Practical Verification of System-Software Components Written in Standard C

Can Cebeci, Yonghao Zou, Diyu Zhou, George Candea, Clément Pit-Claudel
EPFL, Switzerland

Abstract

Systems code is challenging to verify, because it uses constructs (like raw pointers, pointer arithmetic, and bit twiddling) that are hard for tools to reason about. Existing approaches either sacrifice programmer friendliness, by demanding significant manual effort and verification expertise, or generality, by restricting the programming language or requiring that the code adapt to the verification tool.

We identify a new point in the design space of verifiers that enables a different set of trade-offs, which prove practical in the context of verifying critical system components. We use several novel techniques to develop the TPot verification framework, targeted specifically at systems code written in C. With TPot, developers verify critical components they implement in standard, unrestricted C, using a C-based language to write "proof-oriented tests" (POTs) that specify the desired behavior. TPot then runs the POTs to prove correctness. We evaluate TPot on 6 different systems-code bases, and show that it can verify them at the same level as 4 state-of-the-art verifiers, while consistently reducing the annotation burden, ranging up to more than 3×. TPot does not require these code bases to be adapted for verification and achieves verification times compatible with typical continuous-integration workflows.

TPot is open-source and freely available at https://github.com/dslab-epfl/tpot.

1 Introduction

Formally proving that "correct" code is indeed correct entails significant developer effort. Doing this for systems code written in C is even harder, due to low-level programming idioms that are difficult to reason about formally and automatically. This effort impacts developers' short-term productivity and so deters them from using formal methods to demonstrate that code they already believe to be correct is indeed so. Not surprisingly, there are but a few examples of real systems whose entire C implementation has been formally proven to be correct. Yet, in practice, only some parts of the system are truly critical, so it is pragmatic and sensible to limit one's ambition of proving correctness formally for just these components. For example, a cryptographic library is critical to every networked system, so the effort invested in formally proving the correctness of HACL* [73] is justified. In our work, we therefore aim to prove the correctness of critical **components of systems**, not entire systems.

One of the greatest impediments to proving code formally is the human assistance required by verification tools [7, 20, 32, 38, 51]. Provers like Coq [5], Isabelle [52], or Lean [22] are interactive, meaning that the developer iteratively provides instructions to the verifier for how to piece together the correctness proof. This can take several person-years for reasonably sized code bases [36, 73]. Our goal is to enable developers to **prove the correctness of correct code in one go**, without interaction. Other types of verifiers rely on programmers to annotate the code with formal comments that describe the logic and its alignment with the specifications to be proven [34, 39, 61]. These typically take the form of pre-/post-conditions for public APIs and all internal methods plus various proof hints, like invariants. They help the verifier understand the code and cross reasoning gaps it couldn't on its own, around function calls, recursion, and loops. Annotations are in some sense a way of constructing a proof. While we cannot eliminate all need for annotations, we do aim to **reduce the annotation burden**.

One way to reduce this burden is to forgo the use of C and employ languages better suited for automated reasoning. Unfortunately this presents three problems: First, it requires connecting the code to the verifier, either by rewriting it from scratch [15, 16, 33], by building a model of it, or by importing it into a verifier. For example, Perennial [11, 14] supports Go code by translating it to Coq for verification. This, in turn, requires the practitioner to learn new languages merely to prove the correctness of code that she already believes to be correct. Second, it introduces new risks, because the rewritten software or model may deviate from the original in subtle ways not captured by the specification, and linking errors at verified/unverified boundaries are more likely if the verified code is written in a language different from the unverified code [26]. Third, language choice impacts performance, and that often forces mutual exclusion between what is verified and what runs fast. Often, the result is that there is a verified version and a production version meant for actual

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOSP '24, November 4–6, 2024, Austin, TX, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1251-7/24/11…$15.00
https://doi.org/10.1145/3694715.3695980

use, with the latter being fast enough but not verified [33]. Our goal is to **not require system programmers to learn verification-specific languages**.

A compromise approach is to stick with C but restrict how it is used and/or restrict the design itself. First, using subsets of C [4, 20] avoids certain constructs that are verification-hostile, such as untyped pointers, pointer arithmetic, or explicit memory management. For example, Vigor [70] prohibits a number of C constructs from the code that is automatically verified. We want developers to be able to **use all of standard C**. Second, systems code often needs to be designed and implemented differently if it is to be verified automatically. To verify Komodo [25], Serval [49] replaced all pointers with indices into global arrays and removed virtual-to-physical address translation, to avoid the complexity of page walks. CertiKOS [28] likewise removed support for memory reclamation. The Hyperkernel [48] required system call implementation to avoid unbounded loops. Beyond the initial coding, adding a new property to be proven can also impose a rethink of the design and implementation [33]. We want developers to be able to **write and evolve code without having to tailor it to the verifier.**

A new set of trade-offs is afforded by modern development environments in which system software is developed in C, the code is thoroughly tested using continuous integration (CI), and only the critical components are formally verified. We identify this new design point and develop several novel techniques with which we augment the KLEE symbolic executor [8] to obtain a verification framework we call TPot.

To reduce programmer effort, TPot extends automated verification from the *function level* to the *component level*. Existing solver-aided verifiers [34, 39, 49, 57] are designed, or encourage their users, to keep the reasoning gaps that an SMT solver needs to cross as small as possible, aiming to ensure predictable response times for verification [46]. The common wisdom is that widening these gaps gives rise to solver queries that do not terminate ("solver explosion"). Therefore, systems are verified one function at a time and data structure predicates are manually hidden or revealed, minimizing the context exposed to the solver. We show how, *in the context of systems code*, it is possible to achieve more solver-aided automation while avoiding common causes of solver explosion. We keep the logic powerful enough to reason about systems code, but the conventions of our target domain allow us to take logic shortcuts that make verification significantly more predictable.

Scaling to the verification of full systems that are larger than a typical component is out of scope for TPot. While we mitigate *solver explosion*, we inherit the *path explosion* problem from traditional symbolic execution. The latter does not make component-level verification intractable, but presents a scalability bottleneck. Moreover, we target only sequential systems code for now, and so TPot does not support multi-threaded code.

We make the following three contributions:

① **A bespoke SMT encoding** of properties over system components, tuned for systems code. Existing verifiers use typed memory models that lead them to either require programmers to explicitly write lemmas (e.g., for type casting [34]), or they outright do not support untyped pointers, pointer arithmetic, integer↔pointer type casts, bit-twiddling, etc. (e.g., Serval [49] does not support casting integers to pointers, which puts features like page-table walks out of reach). We extend KLEE's byte-level memory model with a symbolic heap representation where addresses are integers (instead of bitvectors), and that supports symbolic base addresses, symbolic object sizes, and lazy object materialization. This allows TPot to use SMT queries to efficiently resolve symbolic addresses and objects, thus fully automating most reasoning about pointers and bit-level operations. As another example, we observe that systems code virtually never operates on infinite, truly recursive data structures (e.g., linked lists typically track free elements that reside in an array of pre-allocated resources, not arbitrary pieces of memory). We leverage this to encode operations on such data structures into SMT queries that do not involve quantifiers.

② **Component-level specifications** that allow programmers to omit the specification of functions internal to the component while keeping verification automated. Solver-aided verifiers [34, 39, 49, 57] require programmers to provide pre- and post-conditions for each internal function of a component to verify properties at the component's interface. Doing so represents a major fraction of the specification burden, because formalizing the exact context that each internal function is expected to be called in is hard. This is unlike describing externally observable operations at the component's API level, which is easy for a developer used to writing component-level test cases and natural-language specifications. TPot's restricted, stable SMT encoding enables aggressive inlining during verification while avoiding common causes of solver explosion: it predictably increases solver time, but within the bounds afforded by a CI workflow.

③ **A C-based specification language** that is sufficiently expressive for systems code but still permits automated verification. As mentioned, the behavior of *systems* components can typically be specified using more restricted logics than those provided by existing general-purpose verifiers. We therefore extend C with 8 specification primitives for describing systems-code behavior, while deliberately excluding recursive predicates and general universal quantification. The resulting specification language is intuitive for C programmers, while allowing for SMT encodings that avoid common causes of solver explosion.

We evaluate TPot on 6 different system-code bases, including a pKVM heap allocator [44], the Vigor resource manager [70], a page table implementation [72], a Linux device driver [35], and the Komodo security monitor [25]. TPot verifies them with the same strength of guarantees as

4 state-of-the-art verifiers, but without needing to adapt the implementations to TPot. TPot reduces the annotation burden, up to more than 3× compared to the 4 verifiers, while flattening the learning curve and achieving typical verification times of <1 hour, compatible with CI workflows. With TPot, developers use sxtandard C to write their code, and an extension of C to write "proof-oriented tests" (POTs) that specify the desired behavior of critical components. TPot then runs the POTs to prove correctness. We envision POTs evolving out of standard tests, so verification follows naturally once the developer is confident in the quality of the code.

We now present related work (§2), our approach (§3), design (§4), evaluation (§5), limitations (§6), and conclusion (§7).

2 Related Work

There are many examples of using formal methods to verify correctness of systems code, including cryptographic libraries [6, 24], transaction libraries [14], key-value stores [7, 30], file systems [13, 15, 16, 33, 74], journaling [12], synchronization primitives [53], and even operating system kernels [28, 36, 48] and hypervisors [38, 47, 66, 67].

Decades of research have led to a rich design space involving trade-offs along multiple dimensions: proof effort (i.e., the degree of proof automation), learning curve (i.e., the ease of learning and applying the verification approach), expressiveness (i.e., richness of properties that can be verified), size of the TCB, verification performance (i.e., how long verification takes), and generality (i.e., how broadly applicable the verification tool is). This section identifies the unique and useful point that TPot occupies in this design space by contrasting it with prior work. We structure our discussion around the degree of proof automation.

2.1 Interactive Verification

Interactive theorem provers. At one end of the spectrum are general-purpose *interactive theorem provers* (e.g., Coq [5] and Isabelle [52]). They involve the lowest degree of built-in proof automation. With this approach, developers interact with the theorem prover to write proofs as detailed sequences of steps, separately from the code.

One of the most notable achievements with this approach is the seL4 project [36], which is the first to demonstrate the possibility of proving the functional correctness of a general-purpose operating system kernel. The seL4 kernel consists of ~8,700 lines of C code and is verified using the Isabelle [52] theorem prover. CertiKOS [27, 28] showed the feasibility of proving the functional correctness of a concurrent operating system kernel.

A significant advantage of interactive theorem provers is flexibility: they enable users to reason about arbitrary languages, semantics, and program properties (e.g., functional correctness, termination) and hyperproperties (e.g., noninterference, availability). Another advantage shared by many interactive theorem provers is a minimal TCB, with the main tool producing proofs that can be independently checked by a much smaller kernel of hundreds to thousands of LOC.

Disadvantages of interactive theorem proving include high human effort and a steep learning curve, both of which TPot is designed to reduce. Most systems-verification frameworks built on top of interactive theorem provers [4] require programmers to supply detailed proofs in powerful and often custom logics. While this methodology enables proofs of complex properties beyond the capabilities of automated verifiers, it incurs a high proof cost that is only partially remedied by framework-specific proof automation tactics. For example, verifying seL4 [36] (~8,700 LOC) required a team of verification experts and took 11 person-years. The steep learning curve is the flip side of flexibility: complex logics and powerful (functional) languages are both far outside the expertise of ordinary systems programmers.

Semi-automated verifiers. To reduce human effort, semi-automated verifiers [2, 3, 20, 34, 39, 57, 61] offload most reasoning to a powerful solver (typically SMT). Unlike in interactive theorem proving, where programmers guide the proof at the logic level, programmers guide semi-automated verifiers by inserting logic annotations in their code. These annotations define specifications (e.g., pre-/post-conditions) and provide proof hints (e.g., loop invariants). Afterward, the verifier automatically generates proof obligations from the annotated code and discharges them with a solver.

Pioneering efforts in this space include the Hyper-V hypervisor [38], verified using VCC [20], and the Ironclad project [32], which fully verifies the functional correctness of a complete software stack, including the operating system, cryptographic libraries, and sample applications.

Compared to interactive theorem proving, semi-automated verification typically requires less manual effort. As an example, the Ironclad project took 3 person-years to verify ~6,500 lines of code [32]. Semi-automated verifiers also have a flatter learning curve: users write annotations at code level to guide the verifier instead of writing proofs. Still, learning to use a semi-automated verifier is not trivial: one needs to learn the annotation language, the underlying logic, how to write specifications for both public and internal functions, how to find correct and sufficient invariants and intermediate assertions, and sometimes even the internals of the verifier, to understand why verification fails.

2.2 Automated Verification

At the other end of the spectrum are fully automated verifiers, offering the highest possible degree of proof automation: complete elimination. In such systems, users only need to write specifications for public APIs.

With systems software, achieving such a degree of automation often requires significant restrictions in expressivity (i.e.,

```c
1  int a, b;
2  // -- Internal functions --
3  void increment(int *p) {*p = *p + 1;}
4  void decrement(int *p) {*p = *p - 1;}
5  // -- Initializer --
6  void init() {
7      a = 0;
8      b = 0;
9  }
10 // -- API functions --
11 void transfer() {
12     increment(&a);
13     decrement(&b);
14 }
15 int get_sum() {return a + b;}
```

(a) System implementation.

```c
1  // -- Global invariant --
2  bool inv__sum_zero() {
3      return a + b == 0;
4  }
5  // -- Proof-oriented tests --
6  void spec__transfer() {
7      int old_a = a, old_b = b;
8      transfer();
9      assert(a == old_a + 1);
10     assert(b == old_b - 1);
11 }
12 void spec__get_sum() {
13     int res = get_sum();
14     assert(res == 0);
15 }
```

(b) TPot specification.

```c
1  void spec__increment() {
2      any(int *, p);
3      assume(p == &a || p == &b);
4      int old_a = a, old_b = b;
5
6      increment(p);
7
8      if (p == &a) {
9          assert(a == old_a + 1);
10         assert(b == old_b);
11     } else if (p == &b) {
12         assert(a == old_a);
13         assert(b == old_b + 1);
14     }
15 }
```

(c) Hypothetical specification of `increment` (not required).

Figure 1. (a) Implementation and (b) TPot specification of a toy system maintaining two integers whose sum is zero. `inv__sum_zero` states a global invariant that holds after initialization and each API function execution. `spec__get_sum` and `spec__transfer` specify API functions. (c) is a hypothetical internal function specification, which is not required by TPot to verify the API functions. It states, for instance, that the pointer argument to `increment` must equal either `&a` or `&b`, as the function would not be memory safe otherwise. TPot inlines `increment` while verifying `spec__transfer`, where the context readily implies this precondition.

	Theorem provers	Semi-automated verifiers	Automated verifiers	TPot
Manual effort	High	Medium	Medium	**Low**
Learning curve	High	Medium	Low	**Low**
Expressiveness	High	Medium	Low	**Low**
Generality	High	High	Low	**Medium**
Verification time	Low	Low	Low	**Medium**

Table 1. Main trade-offs in systems-software verification.

only certain properties can be verified) and flexibility (i.e., only certain code patterns are supported): the verifier has to be designed for a specific domain, and the code has to be written with the verifier in mind. For example, automated verifiers typically require the interfaces of system components to be finite, so that all loops in the implementation are statically bounded [48, 70]. Similarly, they commonly impose severe limits on language features (e.g., disallowing pointer arithmetic [70] and dynamic memory management [48]) to simplify the verification.

Automated verification has shown success in various domains, including file systems [63], network functions [56, 70], compilers [68], and even operating system kernels [48, 49]. Some prior work constructs individual verified systems [48], while others offer programming frameworks [63, 70], each of which builds a specific class of components that are amenable to automated verification.

The key drawback of this approach is that the manual effort essentially shifts from writing proofs to writing code in the way that the verifier needs it to be. It may not be easy to make an input program align with an automated verifier. As a result, effort that would have been invested writing proofs or logical annotations in the interactive setting may end up being invested at code development time, thus incurring a different kind of verification effort. In contrast, TPot does not impose any rules on how to write C code.

2.3 TPot: A New Point in the Design Space

We identify a different set of trade-offs that are well-suited for systems code in a modern development setting. TPot requires much less manual effort than existing approaches. Compared to interactive provers, TPot trades lots of expressiveness and some generality for a flatter learning curve, while remaining more expressive and general than automated verifiers. The price to pay is verification time: it is higher with TPot than with prior approaches, but still suitable for CI integration. Table 1 summarizes our discussion.

3 TPot Approach

With TPot, we aim to address the needs of real-world system development. This section presents *proof-oriented testing*, our approach to doing so. First, we illustrate proof-oriented testing in a simple TPot specification, then we discuss how it fits into the normal workflow of a system developer.

3.1 Specifications as Proof-Oriented Tests

TPot specifications are C programs written in a syntax similar to test suites. We acknowledge the important role that testing plays in the delivery of reliable systems; in fact, we envision verification as a natural evolution from tests to property-based tests [17] to proofs. Further, reports from major industrial outfits, such as Amazon [7, 18, 19, 21, 51, 60], Facebook [9, 54], and Microsoft [20, 32, 38], repeatedly underscore the need for verification to be driven by the programmers themselves. To reduce the cognitive burden associated with encoding correctness properties into a different language, TPot allows programmers to write specifications and annotations in the same language as the implementation.

Fig. 1 shows an example TPot specification, consisting of two proof-oriented tests (POTs) and a global invariant. POTs specify the main functional correctness properties, and

global invariants help prove their correctness. TPot verifies the system in two steps. First, it symbolically executes the initializer followed by the global invariant and ensures that the latter must return `true`. Then, TPot symbolically executes each POT under the assumption that the invariant holds over the initial system state, proving all assertions. For each POT, TPot also ensures that the invariant must return true over the final system state, which concludes an inductive proof.

TPot is built to reduce the amount of intermediate specifications and proofs the developer needs to provide. It verifies the system's public API functions without requiring pre-/post- conditions for internal methods, by inlining all function calls during verification. This significantly alleviates the annotation burden, as it is cumbersome to specify the precise calling context for internal functions. An example of this is shown in Fig. 1c. Moreover, TPot does not require proof code to explicitly fold/unfold global invariants, instead keeping all invariants visible throughout the verification process.

3.2 Verification as a CI Process

Fig. 2 shows how TPot integrates into the normal workflow of a system programmer. Developers work on various components (indicated by rounded square icons) and follow a customary code–test–debug cycle independently of TPot for each one (illustrated at the top of Fig. 2), until they achieve sufficient confidence in the code's correctness. Some of these components (in red) are critical and thus warrant the extra effort of formally proving their correctness.

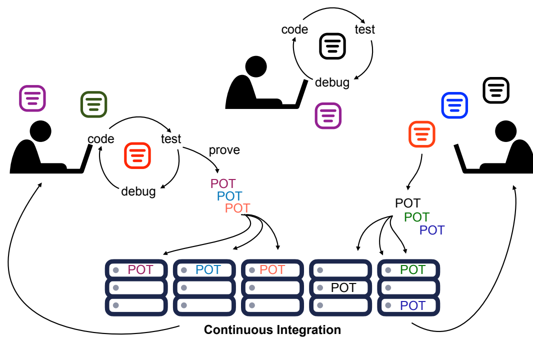

Figure 2. Developer workflow that includes TPot.

For these, the workflow eventually follows a "prove" edge, illustrated on the left side of Fig. 2: First, for each component property that a developer cares to prove, she writes a POT, typically by evolving one or more existing unit or integration tests. Next, the developer smoke-tests the POTs by running them with TPot locally, with a few concrete values for inputs and small bounds on data-structure sizes and loop iterations—this verifies some of the paths through the code and completes in seconds to minutes. She then writes invariants over loops and global state, as needed to make the POTs succeed without constraints on inputs and configurations, and pushes the specification to CI. The POTs

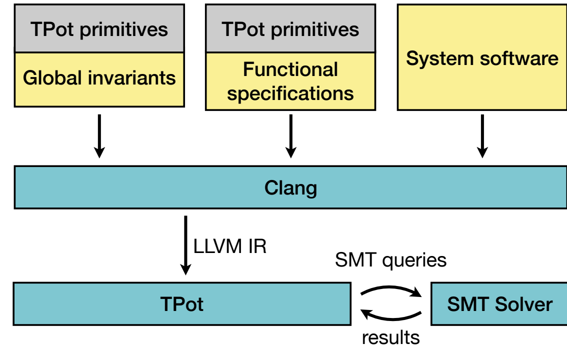

Figure 3. TPot's architecture. The top row of boxes represent inputs from programmers, blue boxes constitute the TCB.

execute in parallel, and total completion time is determined by the longest running POT, which takes from minutes to a couple of hours. TPot's stable SMT encoding, discussed in Section 4.3, helps avoid solver explosion even when the verifier runs for hours.

During the CI run, either all POTs succeed, and so the corresponding properties hold, or at least one POT fails, so there is a deep bug in the code or a problem with the property. For each failing POT, TPot provides a counterexample including (1) an initial state that satisfies all invariants and the failing POT's assumptions, in the form of an assignment of values to variables; (2) a legal code path starting at the beginning of the POT and leading to a violation; and (3) the resulting violation, which is either an assertion failure or an ending state failing an invariant, along with the failing path through the invariant.

The developer first determines whether the starting state reported by the counterexample is invalid. If so, she strengthens the global invariant. For instance, if `inv__sum_zero` was omitted from Fig. 1b, running `spec__get_sum` would result in an assertion failure at line 14. TPot would then provide a counterexample consisting of an initial state (e.g., `(a: 1,b: 0)`), the failing code path (1b:13,1a:12-13,1b:14), and the violated assertion (`res == 0`). The developer expects only states where a and b are additive inverses to be reachable, which makes this state invalid. She adds `inv__sum_zero` to encode her intuition in the specification.

If the starting state is valid, the developer debugs the failure much like a failing classical test case. From counterexamples and POTs, she can construct failing classical tests with concrete initial states and inputs. This is a straightforward manual process with our TPot's current implementation, but it can be automated using techniques similar to KLEE's replay driver. As the failing classical test is an ordinary binary, she can fix the problem using ordinary methods (e.g., using a debugger) independently of TPot. After the fix, thanks to persistent caching (described in Section 4.4), the next CI run will only recheck the affected symbolic branches.

Group	API	Description
General	① any(var_type, var_name)	Defines a symbolic variable named var_name of type var_type.
	② assume(cond_expr)	Introduces an assumption. Used to express preconditions.
	③ assert(cond_expr)	Checks cond_expr. Used to express postconditions.
Heap-related	④ bool points_to(ptr, type, name)	Returns true if ptr points to an object named name as large as sizeof(typ),false if ptr is null, dangling, or points to an object with a different name.
	⑤ bool names_obj(ptr, typ)	Short for points_to(ptr, typ, "ptr"), where "ptr" is a stringification of the first argument.
	⑥ bool names_obj_forall(ptr_f, typ)	Takes a pointer-returning function as its first argument. Returns true if for all integers i, ptr_f(i) is either NULL, or points to an object named "ptr_f" + str(i).
Universal quantification	⑦ bool forall_elem(arr, cond)	Returns true iff for all elements in array arr, cond holds.
	⑧ bool names_obj_forall_cond(ptr_f, typ, cond)	Returns true iff names_obj_forall(ptr_f, typ) returns true and, for all non-null values of ptr_f(i), cond(ptr_f(i)) holds.

Table 2. TPot users write POTs and invariants in C augmented with these eight verification-specific primitives.

4 TPot Design

Fig. 3 shows TPot's architecture. With TPot, developers write, in C, functional correctness specifications as POTs and invariants, using the verification primitives that TPot provides (§4.1). TPot lowers POTs, global invariants, and the implementation code into LLVM intermediate representation (IR), to avoid dealing directly with the complicated semantics of C [43, 48]. TPot then performs exhaustive symbolic execution (SE) over each POT, using a custom symbolic executor based on KLEE [8]. During SE, TPot checks for violations of conditions asserted by the POT, as well as low-level errors, like memory safety violations and division by zero.

During verification, TPot uses the Z3 [23] SMT solver to decide the feasibility of each branch and to enumerate all possible targets for symbolic pointers. The TCB of a system component verified with TPot includes TPot itself (which builds on KLEE), user-written specifications (POTs), Z3, and the LLVM toolchain.

Unlike push-button verifiers TPot avoids restricting the implementation language—instead, we introduce a minimal amount of annotations and interactivity, aiming to strike a balance between flexibility and automation. TPot extends C into a specification language for system components, with a limited logic that is expressive enough to capture properties of interest while being simple enough to automate. It implements this specification language using a custom memory model that supports low-level C idioms common in systems components. Crucially, TPot employs a bespoke encoding of the memory model and the specification language into SMT to avoid common causes of solver explosion [40, 46], and enable automated tools to bridge wider gaps in reasoning, with predictable performance.

4.1 TPot's Specification Language

To enable practical verification for system components, we pick a middle ground between conventional symbolic-execution APIs and powerful specification languages offered by semi-automated verifiers built to verify generic software.

Design goals. First, the specification language should be familiar to developers without verification expertise, and they should not have to learn new languages for the purpose of verification, even if they might need to learn new specification primitives. Second, as much as possible, developers should not have to write intermediate specifications for the purposes of verification—they should only specify the behavior they are interested in verifying. Last, the specification language must enable the right combination of *expressiveness* and *automated reasoning*. It should be expressive enough to specify safety of critical system components and to express functional-correctness properties of interest. On the other hand, it should be restrictive enough that properties can be encoded into SMT in a way that decreases risks of instability [40, 46], and avoids common causes of solver explosion.

Language overview. Table 2 summarizes the 8 primitives with which we extend C. TPot's specification language extends a conventional symbolic execution (SE) API with global invariants, heap primitives, and a restricted form of universal quantification. The SE primitives (①–③) define *symbolic variables*, restrict their values through *assumptions*, and *assert* conditions expressed as C boolean expressions. This style of specification resembles property-based tests [17], which recent research [7] suggests are acceptable to developers. Using these primitives, developers express functional correctness properties as POTs. TPot encapsulates shared properties in functions, making C functions serve the same purpose as logical predicates in other C verifiers [3, 34, 38, 57].

Fig. 4 shows an example of a TPot specification. `spec__create_file`, a typical POT, defines pre-conditions using assumptions, invokes the function(s) to be verified, and checks that the result meets post-conditions. The rest of this section sheds further light on different aspects of Fig. 4.

Global invariants. To facilitate the specification of stateful components, TPot includes the notion of global invariants, which are conditions that are expected to hold over the global state both before and after each API function execution. TPot treats any boolean function whose name starts with `inv__` as a global invariant (such as `inv__owners()` in Fig. 4).

Naming. Specifying heap-manipulating programs requires two facilities missing from conventional SE primitives: memory safety predicates, and mechanisms to concisely control aliasing. Interactive verification tools often provide these facilities through a permission system like dynamic frames [39, 64], separation logic [59], linear types [42], or ownership types [37]. Permission systems often require developers to explicitly manipulate proof state (e.g., fold and

```
1 extern struct file *files;
2 extern unsigned num_files;
3
4 // Helper functions
5 struct file_perm *perm_ptr_i(int i) {
6   if (i < 0 || i >= num_files) return NULL;
7   return files[i].permissions;
8 }
9 bool owner_valid(struct file_perm* p) {
10   return p->owner != PID_INVALID;
11 }
12 bool inode_diff_from(struct file *f, int i, inode_t node) {
13   return f->inode != node
14 }
15 // Global invariant
16 bool inv__owners() {
17   return names_obj(files, struct file[MAX_FILES])
18     && num_files <= MAX_FILES
19     && names_obj_forall_cond(perm_ptr_i,
20         struct file_perm, owner_valid);
21 }
22 // POT that specifies create_file
23 void spec__create_file() {
24     any(inode_t, node);
25     any(pid_t, pid);
26     assume(forall_elem(files, inode_diff_from, node));
27     int idx = create_file(node, pid);
28     if (idx > 0) {
29       assert(files[idx].inode == node);
30       assert(files[idx].permissions->owner == pid);
31     }
32 }
```

Figure 4. Example TPot specification for a file manager component, whose implementation is not shown. The component maintains an array of file structs, each containing an inode number and a pointer to a permissions struct. `spec__create_file` states the main functional correctness property: given any `pid` and `inode`, assuming the latter is not used by an existing file, `create_file` will run without low-level errors. If the returned `idx` is positive, indicating success, the array element at `idx` will reflect that `pid` owns a file located at `inode`. `inv_owners` is a global invariant which specifies (1) that the global pointer `files` names an array of file structs, whose length is `MAX_FILES`, (2) that the current number of files, is less than `MAX_FILES`, and (3) that all elements in `files` at indices less than `num_files` are valid file records, meaning their `permissions` pointers name distinct permission structs with valid owners.

unfold predicates), while linear and ownership types restrict the implementation by disallowing certain forms of aliasing.

TPot instead addresses these needs through a naming-based abstraction exposed through primitives ④–⑥. These primitives simplify and automate heap reasoning in the absence of recursive data structures. This is enabled by a crucial observation: each block of memory can be identified by a unique name, or a unique name-index pair. We next motivate and explain the naming abstraction through an example, then present a rule for renaming and discuss quantified naming.

Naming example. Figure 5 presents a toy system that maintains two dynamically allocated integers whose addresses are stored in global variables `p1` and `p2`. The system's initializer

```
1 // -- System implementation --
2 int *p1, *p2;
3 void init() {
4   p1 = malloc(sizeof(int));
5   p2 = malloc(sizeof(int));
6 }
7 void incr_p1() { // API function
8   *p1++;
9 }
10 // -- Specification --
11 void inv__alloc() { // Global invariant
12   return names_obj(p1, int) && names_obj(p2, int);
13 }
14 void spec_incr_p1() { // POT specifying incr_p1
15   int old_p1 = *p1;
16   int old_p2 = *p2;
17   incr_p1();
18   assert(*p1 == old_p1 + 1);
19   assert(*p2 == old_p2);
20 }
```

Figure 5. The implementation and TPot specification of a toy system involving dynamic memory allocation.

allocates the integers and its single API function increments the integer `p1` points to. `spec_incr_p1` is a POT specifying the behavior expected of this function.

To understand `inv__alloc`'s purpose, consider the scenario where it is omitted. Without a global invariant specifying otherwise, `p1` and `p2` can have arbitrary pointer values, including `null` or a dangling value. Therefore, dereferencing either would cause a memory safety error, and so the POT would fail at line 15. To avoid this, we need a global invariant stating that `p1` and `p2` both point to allocated memory.

Now consider a hypothetical primitive called `is_allocated(p, type)`, specifying precisely that `p` points to an allocated block of memory as large as `sizeof(type)`, and assume that the global invariant was `is_allocated(p1, int) && is_allocated(p2, int)`. In this case, the POT would execute without finding a memory safety error, but it would fail to prove the assertion on line 19. This is because the two pointers may alias, in which case incrementing `*p1` would implicitly increment `*p2` as well.

While it is possible to express non-aliasing arithmetically (`p1+sizeof(int)<=p2||p1>=p2+sizeof(int)`), this quickly becomes complicated when many heap objects are involved. Instead, TPot allows concise specifications of non-aliasing by exposing an abstraction of the heap where each block of memory carries a name, represented as a string. `points_to(p, type, name)` requires `p` to point to an allocated block which carries specified name, and `names_obj(p, type)` is a convenient shorthand for `points_to(p, type, "p")`. In the example, `inv__alloc` requires `p1` to point to a block named `"p1"`, and `p2` to `"p2"`, implying that the two blocks are distinct.

Renaming. The careful reader may wonder how TPot is able to prove that `init` establishes `inv__alloc`, since the blocks returned by `malloc` may carry different names than

"p1" and "p2". TPot enables and automates such proofs by allowing renaming at specification boundaries (i.e., before and after API function executions). Essentially, all TPot specifications are existentially quantified with respect to a mapping between objects and names. In the example, TPot will choose a renaming that maps the first block to "p1" and the second to "p2" to prove inv__alloc.

Accounting for renaming, TPot proves the following top-level theorem for each POT P:

$$INV(s) \Rightarrow \forall s'. s \rightsquigarrow_P s' \Rightarrow \neg error(s') \wedge INV(s') \quad \text{(A)}$$

where

$$INV(s) \triangleq (\exists m. \bigwedge_{inv} inv(s,m)) \quad \wedge \quad \text{(B)}$$

$$(\forall m. (\bigwedge_{inv} inv(s,m)) \Rightarrow \forall l \in alloc(s).\ \ m[l] \neq \text{""}) \quad \text{(C)}$$

We use $s \rightsquigarrow_P s'$ to denote that executing POT P on starting state s produces final state s'. For simplicity, we assume assertion violations and low-level bugs immediately terminate the POT and produce an error state. The theorem says (A) that a state satisfying INV only leads to non-error states that preserve INV. INV says (B) that there exists a mapping m from memory locations to names under which s satisfies all user-written invariants, and (C) that all such mappings map all allocated heap locations in s to non-empty names.

TPot does not directly encode the theorem above into an SMT query. Instead, it checks B by greedily constructing satisfying mappings for each symbolic execution path. It checks C through a per-memory-object check that the global invariant disallows the empty name.

TPot proves C to ensure the absence of memory leaks. Leaking objects are those that INV does not explicitly name. As such, they can be renamed by TPot to the empty name (""), identifying a leak. To prevent subtle incompleteness issues, TPot disallows passing "" as an argument to naming primitives.

Quantified naming. To generalize naming over flat data structures, TPot offers the names_obj_forall primitive. This primitive corresponds to the iterated separating conjunction in separation logic, and allows the specification of (possibly infinitely) many naming pointers at once. It can describe structures like arrays of pointers to distinct objects (as in Fig. 4) or page tables, where each entry potentially contains a pointer naming a distinct page.

Universal quantification. To keep reasoning about the specifications automatable, TPot does not support conditions that are universally quantified over all values of a type. Instead, TPot allows universal quantification over memory regions through primitives ⑦ and ⑧, which TPot's specifications can use to quantify properties over all elements in a collection, e.g., an array. For instance, while one may not quantify over integers generically in TPot specifications, one may quantify over all integers in an array.

Developers can use the forall_elem primitive to quantify properties over contiguous blocks of memory and names_obj_forall_cond to quantify over more flexible groups of objects described by quantified naming. Importantly, TPot can encode these properties into SMT queries that do not involve quantifiers in most cases (§4.3). Restricting quantification on stateful objects in this way also sidesteps gnarly soundness issues related to allocatedness of quantified variables [41].

Loop invariants. TPot also supports loop invariants to keep verification tractable when the implementation includes loops whose bounds are dynamically determined, or too large to be unrolled within an acceptable amount of time. By default, TPot will unroll all loops during symbolic execution and does not require loops to be annotated.

Features in other C verifiers that TPot does not require or chooses not to support. To reduce manual effort, the design of TPot's specification language depends on two insights specific to verifying systems components that are contrary to conventional wisdom in semi-automated verification. We observe that: (1) a large set of critical systems components can be specified using more restrictive logics than those found in existing verifiers, and (2) SMT solvers can be made stable enough to automate reasoning across function calls while keeping verification practical.

First, TPot does not require specifications for internal functions. Semi-automated verifiers ask the user to provide pre- and post-conditions for each internal method to ultimately verify properties of interests at a component's interface. TPot, in contrast, effectively inlines all internal functions, and hence only requires specifications for public API methods. This alleviates a significant portion of the specification burden: describing the effect and requirements of public API functions at a component's interface is relatively easy (these functions are typically the best documented, tested, and informally specified: developers are used to thinking about them in detail), whereas understanding specifying internal functions requires capturing complex behavior that may involve intermediate states with broken invariants.

In principle, semi-automated verifiers can inline functions, too. However, this would increase their solver load, risking solver explosions or noninteractive verification times. The latter is an issue because, unlike TPot, most verification systems do not have a way to smoke-test their specifications: the verifier is often used during development and for debugging. TPot's restricted, stable SMT encoding ensures that aggressive inlining does not lead to explosion: it at worst (predictably) increases verification time. While push-button verifiers similarly inline functions, in their case avoiding explosion comes at the cost of restricting their target programs, such that they are free of code features that could give rise to complex queries.

Second, TPot only supports a restricted form of universal

quantification, as discussed above. Supporting generic quantification would either require TPot to include quantifiers in its SMT encoding, thereby almost guaranteeing verification instability and solver explosion, or ask users to manually instantiate them, therefore increasing annotation burden. Perhaps surprisingly, as we show in §5, this design choice does not severely limit the applicability of TPot in the context of verifying systems components: systems code typically fits within this framework, and in fact prior formalizations already use quantifiers in accordance with these restrictions, without making them explicit.

Third, TPot purposefully excludes *recursive predicates* from its specification language, since proofs involving them require inductive reasoning, which is outside the capability of current SMT solvers. We argue that this design choice is justified for specifying systems components. A significant portion of low-level code operates without the abstraction of infinite memory, as they either explicitly deal with physical addresses or operate on pre-allocated buffers for performance reasons. Therefore, their implementations do not rely on infinite, truly recursive data structures. For instance, systems code typically uses linked lists to track free elements in an array of resources, not dynamically allocated pieces of memory in arbitrary locations. In this case, TPot allows users to specify such data structures using quantified primitives, akin to iterated separating conjunctions in separation logic.

Finally, TPot does not support nor require predicate folding and unfolding, a common primitive in prior semi-automated verifiers (e.g. open/close in VeriFast and wrap/unwrap in VCC). Besides enabling proofs involving recursive predicates, prior verifiers require folding and unfolding to simplify path constraints, thereby minimizing the verification time. TPot, by default, unfolds all predicates to reduce the annotation burden.

4.2 Custom Byte Memory Model

Design goals. We design TPot's memory model to provide smooth support for various C idioms involving the heap, such as type casting (especially pointer↔integer casting), pointer arithmetic, bit-packing of pointers, etc., that are widely used in systems software. However, TPot also must ensure the verification can finish within our time budget for queries generated from this memory model.

Overview of the memory model. TPot's memory model builds on top of KLEE's, which represents objects using SMT arrays of bytes accessed at bounds-checked indices. TPot extends it with a symbolic heap representation supporting lazily materialized objects, symbolic base addresses, and symbolic object sizes. Like KLEE, TPot does not distinguish between pointers and data, and uses SMT queries to resolve addresses and objects.

TPot employs a byte memory model, eliminating all higher-level types, thereby automatically reasoning about low-level C idioms. With this memory model, type casting is interpreted either as a no-op or as a zero- or sign-extension of bitvectors. Pointer arithmetic is interpreted precisely in terms of bitvector arithmetic. §4.3 describes how TPot encodes constraints in this memory model into stable queries.

Our memory model separates the heap from global variables and from the stack. While global variables and stack objects have concrete base addresses and sizes, heap objects have symbolic ones. TPot uses additional SMT constraints to encode that the ranges specified by symbolic base addresses and object sizes do not overlap. To avoid a quadratic number of constraints with respect to the number of heap objects, TPot fixes an ordering of objects in the heap. While TPot fixes an ordering, it does not constrain the distance between subsequent objects. Otherwise out-of-bounds accesses to an object could be unsoundly deemed as safe accesses to another object. Moreover, an additional level of encoding indirection (§4.3) ensures that the fixed order of objects does not unsoundly imply an ordering between pointer values.

Lazy object materialization. TPot's quantified naming primitives (§4.1) can imply the existence of a large number of heap objects, possibly infinitely many. To automate reasoning about such conditions, TPot uses a lazy materialization scheme for memory objects. TPot extends KLEE's symbolic state representation with an uninterpreted SMT function called `heap_safe`, which takes in an address and returns the number of bytes that can be safely accessed at that address. During the interpretation of naming primitives, TPot adds constraints over `heap_safe` to ensure the memory safety of all bytes that are part of named objects. Before resolving pointers to heap objects, TPot issues SMT queries to check whether the pointer is heap safe. If so, we resolve it within the heap and if not, within the stack and global variables. The latter works exactly as it does in KLEE, iterating over all objects and issuing SMT queries to find objects the pointer might fall in and forking a state for each such object. The former works similarly, except that if we determine that the pointer may be out of bounds of all objects in the heap, we fork an additional state in which we allocate a fresh heap object with symbolic content.

4.3 TPot's SMT Encoding

Eliminating quantifiers. Universal quantifiers are a common cause of solver explosion, as they may cause the solver to discover an arbitrary number of facts that are not useful for solving the current query. Worse, some combinations of quantifiers give rise to matching loops [40], compounding the issue and leading to unpredictable verification times.

To ensure the stability of its SMT encoding, TPot handles most quantifiers without sending quantified queries to the solver. Specifically, TPot only makes quantified queries before lazily materializing an object whose existence is implied by a `names_obj_forall` condition. Normally, memory

resolution queries exclude the quantified memory safety constraints (over heap_safe) that encode names_obj_forall. When TPot fails to prove the safety of a memory operation, it retries the safety check once per such quantified constraint, including one of them each time. In this way, the impact of quantifiers on verification performance is controlled: lazily including quantified constraints in this way makes it likely that they will be useful in solving the query; each query is given a single quantifier with triggers picked to avoid self-loops, so matching loops are avoided; and the process happens at most once per object, just before instantiation.

TPot instantiates constraints implied by forall_elem and names_obj_forall_cond primitives without involving the solver. Instead of putting quantified constraints in the path condition, these primitives cause TPot to mark the objects they relate to with a reference to the quantified condition. Later, when a byte associated with a forall_elem is read, or when an object identified by a names_obj_forall_cond is lazily materialized, TPot computes the property over the specific byte or object and adds it to the path condition.

Converting pointer values, heap addresses, and object sizes to integers. Since TPot does not distinguish between pointers and data and supports bit-level operations over both, it needs to represent pointer values, heap addresses, and object sizes as 64-bit vectors. However, we have found that doing so naively tends to leads to solver explosion in the form of *bit-blasting* over these values. When the solver fails to solve a query involving bitvectors heuristically through bitvector arithmetic, it tries to discharge the query using propositional logic, which entails interpreting a 64-bit vector as 64 independent boolean variables, giving rise to 2^{64} combinations.

With a naive encoding, bit-blasting happens frequently during pointer resolution, since heap addresses relate to each other through bitvector arithmetic. To mitigate this, TPot converts bitvectors to integers during pointer resolution, allowing Z3 to use integer arithmetic solvers.

Converting precisely between bitvectors and integers (e.g. using the bv2int function offered by Z3) is prohibitively costly for SMT solving and can itself cause instability. Instead, TPot approximates the bv2int conversion using an uninterpreted function tpot_bv2int that assumes *overflow-free semantics* and is *explicitly axiomatized.*

TPot performs the bv2int conversion only during pointer resolution, where bitvector overflows are impossible. To make this fact clear to the SMT solver, we supply it with instantiations of axiom schemas capturing overflow-free conversion from bitvectors to integers. Figure 6 shows an example axiom schema for the + operator.

TPot does not give a general, quantified axiom to the SMT solver. Instead, TPot allows *explicit* rewrites based on instantiations of the schema, for specific pointers and in only one context: checking whether a pointer $p1 + p2$ falls within the bounds of an object, i.e., $t_{b2i}(base) \leq t_{b2i}(p1 + p2) < t_{b2i}(base + size)$, where t_{b2i} denotes tpot_bv2int. TPot lifts this to checking that $t_{b2i}(base) \leq t_{b2i}(p1) + t_{b2i}(p2) < t_{b2i}(base) + t_{b2i}(size)$. The lifting is sound in this context: since all malloc-ed objects must fit in the heap, $base + size$ may not overflow. Proving the integer sum of $p1$ and $p2$ to be less than that of $base$ and $size$ readily implies that $p1 + p2$ does not overflow either.

```
1 (= (tpot_bv2int (bvadd a b))
2    (+ (tpot_bv2int a) (tpot_bv2int b)))
```

Figure 6. Example axiom schema from the overflow-free semantics for tpot_bv2int, stating that the result of applying tpot_bv2int to the bitvector sum of a and b should be equal to the sum of their integer conversions. TPot soundly instantiates this schema for specific values of a and b during pointer resolution.

In addition, TPot propagates constraints over bitvectors to integers. For example, given that a bitvector bv is less than 0x05, the solver cannot deduce that $t_{b2i}(bv)$ is less than the integer 5. Hence, TPot explicitly adds the corresponding integer constraints, whenever TPot adds a bitvector constraint to the path condition.

Importantly, the bv2int conversion hides the ordering of heap objects (§4.2) from client code, as TPot maintains ordering constraints only over integer values. For instance, if TPot decides to have the 4-byte object that p1 points to come before the object that p2 points to, it will insert the constraint $t_{b2i}(p1) + 4 \leq t_b 2i(p2)$ into the path condition, but not $p1 + 4 \leq p2$. The solver cannot infer the latter from the former, as it sees tpot_bv2int as an uninterpreted function, and it is never given the axiom schema instantiation required to make such a deduction. This prevents client code from unsoundly relying on an ordering between pointers, since the bv2int conversion is transparent to client code.

Query simplification. To further stabilize SMT solving, TPot includes a custom query simplifier. Unlike prior approaches that simply propagate constants and equalities [8], TPot's simplifier performs much more extensive simplification, often involving intermediate SMT queries to check whether certain simplification is possible. While intermediate queries can impact TPot's verification time negatively, they benefit its stability as they may prevent solver explosions in more complex queries. Below, we present two examples of the types of simplification TPot performs.

Read after write. For expressions of the form (Read (Write O x j) i), TPot checks if it is possible to simplify to either (Read O i), or x. To achieve this, TPot makes SMT queries to check if i is provably equal to or provably different from j.

Performed naively, this simplification can lead to a prohibitive number of queries, as (1) the read-after-write scenario occurs often and (2) involves reads that span multiple bytes over objects that have been written multiple times, TPot mitigates the associated cost with two techniques. First, TPot

caches simplification proofs: once a simplification is proven to be sound, it will always be sound for the same symbolic execution state, regardless of what new conditions have been added or how many new objects have been created. Second, instead of making a query per byte written and per byte read, TPot identifies syntactic structures (e.g. `Reads` chained together by `Concat` expressions, and `Writes` at adjacent indices), and attempts to simplify over ranges as opposed to individual byte indices.

Constant offsets. During pointer resolution, if TPot infers that the difference between a pointer and a base address must be constant, it caches and reuses this fact to rewrite byte offsets in later queries. Concretely, while performing a memory operation (say, a read), after resolving a pointer p to an object O, TPot will construct an expression of the form `(Read` O `(Subtract` P base_of(O)`))`. If TPot infers that `P` is equal to base_of(O) plus some constant, it will substitute `P` in the `Read`, and in all subsequent expressions.

4.4 TPot Prototype

We implemented TPot on top of the KLEE symbolic executor [8] by adding or modifying 10,418 LOC. In addition to the design presented above, we added internal capabilities, such as checkpointing function results over specific system states, so that quantified properties can be instantiated over the state they describe. There are two lessons that we expect to be applicable to other solver-aided frameworks.

Solver portfolio and SMT bugs. To reduce the solver query time, TPot uses a portfolio of solvers. Following prior work [29, 45, 55], TPot *races* multiple solvers: it sends the queries to multiple solvers that run in parallel, and uses the earliest returned result.

When using a solver portofolio, one should expect to encounter solver bugs more often. Solvers of course have bugs [69], but what is not obvious is that buggy behavior is often faster than correct behavior. In a portfolio, the solver hitting a bug is likely to return its incorrect result faster than the other solvers. In other words, a solver portfolio is likely to exhibit a bug *any time* one of the individual solvers does so. Therefore, a solver portfolio is more often wrong than an individual solver. To overcome this issue, we recommend that TPot users validate the results returned by the portfolio a posteriori, by re-running each query in a separate CI job that waits for multiple solvers in the portfolio to terminate and checking that they agree.

Persistent query caching. During development, programmers often make commits with frequent small changes to the code and/or specifications of various components, repeating a large number of queries that have been solved before.

Inspired by prior work [65], TPot extends query caching by making the results *persistent on disk*. In this way, the query results can be reused from one run to the next. In a CI setting, this helps ensure that the CPUs are used to verify the modified parts of the code rather than on re-solving previously solved queries.

We found that building a persistent cache early in the development of TPot was crucial to facilitating not only its use but also TPot's development: being able to reproduce errors without having to wait for queries to be re-solved made debugging TPot itself much faster.

5 Evaluation

Our evaluation answers the following questions:

- Can TPot verify a variety of systems code without requiring code to be modified to suit the verifier? (§5.1)
- Does TPot reduce the manual effort required to verify the components, and by how much? (§5.2)
- Is verification time with TPot compatible with a CI pipeline? How does it break down among the different types of work that TPot performs? (§5.3)

Evaluation Environment: TPot is meant to run as part of a CI workflow, so we perform our evaluation on a powerful CI-class machine: 2 x 12-core Intel Xeon Gold 6248R processors with 384GB RAM, hyperthreading enabled, with Ubuntu 20.04 and Linux kernel 5.4.0. We disable the persistent SMT query cache (§4.4), to evaluate the worst case for TPot. The solver portfolio (§4.4) includes 15 instances of Z3, involving 13 versions of Z3 between 4.4.1 and 4.12.5, some with different configuration parameters (e.g., arithmetic solver, branch/cut ratio, number of threads).

5.1 Verifying Diverse Systems-Code Components

We use TPot to verify 6 different code bases that have been previously verified with 4 different verifiers, shown in Table 3. These code bases cover a wide range of systems code with low-level systems programming idioms, such as pointer↔integer conversions, pointer arithmetic, dynamic memory allocation/free, and more.

Target name	Category	Previously verified with	LOC
pKVM emem allocator	Heap allocator	CN [57]	96
Vigor allocator	Resource manager	VeriFast [34]	96
KVM page table	Page table	RefinedC [61]	135
USB driver	Device driver	VeriFast [34]	523
Komodo^S	Security monitor	Serval [49]	1409
Komodo*	Security monitor	n/a	1431

Table 3. Evaluation targets. Some are system components, others are full systems but nevertheless still component-sized.

Evaluation targets: Google pKVM [44] is a hypervisor used on Android devices for isolation between the host Linux kernel and guest virtual machines. The pKVM emem allocator is used by pKVM during boot-up. It tracks memory allocation over a contiguous address range using long integers. It casts

the address of the next free page into a pointer, so that the page can be zero-initialized at allocation time.

The Vigor allocator [71] is used in various network functions to manage a variety of objects (e.g., IP addresses and ports in a NAT). It maintains timestamps associated with each object in an array, so that the object can be reclaimed if the object lease is not renewed prior to expiration.

The KVM page table [62] is a case study used by RefinedC [72], in essence a simplified version of the Linux KVM page table that can set a page table entry (PTE), mark a PTE as invalid, set the protection bits of a PTE, and check whether a page is in use or not. It relies on bitwise operations to operate over PTEs, which pack page-aligned physical addresses and protection bits into 64-bit integers.

USB driver [35] is a Linux device driver for USB mice, used as a case study by VeriFast [34]. It includes code to handle interrupts, mouse probing, and disconnection, and to open/close the mouse device file. It dynamically allocates internal data structures during probing, and frees them during disconnection. It makes calls to the Linux USB core, an abstraction over USB hardware controllers, as well as the Linux input subsystem. It type-casts buffer pointers returned by various Linux APIs into driver-specific control structures.

Komodo [25] is the software-based security monitor for enclaves mentioned in §1. Komodo^{S} [50] is the Komodo version ported by the Serval team, with pointers and virtual-to-physical address translation removed, to be verifiable by Serval. Komodo* is Komodo^{S} with the VA-to-PA translation and pointers and associated arithmetic added back in.

Properties verified with TPot: For the first 5 targets, we ported the specifications written for the 4 baseline verifiers to POTs written in TPot's specification language. We then ran these POTs, without modifying any of the targets' existing code. For Komodo*, the last target, we added back the pointer support and address translation removed for Serval, and re-verified the same specifications. This demonstrates that TPot's specification language is powerful enough to express the specifications written for the baseline verifiers, including functional correctness properties and the absence of low-level bugs. Below, we detail properties verified with TPot.

For the pKVM emem allocator, we verify that allocation APIs return pointers to zero-initialized memory chunks of the expected size, and that they update global variables to ensure the same memory will not be allocated again.

For the Vigor allocator, we verify that: object borrowing (leasing) succeeds only for objects not previously in use; refreshing and returning objects updates their timestamps correctly; the timestamps of unrelated objects remain unchanged after borrowing/refreshing/returning an object. The original VeriFast specification also describes when borrowing succeeds by relating the state of the allocator to an abstract set of previously borrowed objects. Since this abstract state is neither part of the implementation nor observable through the allocator's API, our POTs do not capture it.

For the KVM page table, we verify that each function modifies its PTE arguments as specified by the RefinedC formalization (we reason directly on bitvectors, whereas RefinedC abstracts them into field-based structures).

For the USB driver, we verify that: opening/closing a device respectively submits/cancels USB request blocks; probing initializes the device and allocates necessary data structures on success; and all such data structures are freed upon disconnection. We also verify that the driver calls Linux functions correctly (i.e., meets their pre-conditions). The VeriFast specification does not consider the actual implementation of these Linux functions but relies on trusted VeriFast contracts instead. We take a similar approach: we model their behavior using simple C functions. Both VeriFast's and TPot's proofs, therefore, assume that the models/contracts for these external functions are representative of their actual behavior.

For all evaluation targets, we also prove the absence of low-level bugs, such as out-of-bounds array accesses and divide-by-zero. We also verify that their initializers establish all global invariants that the verification of API functions relies on, and that each API function maintains these invariants.

TPot is designed to verify system components as a whole and is not concerned with how different proofs compose. As such, we leave out compositional aspects of the baseline specifications: for the pKVM emem allocator, we do not specify the transfer or ownership over the allocated memory. For Komodo, we omit the derived properties implied by the composition of functional correctness specifications, such as reference-count consistency. The baseline annotation overheads we report in Table 4 do not count lines of specification related to these aspects.

TPot's specification language does not include (and does not require) the notion of ownership. We do, however, leverage TPot's naming-based heap primitives to specify and verify memory safety (e.g., that only allocated memory is accessed) and the absence of memory leaks.

Summary: TPot can verify diverse systems components without requiring their code to be modified to suit TPot. TPot proves the absence of low-level bugs and achieves the same functional correctness guarantees as the baselines, though without the same level of abstraction or compositionality.

5.2 Annotation Overhead

It is hard to fully quantify the extent to which TPot reduces the developer effort involved in proving code correct: there are aspects that are inherently hard to quantify (e.g., intuitiveness of the specification language) and others that require statistically sound human-subject experiments and usability studies that are beyond the scope of our evaluation. Therefore, we limit ourselves to measuring overhead in terms of what a developer needs to write to verify their code. The results are presented in Table 4. This underestimates TPot's

benefits—for example, it is easier for a C programmer to write annotations in TPot's C-based language than in Coq or VeriFast, but we cannot rigorously assess by how much. We use `cloc` [1] to count the lines of implementation code, and manually-written tags counted by scripts to determine the annotation counts for each category.

	pKVM emem allocator			Vigor allocator		KVM page table		USB driver		Komodo^S		Komodo*
	CN	Verifast	**TPot**	Verifast	**TPot**	RefinedC	**TPot**	Verifast	**TPot**	Serval	**TPot**	**TPot**
Specifications	22	23	34	53	61	132	69	164	63	563	560	574
Internal	13	10	0	4	0	24	0	409	0	0	0	0
Predicates	7	5	0	27	0	0	0	97	0	0	0	0
Proof	0	3	0	84	0	62	0	18	0	0	0	0
Loops	11	7	19	17	25	0	0	0	0	0	0	0
Globals	17	12	5	0	5	0	0	0	22	266	158	178
Linux models	0	0	0	0	12	0	0	0	185	0	0	0
Syntactic total	70	60	58	185	103	218	69	688	270	829	718	752
Semantic total	63	59	38	166	79	208	63	581	209	784	495	520
Syntactic overhead	73%	62%	60%	193%	107%	161%	51%	132%	52%	59%	51%	53%
Semantic overhead	66%	61%	**40%**	173%	**82%**	154%	**47%**	111%	**40%**	56%	**35%**	**36%**

Table 4. Annotation overhead (in lines). *Specifications* is for specifying API functions and related definitions. *Internal* is for specifying pre- and post-conditions of internal functions. *Predicates* are code annotations for folding/unfolding predicates. *Proof* is for proof annotations. *Loops* is for loop invariants. *Globals* is for global invariants and global data structure predicates. *Linux models* is the C code used by TPot to model Linux functions. *Syntactic total* is the total annotation overhead counted syntactically, as is the standard in prior literature. *Semantic total* is our count of semantically relevant lines, excluding purely syntactic elements. *Syntactic overhead* reports the syntactic proof-to-code ratio, while *Semantic overhead* is the more meaningful ratio, based on *Semantic total*.

TPot specifications aim to remain true to standard C syntax. As a result, they involve many lines that result from syntactic and stylistic choices. We therefore list in Table 4, under *Semantic total*, our count of lines that are semantically relevant, for each verifier. We omit from this count lines such as sole delimiters (e.g., `)`,`:`,`}`,`/*@`, `|}`) and include/import statements in all five languages, as well as empty return statements and loop invariant signatures in our C specifications. The precise accounting for each line is accessible as part of our artifact [10]. Our main overhead metric (*Semantic overhead*) excludes such lines, because their associated cognitive load for developers is essentially zero.

Compared to our baselines, TPot incurs 1.03–3.15× less syntactic overhead and 1.53–3.28× less semantic overhead. This reduction results primarily from two design choices:

First, TPot expresses constraints directly over C variables, instead of abstract logic counterparts, so its specifications are more succinct. As a simple example, the VeriFast specification for the pKVM emem allocator includes a pre-condition requiring that the address marking the end of the memory range be < `UINTPTR_MAX`. The TPot specification does not need to explicitly state this condition, because it is known to hold for all long integers in C. The overhead of bridging the gap between implementation state and specification state is particularly aggravated in interactive frameworks: for the KVM page table, 79% of the overhead in the RefinedC specification comes from Coq definitions that abstract over implementation state.

Second, TPot removes the need for internal function specifications, proof hints, or explicit folding/unfolding of predicates, which further reduces the annotation overhead. This is especially true of code involving custom data structures with pointers, as in the Vigor allocator. For the USB driver, replacing VeriFast's specifications of internal Linux functions with TPot's simplified C models reduces by 55% the lines of code that need to be written for this aspect of the proof.

Summary: Compared to the baseline verifiers, the annotation overhead TPot incurs is consistently less, with reductions ranging up to more than 3×. This is an underestimate of the true benefit that TPot offers practitioners in terms of reduction in effort, and thus increase in productivity.

5.3 Verification Time

TPot verifies a component by running all POTs in parallel, as POTs are independent of each other (§3). Table 5 shows the verification time for our 6 targets, as well as the number of POTs for each one.

	pKVM emem allocator	Vigor allocator	KVM page table	USB driver	Komodo^S	Komodo*
# of POTs	4	5	3	5	16	16
Avg	21s	1m36s	6s	4m6s	11m36s	25m54s
Min	1s	21s	3s	1m54s	7m48s	7m54s
Max	49s	5m18s	9s	7m36s	18m18s	59m36s
CI time	**2m18s**	**7m18s**	**2m18s**	**10m6s**	**20m24s**	**1h4m**
CPU time	1m24s	8m24s	18s	20m24s	3h5m	6h54m

Table 5. Number of POTs and verification time. *Min/Max* show the time it takes to run the fastest/slowest POT, and *Avg* shows the average. *CI time* shows how long it takes end-to-end to run all POTs for a given target. *CPU time* shows total CPU time used.

The most important results are shown in the *CI time* row: this comprises the time to set up all the POTs, execute them in parallel, and tear them down. This is the amount of time that a CI pipeline must wait for the verification to complete and includes all the waits, including I/O. The *CPU time* column shows total CPU time consumed for the verification, and is a good proxy for the monetary cost of the verification in a CI cloud that charges per CPU-hour.

The contrast in trade-offs between TPot and the baseline verifiers is significant: their verification times can be substantially lower than TPot's (numbers not shown: tens of minutes for Komodo^S, verified with a push-button verifier, to as low as seconds for the other targets, verified with semi-automated verifiers), but they require more effort to use. TPot shifts most of the effort to the verifier by targeting verification times compatible with a CI workflow as opposed to pseudo-interactive use.

Fig. 7 shows where TPot spends its time: Most of it (53–80%) is in the solver, which is not surprising, given how much

TPot relies on the solver for automation. Depending on the characteristics of the code, a larger fraction may go toward resolving pointers to memory objects or toward determining branch feasibility. A non-negligible amount of time (8–28%) is spent on serializing the queries into a format compatible with the solver portfolio. This overhead can potentially be eliminated by re-designing the interface to the solver portfolio, or using memory shared between TPot and the solver to pass the queries (though this may be less compatible with a cloud-based solver portfolio service).

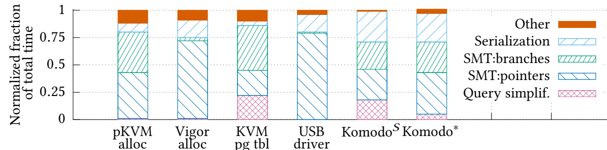

Figure 7. Breakdown of verification time. *Query simplif* is time spent simplifying queries (§4.3), potentially using the solver. *SMT:pointers* is time spent on queries made to resolve pointers to memory objects. *SMT:branches* is time spent on queries made to determine branch feasibility. *Serialization* is time spent on serializing queries before sending to the solver.

Summary: The amount of time it takes to verify system components using TPot is compatible with modern CI pipelines. The practitioner therefore gets to trade verification time for a reduction in annotation overhead.

6 Limitations, Future Work, Discussion

There are four main limitations to the current implementation of TPot. First, it makes simplifications that rule out quantification over stack depth, and hence unbounded recursion (§4.1) and certain recursive data structures. This is a pragmatic choice: as shown in §5, TPot can still handle many interesting code and data structure patterns. Second, it still requires user effort to specify invariants for unbounded loops. We posit that many such invariants could be inferred, using e.g. loop templates [68]. Third, it has no support for concurrency. This could be fixed by extending KLEE and updating our memory model accordingly [14, 31, 53, 66, 67]. Fourth, it is restricted to verifying individual systems components. This is by design: we aim to reduce the effort required for component-level verification before attempting system-level verification. We do not tackle the problem of composing component-level proofs into end-to-end verified systems.

We only tested TPot on relatively small components, up to 1500 lines. The path explosion problem, which TPot inherits from traditional symbolic execution, presents a challenge in scaling to larger components or full systems. Though supporting loop invariants partially addresses path explosion, verifying large systems is still out of scope for TPot.

While the limited quantification offered by TPot's primitives proved expressive enough for our case studies, in general there are many properties it cannot express. For instance, properties involving aggregation (e.g., the sum of elements in an array) cannot be expressed in a per-element fashion. We posit that TPot could be extended to handle such properties while keeping automated verification tractable. This could be achieved either by using more powerful array theories [58] in the solver backend, or by providing specification primitives that offer scalable encodings for common idioms in systems code, akin to Serval's treatment of reference counting.

We chose C as the language for both implementations and specifications. For the most complex verification problems, a clean and pure functional specification language with a powerful logic would be a better fit: imperative C is not a good language to write high-level functional models in. However, one surprising result of our experiments is that C expressions are sufficient for our purposes, once we augment them with limited quantification. Even better, this means that we can reuse existing unit tests, already written in C, as the basis for our specifications.

Rust is another promising implementation language for verified systems code, as demonstrated by Verus [37]. We target C as it is still the de-facto language for systems software, but we believe that most of the techniques in TPot would be applicable to the verification of Rust code, particularly around unsafe sections. Unsafe code is typically short and hidden behind well-isolated components, aligning with the component-level verification approach. Moreover, similar to C, unsafe Rust includes verification-hostile features such as raw pointers, which TPot is specifically designed to handle.

7 Conclusion

This paper presents TPot, a novel verifier that occupies a new design point for system component verification. TPot 1) incurs less manual effort than existing semi-automated verifiers; 2) is friendly for ordinary programmers; 3) is not limited to specific types of applications; and 4) does not require adapting systems code for verification. TPot employs custom SMT encoding for systems code to automate pointer operation reasoning, aggressively inlines internal functions, and includes a C-based specification language that is expressive enough for most properties in systems software while ensuring automated verification. Our evaluation shows that TPot can verify various systems components with the same guarantees as state-of-the-art verifiers, without requiring code changes. TPot consistently reduces manual effort, with reductions ranging up to more than 3×, compared to the state of the art. It takes at most around an hour to complete verification, making it suitable for CI deployment.

8 Acknowledgements

We thank our shepherd Jay Lorch and the anonymous reviewers for their generous feedback that significantly improved the paper. We are also grateful to Rishabh Iyer, Solal Pirelli, Yugesh Kothari, and Vijay Chidambaram for insightful conversations at various stages of this project.

References

[1] Al Danial. Cloc. http://cloc.sourceforge.net/.

[2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, 2005.

[3] P. Baudin, F. Bobot, D. Bühler, L. Correnson, F. Kirchner, N. Kosmatov, A. Maroneze, V. Perrelle, V. Prevosto, J. Signoles, and N. Williams. The dogged pursuit of bug-free C programs: The Frama-C software analysis platform. *Comm. ACM*, 2021.

[4] L. Beringer and A. W. Appel. Abstraction and subsumption in modular verification of C programs. In *FM*, 2019.

[5] Y. Bertot and P. Castéran. Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions, 2013.

[6] K. Bhargavan, B. Bond, A. Delignat-Lavaud, C. Fournet, C. Hawblitzel, C. Hritcu, S. Ishtiaq, M. Kohlweiss, R. Leino, J. R. Lorch, K. Maillard, J. Pan, B. Parno, J. Protzenko, T. Ramananandro, A. Rane, A. Rastogi, N. Swamy, L. Thompson, P. Wang, S. Z. Béguelin, and J. K. Zinzindohoue. Everest: Towards a verified, drop-in replacement of HTTPS. In *SNAPL*, 2017.

[7] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, J. V. Geffen, and A. Warfield. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *SOSP*, 2021.

[8] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[9] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *NFM*, 2015.

[10] C. Cebeci, Y. Zou, D. Zhou, G. Candea, and C. Pit-Claudel. TPot artifact. https://github.com/dslab-epfl/tpot, 2024.

[11] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich. Verifying concurrent, crash-safe systems with Perennial. In *SOSP*, 2019.

[12] T. Chajed, J. Tassarotti, M. Theng, R. Jung, M. F. Kaashoek, and N. Zeldovich. GoJournal: a verified, concurrent, crash-safe journaling system. In *OSDI*, 2021.

[13] T. Chajed, J. Tassarotti, M. Theng, M. F. Kaashoek, and N. Zeldovich. Verifying the DaisyNFS concurrent and crashsafe file system with sequential reasoning. In *OSDI*, 2022.

[14] Y.-S. Chang, R. Jung, U. Sharma, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich. Verifying vMVCC, a high-performance transaction library using multi-version concurrency control. In *OSDI*, 2023.

[15] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using crash Hoare logic for certifying the FSCQ file system. In *SOSP*, 2015.

[16] H. Chen, T. Chajed, A. Konradi, S. Wang, A. İleri, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *SOSP*, 2017.

[17] Z. Chen, C. Rizkallah, L. O. Connor, P. Susarla, G. Klein, G. Heiser, and G. Keller. Property-based testing: Climbing the stairway to verification. In *SLE*, 2022.

[18] N. Chong, B. Cook, K. Kallas, K. Khazem, F. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle. Code-level model checking in the software development workflow. In *ICSE-SEIP*, 2020.

[19] A. Chudnov, N. Collins, B. Cook, J. Dodds, B. Huffman, C. MacCárthaigh, S. Magill, E. Mertens, E. Mullen, S. Tasiran, A. Tomb, and E. Westbrook. Continuous formal verification of Amazon s2n. In *CAV*, 2018.

[20] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, 2009.

[21] B. Cook. Formal reasoning about the security of Amazon Web Services. In *CAV*, 2020.

[22] L. de Moura and S. Ullrich. The Lean 4 theorem prover and programming language. In *CADE*, 2021.

[23] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[24] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *IEEE S&P*, 2019.

[25] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *SOSP*, 2017.

[26] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *EuroSys*, 2017.

[27] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *POPL*, 2015.

[28] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent os kernels. In *OSDI*, 2016.

[29] F. Haedicke, S. Frehse, G. Fey, D. Große, and R. Drechsler. metaSMT: Focus on your application not on solver integration. In *DIFTS*, 2011.

[30] T. Hance, A. Lattuada, C. Hawblitzel, J. Howell, R. Johnson, and B. Parno. Storage systems are distributed systems (so verify them that way!). In *OSDI*, 2020.

[31] T. Hance, Y. Zhou, A. Lattuada, R. Achermann, A. Conway, R. Stutsman, G. Zellweger, C. Hawblitzel, J. Howell, and B. Parno. Sharding the state machine: Automated modular reasoning for complex concurrent systems. In *OSDI*, 2023.

[32] C. Hawblitzel, J. Howell, J. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *OSDI*, 2014.

[33] A. Ileri, T. Chajed, A. Chlipala, F. Kaashoek, and N. Zeldovich. Proving confidentiality in a file system using DiskSec. In *OSDI*, 2018.

[34] B. Jacobs and F. Piessens. The VeriFast program verifier, 2008. URL https://lirias.kuleuven.be/retrieve/30786.

[35] B. Jacobs and F. Piessens. VeriFast repository, 2024. https://github.com/verifast/verifast.

[36] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. E. R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, 2009.

[37] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel. Verus: Verifying Rust programs using linear ghost types. In *OOPSLA*, 2023.

[38] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *FM*, 2019.

[39] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, 2010.

[40] K. R. M. Leino and C. Pit-Claudel. Trigger selection strategies to stabilize program verifiers. In *CAV*, 2016.

[41] R. Leino. Range of bounds variables in set comprehensions not limited to allocated objects. https://github.com/dafny-lang/dafny/issues/3605, 2023.

[42] J. Li, A. Lattuada, Y. Zhou, J. Cameron, J. Howell, B. Parno, and C. Hawblitzel. Linear types for large-scale systems verification. In *OOPSLA*, 2022.

[43] X. Li, X. Li, W. Qiang, R. Gu, and J. Nieh. Spoq: Scaling machine-checkable systems verification in Coq. In *OSDI*, 2023.

[44] D. Makwana and C. Pulte. CN pKVM early allocator case study, 2024. https://github.com/rems-project/CN-pKVM-early-allocator-case-study.

[45] M. Mann, A. Wilson, C. Tinelli, and C. Barrett. Smt-Switch: A solver-agnostic C++ API for SMT solving (extended abstract). In *SMT*, 2020.

[46] S. McLaughlin, G.-A. Jaloyan, T. Xiang, and F. Rabe. Enhancing proof stability. In *Dafny Workshop*, 2024.

[47] V. Narayanan, T. Huang, D. Detweiler, D. Appel, Z. Li, G. Zellweger, and A. Burtsev. RedLeaf: Isolation and communication in a safe operating system. In *OSDI*, 2021.

[48] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. Wang. Hyperkernel: Push-button verification of an OS kernel. In *SOSP*, 2017.

[49] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang. Scaling symbolic evaluation for automated verification of systems code with Serval. In *SOSP*, 2019.

[50] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang. Serval repository, 2024. https://github.com/uw-unsat/serval.

[51] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon Web Services uses formal methods. *Comm. ACM*, 2015.

[52] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer Science & Business Media, 2002.

[53] J. Oberhauser, R. L. de Lima Chehab, D. Behrens, M. Fu, A. Paolillo, L. Oberhauser, K. Bhat, Y. Wen, H. Chen, J. Kim, and V. Vafeiadis. VSync: Push-button verification and optimization for synchronization primitives on weak memory models. In *ASPLOS*, 2021.

[54] P. W. O'Hearn. Continuous reasoning: Scaling the impact of formal methods. In *LICS*, 2018.

[55] H. Palikareva and C. Cadar. Multi-solver support in symbolic execution. In *CAV*, 2013.

[56] S. Pirelli, A. Valentukonytė, K. Argyraki, and G. Candea. Automated verification of network function binaries. In *NSDI*, 2022.

[57] C. Pulte, D. C. Makwana, T. Sewell, K. Memarian, P. Sewell, and N. Krishnaswami. CN: Verifying systems C code with separation-logic refinement types. In *POPL*, 2023.

[58] R. Raya and V. Kunčak. NP satisfiability for arrays as powers. In *VMCAI*, 2022.

[59] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *IEEE-LCS*, 2002.

[60] N. Rungta. A billion SMT queries a day (invited paper). In *CAV*, 2022.

[61] M. Sammler, R. Lepigre, R. Krebbers, K. Memarian, D. Dreyer, and D. Garg. RefinedC: Automating the foundational verification of C code with refined ownership types. In *PLDI*, 2021.

[62] M. Sammler, R. Lepigre, R. Krebbers, K. Memarian, D. Dreyer, and D. Garg. KVM page table case study with RefinedC, 2024. https://gitlab.mpi-sws.org/iris/refinedc/blob/7945a29d1647970709a9b5ad2ffc53c757e130cc/linux/casestudies/pgtable.c.

[63] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang. Push-button verification of file systems via crash refinement. In *OSDI*, 2016.

[64] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. In *ACM Trans. Prog. Lang. Syst.*, 2012.

[65] J. Taljaard, J. Geldenhuys, and W. Visser. Constraint caching revisited. In *NFM*, 2020.

[66] R. Tao, J. Yao, X. Li, S.-W. Li, J. Nieh, and R. Gu. Formally verified memory protection for a commodity multiprocessor hypervisor. In *USENIX Security*, 2021.

[67] R. Tao, J. Yao, X. Li, S.-W. Li, J. Nieh, and R. Gu. Formal verification of a multiprocessor hypervisor on Arm relaxed memory hardware. In *SOSP*, 2021.

[68] R. Tao, Y. Shi, J. Yao, X. Li, A. Javadi-Abhari, A. W. Cross, F. T. Chong, and R. Gu. Giallar: push-button verification for the qiskit quantum compiler. In *PLDI*, 2022.

[69] D. Winterer, C. Zhang, and Z. Su. Validating SMT solvers via semantic fusion. In *PLDI*, 2023.

[70] A. Zaostrovnykh, S. Pirelli, R. R. Iyer, M. Rizzo, L. Pedrosa, K. J. Argyraki, and G. Candea. Verifying software network functions with no verification expertise. In *SOSP*, 2019.

[71] A. Zaostrovnykh, S. Pirelli, R. R. Iyer, M. Rizzo, L. Pedrosa, K. J. Argyraki, and G. Candea. Vigor repository. https://github.com/vigor-nf/vigor, 2019.

[72] F. Zhu, M. Sammler, R. Lepigre, D. Dreyer, and D. Garg. BFF: Foundational and automated verification of bitfield-manipulating programs. In *OOPSLA*, 2022.

[73] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: a verified modern cryptographic library. In *CCS*, 2017.

[74] M. Zou, H. Ding, D. Du, M. Fu, R. Gu, and H. Chen. Using concurrent relational logic with helpers for verifying the AtomFS file system. In *SOSP*, 2019.

A Example Specification and Verification

In this appendix, we present the TPot specification for the pKVM emem allocator case study. Then, we explain in detail how TPot verifies a part of the specification. The appendix has not been peer-reviewed.

A.1 TPot Specification for pKVM emem allocator

Below is the specification we use in our pKVM emem allocator case study. The specification consists of a global invariant, four POTs specifying API functions, and a helper function passed to `forall_elem` in two of the POTs.

```
1  // Global invariant
2  bool inv__early_alloc() {
3    return names_obj((char *)base, char[NUM_PAGES*PAGE_SIZE])
4      && end == base + NUM_PAGES * PAGE_SIZE
5      && cur >= base && cur <= end;
6  }
7
8  // Helper function
9  bool alloc_range_zero(int64_t i, int64_t start, int64_t end) {
10   if (i < start || i >= end) return true;
11   return ((char *)base)[i] == 0;
12 }
13
14 // POTs
15 void spec__alloc_page() {
16   assume(cur + PAGE_SIZE < end);
17
18   unsigned long prev_end = end, prev_cur = cur;
19
20   char *result = hyp_early_alloc_page();
21
22   assert(result != NULL);
23   assert(forall_elem((char *)base, &alloc_range_zero,
24     result - base, result - base + PAGE_SIZE));
25
26   assert(cur == prev_cur + PAGE_SIZE);
27   assert(end == prev_end);
28 }
29
30 void spec__alloc_contig() {
31   any(unsigned int, nr_pages);
32   assume(nr_pages > 0);
33   assume(cur + PAGE_SIZE  * nr_pages < end);
34
35   unsigned long prev_end = end, prev_cur = cur;
36
37   char *result = hyp_early_alloc_contig(nr_pages);
38
39   assert(result != NULL);
40   assert(forall_elem((char *)base, &alloc_range_zero,
41     result - base, result - base + PAGE_SIZE * nr_pages));
42
43   assert(cur == prev_cur + PAGE_SIZE * nr_pages);
44   assert(end == prev_end);
45 }
46
47 void spec__nr_pages() {
48   unsigned long result = hyp_early_alloc_nr_pages();
49   assert(result == (cur - base) / PAGE_SIZE);
50 }
51
52 void spec__init() {
53   any(unsigned long, virt);
54   assume(names_obj((char *)virt, char[NUM_PAGES * PAGE_SIZE]));
55
56   hyp_early_alloc_init(virt, NUM_PAGES * PAGE_SIZE);
57 }
```

pKVM uses the early allocator during initialization, to manage memory allocation over a flat region. The allocator does not support memory reclamation, so it only tracks how much of the region has been allocated. Its state consists of three global long integers: `base` and `end` store the addresses that mark the beginning and end of the allocatable region, and `cur` stores the base address of the next block to be allocated.

`inv__early_alloc` is the global invariant. It states that `base` names the allocatable region, that `end` is equal to `base` plus the size of the buffer, and that `cur` is between `base` and `end`.

`spec__alloc_page` is a POT specifying the functional correctness of the API function `hyp_early_alloc_page`: assuming there is at least one page left to allocate, `hyp_early_alloc_page` will run and terminate without low-level errors and return a non-null pointer. All allocated bytes, which are at offsets between `result-base` and `result-base+PAGE_SIZE` from `base`, will be zero initialized. `cur` will be incremented by `PAGE_SIZE` and `end` will remain unchanged. Stated precisely, line 23 asserts the following: for all elements of the array of characters located at `base`, `alloc_range_zero(i, result-base, result-base+PAGE_SIZE)` returns true, where `i` is the element's index.

`spec__alloc_contig` specifies `hyp_early_alloc_page`, which allocates a number of contiguous pages. The specification closely mimics `spec__alloc_page`.

`spec__nr_pages` specifies the value `hyp_early_alloc_nr_pages` is expected to return, and `spec__init` says that the region used to initialize the allocator must be named by the `virt` pointer argument. Implicitly, `inv__early_alloc` must be established after `hyp_early_alloc_init`, and maintained after the other API functions.

A.2 Verifying `spec__alloc_page`

This section provides a walkthrough of how TPot verifies `spec__alloc_page`. To this end, we present below the body of the API function `hyp_early_alloc_page`, along with the internal function `clear_page`. The former is annotated with a loop invariant (`loopinv_clear_page`), whose body is omitted for brevity.

```
1  void clear_page(void *to) {
2    int i = 0;
3    while(i < 4096) {
4          _tpot_inv(&loopinv_clear_page, &i, &to,
5                          // modified memory
6                &i, sizeof(i), to, PAGE_SIZE);
7      *((char *) to+i) = 0;
8      i++;
9    }
10 }
11
12 void * hyp_early_alloc_page() {
13   unsigned long ret = cur;
14
15   cur += PAGE_SIZE;
16   if (cur > end) {
17     cur = ret;
18         return NULL;
19   }
20   clear_page((void*)ret);
21
22   return (void *)ret;
23 }
```

TPot's symbolic state representation includes a heap, a set of stack variables and globals, a program counter, and a path condition (denoted C). The latter is the set of constraints over initial state required for a concrete execution to follow a particular code path. To verify the POT, TPot (1) adds the constraint the global invariant must hold to C, (2) symbolically executes the POT, forking the state when execution branches, (3) proves that the respective path condition implies each assertion for all resulting states, and (4) proves the global invariant holds over all resulting states.

To achieve (1), TPot computes `inv__early_alloc` into a formula by symbolically executing it and merging the return values of all paths. Concretely, symbolic execution produces five paths (due to the short-circuiting semantics of `&&`), four returning `false` and one returning `true`. TPot then computes a disjunction over all paths $((C_1 \wedge \text{retval}_1) \vee (C_2 \wedge \text{retval}_2) \vee \ldots)$, which in this case simplifies to $\text{names_obj}(\ldots) \wedge \text{end} = \text{base} + \text{NUM_PAGES} \times \text{PAGE_SIZE} \wedge \text{cur} \geq \text{base} \wedge \text{cur} \leq \text{end}$.

We use $\text{names_obj}(\ldots)$ as a shortcut here. In reality, TPot creates a heap object at a fresh address objaddr_base, adding $\text{heap_safe}(\text{tpot_bv2int}(\text{objaddr_base})) = \text{NUM_PAGES} \times \text{PAGE_SIZE}$ to C, and replaces $\text{names_obj}(\ldots)$ with $\text{tpot_bv2int}(\text{base}) = \text{tpot_bv2int}(\text{objaddr_base})$. TPot completes (1) by adding the disjunction to the C, then starts symbolically executing `spec__alloc_page`.

At line 16, TPot makes an SMT query ($C \land \text{cur} + \text{PAGE_SIZE} < \text{end}$) to check that the assumption is feasible, and then adds $\text{cur}+\text{PAGE_SIZE} < \text{end}$ to C. After reaching line 20, symbolic execution follows the control flow into the body of `hyp_early_alloc_page`.

Line 16 in `hyp_early_alloc_page` is a potential branch. TPot determines the feasibility of each side of the branch by making two SMT queries ($C \land \text{cur} + \text{PAGE_SIZE} > \text{end}$ and $C \land \text{cur} + \text{PAGE_SIZE} \leq \text{end}$), where cur denotes the initial value of the global variable. The first SMT query returns UNSAT, indicating that the true branch is infeasible, and so TPot does not fork.

At line 4 in `clear_page`, TPot handles the loop invariant. The body of the loop invariant (`loopinv_clear_page`, omitted) says that the byte at `to + j` is equal to zero for all j between zero and `i`. TPot (a) symbolically executes `loopinv_clear_page` and checks that it must return true, (b) havocs all state modified by the loop (`sizeof(i)` bytes at `&i` and `PAGE_SIZE` bytes at `to`), (c) assumes the loop invariant over the havoced state. Furthermore, for each state that and reaches the invariant again, TPot checks that the invariant is maintained, and that no other state modified than the bytes that were havoced.

At line 7, TPot performs pointer resolution to determine whether writing through `((char *)to+i)` is memory safe, and if so, to enumerate the objects it might point to. The memory safety check takes the form of an SMT query ($C \land (\exists b.\text{tpot_bv2int}(\text{to}) + \text{tpot_bv2int}(i) \geq b \land \text{tpot_bv2int}(\text{to}) + \text{tpot_bv2int}(i) < b + \text{heap_safe}(b))$), which succeeds due to the naming-related constraints added earlier to C. Enumerating target objects generally entails making one SMT query per heap object to determine whether it is feasible for the pointer to fall in bounds of the object, and forking a state for each such object. In this example, the heap contains a single object and so TPot does not need to fork.

When symbolic execution loops back to line 3, TPot forks the state, since the value of `i` is havoced, making both sides of the branch ($i + 1 < 4096$ and $i + 1 \geq 4096$) feasible. The $<$ branch reaches the invariant again, and is terminated after the checks described above.

The $\geq$ branch exits the loop, returns from `clear_page` and `hyp_early_alloc_page`, and reaches line 23 in `spec__alloc_page`. Importantly, the path condition for this state includes the loop invariant, and $i + 1 \geq 4096$, implying that the byte at `to + j` is equal to zero for all j between zero and `4096`. TPot maintains this fact as a `forall_elem` property, explicitly instantiated when the heap object named by `base` is read.

To handle the `forall_elem` on line 40, TPot checks whether `alloc_range_zero(k, result-base, result-base+PAGE_SIZE)` is provable for any `k`. To this end, TPot symbolically executes the body of `alloc_range_zero` in the current state with a fresh `k`. During this process, at line 11, the read on `base` triggers an instantiation of the `forall_elem` property resulting from the earlier loop invariant. This lets TPot prove that `alloc_range_zero` must return true.

TPot proves the rest of the assertions in `spec__alloc_page` with an SMT query each. For each `assert(cond)`, TPot expects an UNSAT result for the query ($C \land \neg\text{cond}$).

Lastly, TPot achieves (4) by symbolically executing `inv__early_alloc` over the final state and checking that all feasible paths must return true. In this case, the path condition makes all short-circuiting paths infeasible, and the only feasible path returns $\text{cur} + \text{PAGE}_\text{S}\text{IZE} \leq \text{end}$. TPot determines through an SMT query that this must be true, due to the earlier assumption (on line 16) that added $\text{cur} + \text{PAGE_SIZE} < \text{end}$ to C.