Effort

Table 1 summarizes the manual effort required for each verification target. We also compare against prior verification work as baselines if they exist, with security-irrelevant efforts in each baseline excluded to provide a conservative comparison. For RMM, we compare against VIA [55], which verified an early RMM prototype roughly half the size of TF-RMM. For TF-A, there is no baseline, as it has not been previously verified. For SeKVM, we compare against Spoq [56], which verified an earlier SeKVM version for Linux 5.4 of similar code size. For Komodo, we compare against Serval, which verified the same Komodo codebase.

Spoq2 eliminates the need for manually constructing security proofs, which in prior frameworks required months of effort and thousands of lines of Coq [53, 56]. For example, VIA required 3.4K LoC for manual security proofs for an RMM prototype, and Spoq required 3.0K LoC for manual security proofs for SeKVM. In contrast, Spoq2 requires only 156 and 165 lines of security property definitions, respectively. The security properties are written as inductive invariants in Coq representation, and are generally small. Notably, revising these invariants is considered the most laborious aspect of our evaluation and the Spoq2 workflow, which requires developers to have a deep understanding of systems and formal methods. Security properties such as noninterference have been manually proven using inductive invariants in prior verified systems [26, 33, 53, 55, 64, 76, 79, 81]. These existing proofs give users examples of how such inductive invariants can be structured, making the definition process easier. For our verification targets, we defined the inductive invariants for each system's security properties within weeks.

Most of the manual effort required for using Spoq2 involved specifying machine configurations include defining registers, memory objects, and `load`/`store` primitives. Only modest manual effort was required for specifying pointer configurations for Spoq2 to verify RMM, TF-A, and SeKVM, demonstrating the practicality and effectiveness of pointer abstractions. The configurations primarily cover pointers for page table entries, and also required layouts to express low-level memory assumptions. No such effort was required for Komodo, which is pointer-free. Only tens of lines of Coq were required to specify loop invariants and ranking functions, which were generally simple and easy to prove. For example, SeKVM has 19 loops which all share the same pattern `for(i=0;i<=ITER;i++)`, for which simple loop invariants are sufficient. Such simplicity further eases the difficulty for users to define the ranking functions and for Spoq2 to check them. For SeKVM, we simply define ranking functions as `f(i)=ITER-i`, which was sufficient to automatically complete loop termination proofs.

	RMM	TF–A	SeKVM	Komodo
Generate Coq Representation	38m37s	44m08s	8m04s	0m10s
Prove Safety	42m18s	N/A	3m15s	0m14s
Prove Info-Flow Security	174m06s	39m23s	36m47s	0m16s
Total: End-to-end Time	**255m01s**	**83m31s**	**48m06s**	**0m40s**

Table 2. End-to-end Spoq2 performance breakdown.

Overall, Spoq2 reduces manual effort by 80% for RMM, despite targeting a codebase roughly twice the size of the RMM prototype verified with VIA, and by 78% for SeKVM. For Komodo, where the baseline is also a highly automated framework, Spoq2 still saves 13%. For TF-A, the total manual effort is only 1.6K lines of code.

7.3 Verification Performance

Table 2 presents the end-to-end verification performance of Spoq2 with all optimizations enabled, including state simplification (§5.1), path pruning (§5.2), clause pruning (§5.3), and Z3 query caching (§5.4). Total verification times for all four verification targets were no more than a few hours, ranging from less than a minute for Komodo to more than four hours for RMM. The largest portion of time was for proving information-flow security properties.

To the best of our knowledge, Serval [64] is the only existing framework that can automate security verification of system software. On our machine, Serval takes 280 seconds to verify Komodo's security properties, whereas Spoq2 achieves the same verification 7× faster. However, Serval lacks support for concurrency and complex pointer reasoning, which prevents it from handling unmodified systems such as RMM, TF-A, and SeKVM.

To quantify the contribution of each optimization, we conducted an ablation study by disabling them individually (Figure 10). In the figure, a value of 1.0 on the vertical axis denotes the baseline with all optimizations disabled, while the first bar for each system shows performance with all optimizations enabled; lower is better. Pointer abstraction is always enabled, since verification cannot complete without it when reasoning about low-level bitwise pointer operations. Disabling certain optimizations—e.g., state simplification in RMM or path pruning in TF-A—caused some proof tasks

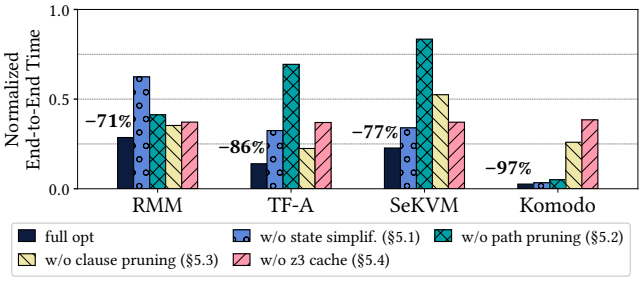


Figure 10. Ablation study of Spoq2 optimizations.

to fail to terminate within a two-day timeout, which we excluded when calculating speedup and measuring unoptimized verification time. Overall, enabling all optimizations achieved an 8× geometric mean speedup, while disabling any single optimization led to measurable slowdowns in most systems, and in some cases, non-termination.

For RMM, with all optimizations enabled, Spoq2 reduced 73% of proof goals (26,477 total) and 71% of end-to-end verification time (879 minutes total). State simplification was most effective due to RMM's large machine state.

For TF-A, with all optimizations enabled, Spoq2 reduced 85% of proof goals (61,749 total) and 86% of verification time (592 minutes total). Because many branches in TF-A are irrelevant to Realm CPU context, path pruning was the most impactful optimization.

For SeKVM, with all optimizations enabled, Spoq2 reduced 82% of proof goals (11,806 total) and 77% of verification time (213 minutes total). Most branch conditions—72% of 598 branches—are independent of private data, allowing path pruning and clause pruning to stand out as the most impactful optimizations.

For Komodo, with all optimizations enabled, Spoq2 reduced 92% of proof goals (597 total) and 97% of verification time (22 minutes total). Z3 query caching was most effective, yielding a 13.7× speedup, while clause pruning eliminated over 71% of equality clauses before reaching Z3.

8 Related Work

Systems Verification. seL4 [40] developed the first machine-checkable functional correctness proof of an OS kernel, which required a significant manual effort of 12 person-years. Pointers to local variables were disallowed by the simplified C semantics. Many later verified systems [1, 11, 16, 31–33, 39, 41, 53, 55, 59, 79] used ClightGen [49] and conducted proofs manually. However, ClightGen only supports a subset of the C language, so significant manual effort was required to retrofit real-world system code before verification [17, 53, 56]. Spoq [56] leveraged Clang and LLVM to translate C code into Coq to provide broad support for real-world C system code. Spoq2 leverages Spoq's LLVM-to-Coq translator to enable verification of C system code without retrofitting. CN [68] introduced a separation logic and refinement type system to verify a buddy allocator from Android's pKVM (364 LoC). Unlike Spoq2, none of these systems enables highly automated verification of security properties for unmodified system software.

Automated Verification. Automated verification has been applied in a range of domains, including file systems [73, 87], device drivers [36], kernels [64, 65, 74], and quantum systems [77, 78, 80]. These systems demonstrated the promise of automation, but typically relied on strong restrictions on the input code (e.g., no unbounded loops [65]) or worked within narrow domains. Serval [64] used symbolic execution to verify noninterference of kernels, but its lack of support for pointers and concurrency prevents it from handling realistic system code. Spoq reduces proof effort for functional correctness, but provides no automation for security proofs. TPot [9] employed KLEE [8] to check functional correctness of C components, but it did not address concurrency or security properties. RefinedC [72] combined refinement types with Hoare-style reasoning, but required detailed programmer-provided specifications. VeriFast [36] provided automated verification for C and Java using separation logic, but demanded extensive annotations, limiting scalability to large systems. AutoCorres [30] automatically derived specifications from C, but supported only a restricted subset of the language and assumed a simplistic machine model. Verus [46, 47] leveraged Rust's ownership types to support modular verification, but still required intensive manual annotations and was limited to Rust. In contrast, Spoq2 supports realistic C system code used in real system software, including pointers, concurrency, and low-level hardware interactions. It highly automates security proofs for unmodified, large-scale system components such as RMM, TF-A, and SeKVM—a level of automation not achieved by prior tools.

Pointer Verification. The concrete memory model [30, 40, 68] unsoundly treats pointers as integers [48]. CompCert [49] supports only simple offset adjustments, not bit operations. The provenance memory model [61] enriches pointers with provenance but discards it when casting to integers. RefinedVIP [48] introduces refinement types to verify `tag/untag` operations for a 25-line tagging library, but does not scale to real systems. In contrast, Spoq2 supports low-level bitwise pointer operations in real systems, where prior models either fail or require extensive manual intervention.

9 Conclusions and Future Work

Spoq2 is the first framework to highly automate security verification of unmodified, real system software. Spoq2 decomposes security proofs into independent transition-level proof goals, which can then be discharged automatically by Z3. To make this scale, Spoq2 combines several key techniques—fine-grained COI analysis, pointer abstraction, and query

caching—that dramatically reduce both the number and complexity of proof goals. Our results demonstrate the scalability of Spoq2 by verifying the security properties of unmodified, real systems with minimal manual effort.

Spoq2 currently relies on users to provide inductive invariants, pointer configurations, loop invariants, and ranking functions. Developing a sound toolchain that can automatically construct these configurations and infer system-level loop invariants is an area of future work.

Acknowledgments

We thank our shepherd, Stephen Freund, and the anonymous reviewers for valuable feedback that helped improve this paper. Raphael Sofaer provided helpful feedback on earlier drafts of this paper. This work was supported in part by an NSF CAREER Award CCF-2239484, an Amazon Research Award, a VMware Systems Research Award, DARPA contract N66001-21-C-4018, and NSF grants CCF-2124080, CNS-2052947, and CNS-2247370. Ronghui Gu is a co-founder of and has an equity interest in CertiK.

A Artifact Appendix

A.1 Abstract

This artifact includes the Spoq2 framework and the system software that we have evaluated.

A.2 Artifact check-list (meta-information)

- **Run-time environment:** Linux
- **Hardware:** x86
- **How much disk space required (approximately)?:** 8 GB
- **How much time is needed to prepare workflow (approximately)?:** 5–10 minutes to download and import the Docker image
- **How much time is needed to complete experiments (approximately)?:** 8–16 hours, depending on CPU performance
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT License
- **Archived (provide DOI)?:** Yes

A.3 Description

A.3.1 How to access. The artifact can be downloaded from the link: https://doi.org/10.5281/zenodo.17861493.

A.3.2 Hardware dependencies. A standard x86 machine.

A.3.3 Software dependencies. Docker.

A.4 Installation

The Docker image is provided in a pre-configured format, obviating the need for any installation.

A.5 Experiment workflow

See `README.md` in the top directory of the Zenodo repository.

A.6 Evaluation and expected results

We provide the necessary dataset and scripts to reproduce the evaluation results presented in Section 7. Specifically, the results shown in Tables 1 and 2, and Figure 10 can be reproduced using the provided resources. For further details, please refer to the accompanying website or archive.

References

[1] Andrew W. Appel. 2011. Verified Software Toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software* (Saarbrücken, Germany) *(ESOP '11/ETAPS '11)*. 1–17.

[2] ARM Ltd. 2009. ARM Security Technology: Building a Secure System using TrustZone Technology. https://documentation-service.arm.com/static/5f212796500e883ab8e74531. Accessed 22 Dec. 2025.

[3] ARM Ltd. 2023. TF-RMM. https://www.trustedfirmware.org/projects/tf-rmm/. Accessed 22 Dec. 2025.

[4] D Elliot Bell and Leonard J LaPadula. 1973. *Secure Computer Systems: Mathematical Foundations*. Technical Report MTR-2547-VOL-1. MITRE Corp.

[5] Kenneth J Biba. 1977. *Integrity Considerations for Secure Computer Systems*. Technical Report MTR-3153-REV-1. MITRE Corp.

[6] Aaron R. Bradley and Zohar Manna. 2007. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer. doi:10.1007/978-3-540-74113-8

[7] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. 2017. *Hardware and Software Support for Virtualization*. Morgan and Claypool Publishers. doi:10.2200/S00754ED1V01Y201701CAC038

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation* (San Diego, CA USA) *(OSDI '08)*. 209–224.

[9] Can Cebeci, Yonghao Zou, Diyu Zhou, George Candea, and Clément Pit-Claudel. 2024. Practical Verification of System-Software Components Written in Standard C. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX USA) *(SOSP '24)*. 455–472. doi:10.1145/3694715.3695980

[10] Dongwei Chen, Dong Tong, Chun Yang, Jiangfang Yi, and Xu Cheng. 2023. FlexPointer: Fast Address Translation Based on Range TLB and Tagged Pointers. *ACM Transactions on Architecture and Code Optimization* 20, 2, Article 30 (March 2023), 24 pages. doi:10.1145/3579854

[11] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. 2016. Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA USA) *(PLDI '16)*. 431–447. doi:10.1145/2908080.2908101

[12] David D Clark and David R Wilson. 1987. A Comparison of Commercial and Military Computer Security Policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy* (Oakland, CA USA) *(IEEE S&P '87)*. 184–194.

[13] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 2000. *Model Checking*. MIT Press.

[14] Michael R. Clarkson and Fred B. Schneider. 2008. Hyperproperties. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium* (Pittsburgh, PA USA) *(CSF '08)*. 51–65. doi:10.1109/CSF.2008.7

[15] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. Cryptology ePrint Archive, Paper 2016/086. https://eprint.iacr.org/2016/086

[16] David Costanzo, Zhong Shao, and Ronghui Gu. 2016. End-to-End Verification of Information-Flow Security for C and Assembly Programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA USA) *(PLDI '16)*. 648–664. doi:10.1145/2908080.2908100

[17] Zhenyang Dai, Shuang Liu, Vilhelm Sjoberg, Xupeng Li, Yu Chen, Wenhao Wang, Yuekai Jia, Sean Noble Anderson, Laila Elbeheiry, Shubham Sondhi, et al. 2024. Verifying Rust Implementation of Page Tables in a Software Enclave Hypervisor. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1218–1232.

[18] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, and Jason Nieh. 2018. ARM Virtualization: Performance and Architectural Implications. *ACM SIGOPS Operating Systems Review* 52, 1 (Aug. 2018), 45–56. doi:10.1145/3273982.3273987

[19] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos. 2016. ARM Virtualization: Performance and Architectural Implications. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. 304–316. doi:10.1109/ISCA.2016.35

[20] Christoffer Dall, Shih-Wei Li, and Jason Nieh. 2017. Optimizing the Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 2017 USENIX Annual Technical Conference* (Santa Clara, CA USA) *(USENIX ATC '17)*. 221–233.

[21] Christoffer Dall and Jason Nieh. 2013. Supporting KVM on the ARM Architecture. *LWN Weekly Edition* (July 2013), 18–22.

[22] Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, and Oliver Schwarz. 2013. Formal Verification of Information Flow Security for a Simple ARM-Based Separation Kernel. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security* (Berlin, Germany) *(CCS '13)*. 223–234. doi:10.1145/2508859.2516702

[23] Ádám Darvas, Reiner Hähnle, and David Sands. 2005. A Theorem Proving Approach to Analysis of Secure Information Flow. In *Proceedings of the 2nd International Conference on Security in Pervasive Computing* (Boppard, Germany) *(SPC'05)*. 193–209. doi:10.1007/978-3-540-32004-3_20

[24] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340.

[25] Defense Advanced Research Projects Agency. 2025. V-SPELLS: Verified Security and Performance Enhancement of Large Legacy Software. https://www.darpa.mil/research/programs/verified-security-and-performance-enhancement-of-large-legacy-software. Accessed 1 Aug. 2025.

[26] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. In *Proceedings of the ACM SIGOPS 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. 287–305. doi:10.1145/3132747.3132782

[27] Anthony C. J. Fox, Gareth Stockwell, Shale Xiong, Hanno Becker, Dominic P. Mulligan, Gustavo Petri, and Nathan Chong. 2023. A Verification Methodology for the Arm Confidential Computing Architecture: From a Secure Specification to Safe Implementations. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1, Article 88 (April 2023), 30 pages. doi:10.1145/3586040

[28] Joseph A Goguen and J Meseguer. 1982. Security Policies and Security Models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy* (Oakland, CA USA) *(IEEE S&P '82)*. 11–20.

[29] Amogha Udupa Shankaranarayana Gopal, Raveendra Soori, Michael Ferdman, and Dongyoon Lee. 2023. TailCheck: A Lightweight Heap Overflow Detection Mechanism with Page Protection and Tagged Pointers. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation* (Boston, MA USA) *(OSDI '23)*. 535–552.

[30] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. 2014. Don't Sweat the Small Stuff: Formal Verification of C Code Without the Pain. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. 429–439. doi:10.1145/2594291.2594296

[31] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep specifications and certified abstraction layers. *ACM SIGPLAN Notices* 50, 1 (2015), 595–608.

[32] Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan Newman Wu, Vilhelm Sjöberg, and David Costanzo. 2019.

Building certified concurrent OS kernels. *Commun. ACM* 62, 10 (2019), 89–99.

[33] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (Savannah, GA USA) *(OSDI '16)*. 653–669.

[34] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified Concurrent Abstraction Layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA USA) *(PLDI '18)*. 646–661. doi:10.1145/3192366.3192381

[35] Constance L. Heitmeyer, Myla Archer, Elizabeth I. Leonard, and John McLean. 2006. Formal Specification and Verification of Data Separation in a Separation Kernel for an Embedded System. In *Proceedings of the 13th ACM Conference on Computer and Communications Security* (Alexandria, VA USA) *(CCS '06)*. 346–355. doi:10.1145/1180405.1180448

[36] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *Proceedings of the 3rd International Conference on NASA Formal Methods* (Pasadena, CA USA) *(NFM '11)*. 41–55. doi:10.1007/978-3-642-20398-5_4

[37] Fuqi Jia, Rui Han, Pei Huang, Minghao Liu, Feifei Ma, and Jian Zhang. 2023. Improving Bit-Blasting for Nonlinear Integer Constraints. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA USA) *(ISSTA '23)*. 14–25. doi:10.1145/3597926.3598034

[38] David Kaplan, Jeremy Powell, and Tom Woller. 2020. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. White Paper. Advanced Micro Devices, Inc.

[39] Jieung Kim, Vilhelm Sjöberg, Ronghui Gu, and Zhong Shao. 2017. Safety and Liveness of MCS Lock—Layer by Layer. In *Programming Languages and Systems*, Bor-Yuh Evan Chang (Ed.). Springer International Publishing, 273–297. doi:10.1007/978-3-319-71237-6_14

[40] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, MT USA) *(SOSP '09)*. 207–220. doi:10.1145/1629575.1629596

[41] Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Cascais, Portugal) *(CPP '19)*. 234–248. doi:10.1145/3293880.3294106

[42] Orna Kupferman and Moshe Y. Vardi. 1999. Model Checking of Safety Properties. In *Proceedings of the 11th International Conference on Computer Aided Verification* (Trento, Italy) *(CAV '99)*. Springer-Verlag, 172–183.

[43] Leslie Lamport. 1977. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering* 3, 2 (March 1977), 125–143. doi:10.1109/TSE.1977.229904

[44] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* 28, 9 (Sept. 1979), 690–691. doi:10.1109/TC.1979.1675439

[45] Chris Lattner. 2008. LLVM and Clang: Next Generation Compiler Technology. In *Proceedings of the BSDCan 2008* (Ottawa, ON Canada), Vol. 5. 1–20.

[46] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A Practical Foundation for Systems Verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX USA) *(SOSP '24)*. 438–454. doi:10.1145/3694715.3695952

[47] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1, Article 85 (April 2023), 30 pages. doi:10.1145/3586037

[48] Rodolphe Lepigre, Michael Sammler, Kayvan Memarian, Robbert Krebbers, Derek Dreyer, and Peter Sewell. 2022. VIP: Verifying Real-World C Idioms with Integer-Pointer Casts. *Proceedings of the ACM on Programming Languages* 6, POPL, Article 20 (Jan. 2022), 32 pages. doi:10.1145/3498681

[49] Xavier Leroy, Andrew W Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert Memory Model, Version 2*. Ph. D. Dissertation. Inria.

[50] Peng Li and Steve Zdancewic. 2005. Downgrading Policies and Relaxed Noninterference. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, CA USA) *(POPL '05)*. 158–170. doi:10.1145/1040305.1040319

[51] Shih-Wei Li, John S. Koh, and Jason Nieh. 2019. Protecting Cloud Virtual Machines from Commodity Hypervisor and Host Operating System Exploits. In *Proceedings of the 28th USENIX Security Symposium* (Santa Clara, CA USA) *(USENIX Security '19)*. 1357–1374.

[52] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2020. *Microverification of the Linux KVM Hypervisor: Proving VM Confidentiality and Integrity*. Technical Report CUCS-003-20. Department of Computer Science, Columbia University.

[53] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. A Secure and Formally Verified Linux KVM Hypervisor. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy* (San Francisco, CA USA) *(IEEE S&P '21)*. 1782–1799.

[54] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In *Proceedings of the 30th USENIX Security Symposium* (Vancouver, BC Canada) *(USENIX Security '21)*. 3953–3970.

[55] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. 2022. Design and Verification of the Arm Confidential Compute Architecture. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation* (Carlsbad, CA USA) *(OSDI '22)*. USA, 465–484.

[56] Xupeng Li, Xuheng Li, Wei Qiang, Ronghui Gu, and Jason Nieh. 2023. Spoq: Scaling Machine-Checkable Systems Verification in Coq. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation* (Boston, MA USA) *(OSDI '23)*. 851–869.

[57] Xuheng Li, Xupeng Li, Wei Qiang, Ganxiang Yang, Yi Rong, Ronghui Gu, and Jason Nieh. 2025. *Specification Projections: Scaling Formal Verification of Security Properties for Unmodified System Software*. Technical Report CUCS-003-25. Department of Computer Science, Columbia University.

[58] Arm Limited and Contributors. 2023. Trusted Firmware-A. https://trustedfirmware-a.readthedocs.io/en/latest/. Accessed 22 Dec. 2025.

[59] Mengqi Liu, Lionel Rieg, Zhong Shao, Ronghui Gu, David Costanzo, Jung-Eun Kim, and Man-Ki Yoon. 2019. Virtual Timeline: A Formal Abstraction for Verifying Preemptive Schedulers with Temporal Isolation. *Proceedings of the ACM on Programming Languages* 4, POPL, Article 20 (Dec. 2019), 31 pages. doi:10.1145/3371088

[60] Daryl McCullough. 1987. Specifications for Multi-Level Security and a Hook-Up Property. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy* (Oakland, CA USA) *(IEEE S&P '87)*. 161–166.

[61] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. 2019. Exploring C Semantics and Pointer Provenance. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 67 (Jan. 2019), 32 pages.

doi:10.1145/3290380

[62] J. Donald Monk. 1976. *Mathematical Logic*. Springer. doi:10.1007/978-1-4684-9452-5

[63] Roger M. Needham. 1993. Denial of Service. In *Proceedings of the 1st ACM Conference on Computer and Communications Security* (Fairfax, VA USA) *(CCS '93)*. 151–153. doi:10.1145/168588.168607

[64] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *Proceedings of the ACM SIGOPS 27th Symposium on Operating Systems Principles* (Huntsville, ON Canada) *(SOSP '19)*. 225–242. doi:10.1145/3341301.3359641

[65] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. 252–269. doi:10.1145/3132747.3132748

[66] Aina Niemetz, Mathias Preiner, and Yoni Zohar. 2024. Scalable Bit-Blasting with Abstractions. In *Computer Aided Verification: 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part I* (Montreal, QC Canada). Springer-Verlag, 178–200. doi:10.1007/978-3-031-65627-9_9

[67] François Pottier and Vincent Simonet. 2002. Information flow inference for ML. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, OR USA) *(POPL '02)*. 319–330. doi:10.1145/503272.503302

[68] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. 2023. CN: Verifying Systems C Code with Separation-Logic Refinement Types. *Proc. ACM Program. Lang.* 7, POPL, Article 1 (Jan. 2023), 32 pages. doi:10.1145/3571194

[69] Cindy Rubio-González and Ben Liblit. 2011. Defective Error/Pointer Interactions in the Linux Kernel. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, ON Canada) *(ISSTA '11)*. 111–121. doi:10.1145/2001420.2001434

[70] John Rushby. 1992. *Noninterference, Transitivity, and Channel-Control Security Policies*. Technical Report CSL-92-02. Computer Science Laboratory, SRI International.

[71] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. 2020. CLN2INV: Learning Loop Invariants with Continuous Logic Networks. In *Proceedings of the 8th International Conference on Learning Representations (ICLR 2020)*.

[72] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the Foundational Cerification of C Code with Refined Ownership Types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*. 158–174. doi:10.1145/3453483.3454036

[73] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (Savannah, GA USA) *(OSDI '16)*. 1–16.

[74] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. 2018. Nickel: A Framework for Design and Verification of Information Flow Control Systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation* (Carlsbad, CA USA) *(OSDI '18)*. 287–305.

[75] Geoffrey Smith. 2007. Principles of Secure Information Flow Analysis. In *Malware Detection*, Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff Wang (Eds.). Springer, 291–307. doi:10.1007/978-0-387-44599-1_13

[76] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A. Seshia. 2017. A Formal Foundation for Secure Remote Execution of Enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, TX USA) *(CCS '17)*. 2435–2450. doi:10.1145/3133956.3134098

[77] Runzhou Tao, Yunong Shi, Jianan Yao, John Hui, Frederic T Chong, and Ronghui Gu. 2021. Gleipnir: toward practical error analysis for Quantum programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 48–64.

[78] Runzhou Tao, Yunong Shi, Jianan Yao, Xupeng Li, Ali Javadi-Abhari, Andrew W Cross, Frederic T Chong, and Ronghui Gu. 2022. Giallar: push-button verification for the qiskit Quantum compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. 641–656.

[79] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. 2021. Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. 866–881. doi:10.1145/3477132.3483560

[80] Runzhou Tao, Hongzheng Zhu, Jason Nieh, Jianan Yao, and Ronghui Gu. 2025. Quantum Virtual Machines. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 411–428.

[81] Ganxiang Yang, Chenyang Liu, Zhen Huang, Guoxing Chen, Hongfei Fu, Yuanyuan Zhang, and Haojin Zhu. 2025. A Formal Approach to Multi-Layered Privileges for Enclaves. In *Proceedings of the Network and Distributed System Security Symposium 2025* (San Diego, CA USA) *(NDSS '25)*. doi:10.14722/ndss.2025.241301

[82] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. 2020. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 106–120.

[83] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. 2022. DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation* (Carlsbad, CA USA) *(OSDI '22)*. 485–501.

[84] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. 2024. Mostly Automated Verification of Liveness Properties for Distributed Protocols with Ranking Functions. *Proceedings of the ACM on Programming Languages* 8, POPL, Article 35 (Jan. 2024), 32 pages. doi:10.1145/3632877

[85] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. 2021. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation* (Santa Clara, CA USA) *(OSDI '21)*. 405–421.

[86] Steve Zdancewic and Andrew C Myers. 2003. Observational Determinism for Concurrent Program Security. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop* (Pacific Grove, CA USA) *(CSFW '16)*. 29–43.

[87] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. 2019. Using concurrent relational logic with helpers for verifying the AtomFS file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 2019)*. ACM, 259–274.