Verification with Small and Short Worlds

Rohit Sinha
UC Berkeley

Cynthia Sturton
UC Berkeley

Petros Maniatis
Intel Labs

Sanjit A. Seshia
UC Berkeley

David Wagner
UC Berkeley

Abstract—**We consider the verification of safety properties in systems with large arrays and data structures. Such systems are common at the low levels of software stacks; examples are hypervisors and CPU emulators. The very large data structures in such systems (e.g., address-translation tables and other caches) make automated verification based on straightforward state-space exploration infeasible. We present S^{2}W, a new abstraction-based model-checking methodology to facilitate automated verification of such systems. As a first step, inductive invariant checking is performed. If that fails, we compute an abstraction of the original system by precisely modeling only a subset of state variables while allowing the rest of the state to evolve arbitrarily at each step. This subset of the state constitutes a "small world" hypothesis, and is extracted from the property. Finally, we verify the safety property on the abstract model using bounded model checking. We ensure the verification is sound by first computing a bound on the reachability diameter of the abstract model. For this computation, we developed a set of heuristics that we term the "short world" approach. We present several case studies, including verification of the address translation logic in the Bochs x86 emulator, and verification of security properties of several hypervisor models.**

I. Introduction

CPU emulators lie in the foundational layer of much of today's computing infrastructure: CPU emulation is used by virtualization software (hypervisors and virtual machine monitors) in both testing and production to secure, analyze, and multiplex critical systems [7], [15], [28]. Unfortunately, although critical, CPU emulators are not often verified, and are frequently found incorrect [19], [20] or – worse – vulnerable to attacks [4].

A particular challenge for the verification of CPU emulators and hypervisors is their use of large data structures. For example, logical-to-physical address translation requires data structures to store the CPU's Translation Look-aside Buffer (TLB) and page tables. While these structures are finite-length for any given processor, they are usually too large to represent precisely for verification; often, they are abstracted to be of unbounded length. The data structures in the resulting model of the system are thus *parametrized*: the indices into those structures are *parameters*, taking values in a very large or even infinite domain (typically finite-precision bit-vectors or the integers). The techniques proposed for verifying such parametrized systems fall into two classes: those based on a small-model or cut-off theorem (e.g., [11], [12], [24]), or those based on abstraction (e.g., [8], [14], [18]). While existing approaches are elegant and effective for their respective problem domains, they fall short for the problems we consider: the small-model approaches usually restrict expressiveness, while abstraction-based approaches either focus on control properties (as opposed to equivalence/refinement) or handle only certain kinds of data structures. In both cases, some of the realistic case studies we consider cannot be handled. (We make a fuller comparison in Sec. VI.)

In this paper, we present a new semi-automatic methodology for verifying safety properties in systems with large data structures. Our approach comprises three steps. First, we employ standard (mathematical) induction to verify the safety property, and if that succeeds, the process is complete. Second, if induction fails, we create an over-approximate abstraction of the system, the "small world," in which unbounded data structures are parametrized and, in general, only a subset of the state is updated as per the original transition relation (e.g., only a few entries of the unbounded data structures); the rest of the state is updated with arbitrary values at each step. With this abstraction, the model is more amenable to state-space exploration. Third, we attempt to find a bound k on the reachability diameter of the small world so that, if bounded model checking for k steps succeeds in the small world, then the safety property must hold in the small world, and since that is an over-approximation of the original system model, then the safety property holds there as well. Heuristics are presented for finding k that are effective for the class of systems we consider. We term this BMC-based approach the "short world" method, since it relies on computing a "short" bound for BMC. Our overall approach, termed *Small-Short-World (S^{2}W)*, is implemented on top of the UCLID system [9], which verifies abstract term-level models using satisfiability modulo theories (SMT) solving. Note that the temporal safety verification problem for our class of systems is undecidable. As a result, S^{2}W is a semi-decision procedure.

In summary, the novel contributions of this paper include:

- A semi-automatic procedure, S^{2}W, for verifying systems with large or unbounded data structures, using a combination of induction and abstraction-based model checking. The key new ideas are a set of heuristics for creating an abstract model and computing a bound on the reachability diameter of its state space.
- An extensive evaluation of S^{2}W on a wide range of CPU emulator and hypervisor models, proving safety properties critical to the security and correctness of those systems. Our examples include the TLB implementation in the Bochs x86 emulator [23] and a shadow page table system.

II. Running Example

We introduce here a running example: a simple read-only memory system with a single-entry cache. We prove an invariant about the value returned by a read command. We build a model in our modeling language and demonstrate the verification of the safety property using S^{2}W. Our example is meant to be small, understandable, and illustrative, rather than "real-world."

Our example system (Fig. 1) takes only one command, *read*, with a single parameter, the 32-bit address to be read; it returns a single-bit data value. At each read command, the cache is

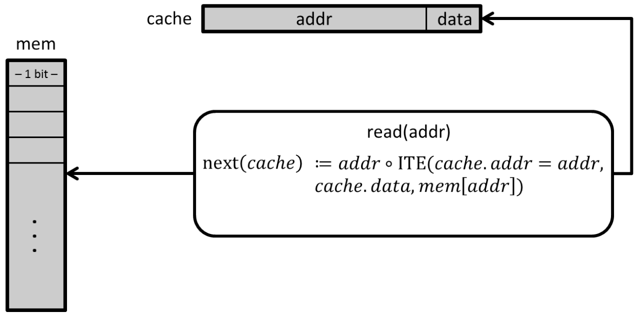
cache
addr
data
mem
– 1 bit –
read(addr)
next(cache) := addr ∘ ITE(cache.addr = addr, cache.data, mem[addr])

Fig. 1: An illustration of our running example: A read-only memory and a single-entry cache. The cache is updated on each read command.

first checked. If the cache contains the data for the address requested, that value is returned. Otherwise, the value is read from memory. In either case, the cache is updated with the requested address and the returned data value. The update to cache is shown in the above figure (we use "$\circ$" to mean concatenation). We prove an invariant about the cache: if the cache holds a valid address, then the cached data value is equal to the value stored in memory at that address. In other words, we show that the cache is correct.

III. Formal Description of Problem

A. Notation and Terminology

A system is modeled as a tuple $\mathcal{S}$ = ($\mathcal{I}$, $\mathcal{O}$, $\mathcal{V}$, *Init*, $\mathcal{A}$) where

- $\mathcal{I}$ is a finite set of input variables;
- $\mathcal{O}$ is a finite set of output variables;
- $\mathcal{V}$ is a finite set of state variables;
- *Init* is a set of initial states, and
- $\mathcal{A}$ is a finite set of assignments to variables in $\mathcal{V}$. Assignments define how state variables are updated, and thus define the transition relation of the system.

Input and output variables are assumed combinational (stateless), without loss of generality. $\mathcal{V}$ is the only set of state-holding variables. Variables can be of two types: *primitives*, such as Boolean or bit-vector; and *memories*, which includes arrays, content-addressable memories (CAMs), and tables. An output variable is a function of $\mathcal{V} \uplus \mathcal{I}$. When representing a system without outputs, we will omit $\mathcal{O}$ from the representation. The set of initial states, *Init*, can either be viewed as a symbolic vector of terms representing any initial state, or as a Boolean-valued function of $\mathcal{V}$, written $Init(\mathcal{V})$.

Fig. 2 denotes the grammar for expressions in our modeling language. The language has three expression types: Boolean, bit-vector, and memory.

$$\begin{aligned}
bE \quad &::= \mathbf{true} \mid \mathbf{false} \mid b \mid \neg bE \mid bE_1 \vee bE_2 \\
&\mid bE_1 \wedge bE_2 \mid bvE_1 = bvE_2 \mid \texttt{bvrel}(bvE_1, \ldots, bvE_k) \quad (k \geq 1) \\
&\mid UP(bvE_1, \ldots, bvE_k) \quad (k \geq 0) \\
bvE \quad &::= c \mid v \mid \texttt{ITE}(bE,\, bvE_1,\, bvE_2) \\
&\mid \texttt{bvop}(bvE_1, \ldots, bvE_k) \ (k \geq 1) \\
&\mid mE(bvE_1, \ldots, bvE_l) \mid UF(bvE_1, \ldots, bvE_k) \quad (l \geq 1, k \geq 0) \\
mE \quad &::= A \mid M \mid \lambda(x_1, \ldots, x_k).bvE \quad (k \geq 0)
\end{aligned}$$

Fig. 2: Expression Syntax. c and v denote a bit-vector constant and variable, respectively, and b is a Boolean variable. bvop denotes any arithmetic/bitwise operator mapping bit vectors to bit vectors, while bvrel is a relational operator other than equality mapping bit vectors to a Boolean value. UF and UP denote an uninterpreted function and predicate symbol respectively. A and M denote constant and variable memories. $x_1, \ldots, x_k$ denote parameters (typically indices into memories) that appear in *bvE*.

The simplest Boolean expressions (*bE*) are the constants **true** and **false** or Boolean variables; more complicated expressions can be constructed using standard Boolean operators or using relational operators amongst bit-vector expressions. We also allow a Boolean expression to be an application of an uninterpreted predicate to bit-vector expressions.

Bit-vector expressions (*bvE*) include bit-vector constants, variables, if-then-else expressions (ITE), and expressions constructed using standard bit-vector arithmetic and bitwise operations. Additionally, bit-vector expressions can be constructed as applications of uninterpreted functions returning bit-vector values and applications of memories to bit-vector arguments. Each bit-vector expression has an associated bitwidth.

Finally, the primitive memory expressions (*mE*) can be (symbolic) constants or variables. More complex memory expressions can be modeled using the Lambda notation introduced by Bryant et al. [9] for term-level modeling; this includes the standard **write** (**store**) primitive for modeling arrays, as well as more general operations such as parallel updates to arrays, operations on CAMs, queues, and other data structures.

In addition to the above expressions, we will use the wildcard "$*$" to denote an arbitrary value of the appropriate type; it is used primarily to express non-determinism in the state update.

A *next-state assignment* α denotes assignment to a state variable and is a rule of the form $\texttt{next}(x) := e$ or $\texttt{next}(x) := \{e_1, e_2, \ldots, e_n\}$, where x is a signal in $\mathcal{V}$, and $e, e_1, e_2, \ldots, e_n$ are expressions that are a function of $\mathcal{V} \uplus \mathcal{I}$. The curly braces and "$*$" express non-deterministic choice. The set of all next-state assignments defines the *transition relation* $\mathcal{R}$ of the system. Formally, $\mathcal{R} = \bigwedge_{\alpha \in \mathcal{A}} r(\alpha)$, where $r(\texttt{next}(x) := e) \doteq (x' = e)$ and $r(\texttt{next}(x) := \{e_1, e_2, \ldots, e_n\}) \doteq \bigvee_{i=1}^{n} (x' = e_i)$, where x' denotes the next-state version of variable x. The wildcard "$*$" is translated at each transition into a fresh symbolic constant of the appropriate type. We will sometimes write the transition relation as $\mathcal{R}(\mathcal{V}, \mathcal{I}, \mathcal{V}')$ to emphasize that it relates current-state variables $\mathcal{V}$ and next-state variables $\mathcal{V}'$ based on the inputs $\mathcal{I}$ received.

Example 1: We formally describe our model from Sec. II. Let $\mathcal{S}_T$ = ($\mathcal{I}$, $\mathcal{O}$,$\mathcal{V}$, *Init*, $\mathcal{A}$) be the system, with

- $\mathcal{I} = \{addr\}$. *addr* is the 32-bit address to read from memory.
- $\mathcal{O} = \{out\}$. *out* is the value read from either memory or the cache.
- $\mathcal{V} = \{mem, cache\}$. *mem* is constant and is modeled as an array of (one-bit) bit-vectors. It is represented by an uninterpreted function that maps a 32-bit address to a single bit. *cache* is a 33-bit vector; it holds the one-bit data value and 32-bit address of that value.
- $Init = (mem_0, cache_0)$. *mem* is initialized to hold arbitrary data values at each address. *cache* is initialized to hold an invalid address, 0x00000000, with an arbitrary data value.
- $\mathcal{A}$. On each read command *cache* is updated with the address read and the value returned by the read; *mem* remains constant.

B. Problem Definition

Consider a system $\mathcal{S}$ modeled as described in the preceding section. We similarly model the environment $\mathcal{E}$ that provides the inputs for $\mathcal{S}$ and consumes its outputs. The composition of $\mathcal{S}$ and $\mathcal{E}$, written $\mathcal{S}\|\mathcal{E}$, is the model under verification, $\mathcal{M}$. The form of the composition depends on the context; we use both synchronous and asynchronous compositions. We will represent the closed system $\mathcal{M}$ as a transition system $(\mathcal{V}_{\mathcal{M}}, \mathit{Init}_{\mathcal{M}}, \mathcal{R}_{\mathcal{M}})$, where the elements respectively denote state variables, initial states, and transition relation. In all of our examples, the environment $\mathcal{E}$ is stateless, generating completely arbitrary inputs to $\mathcal{S}$ at each step; thus $\mathcal{V}_{\mathcal{M}} = \mathcal{V}$, $\mathit{Init}_{\mathcal{M}} = \mathit{Init}$ and $\mathcal{R}_{\mathcal{M}} = \mathcal{R}$.

This paper is concerned with verification of temporal safety properties of the form $\mathbf{G}\Phi$, where $\mathbf{G}$ is the temporal operator "always" and Φ is a state invariant of the form

$$\forall x_1, \ldots, x_k.\ \phi(x_1, \ldots, x_k) \tag{1}$$

where ϕ is a Boolean expression following the syntax of *bE*. The parameters $x_1, \ldots, x_k$ are bit-vector valued, but usually too large to exhaustively case split on.

Example 2: In our running example, we verify $\mathbf{G}\Phi_2$, where

$$\begin{aligned}\Phi_2 \doteq\ & \forall x.\ (\mathit{addr} = x) \rightarrow \\ & \quad ((\mathit{cache.addr} = \mathit{addr} \wedge \mathit{cache.addr} \neq 0) \rightarrow \\ & \quad\quad \mathit{cache.data} = \mathit{mem}[\mathit{addr}])\end{aligned} \tag{2}$$

The problem tackled by this paper, temporal safety verification for systems with large data structures, is formally defined as follows.

Definition 1 (Large Data Safety Verification): Given a model $\mathcal{M}$ formed as a composition of system $\mathcal{S}$ and its environment $\mathcal{E}$, and a temporal safety property $\mathbf{G}\Phi$, determine whether or not $\mathcal{M}$ satisfies $\mathbf{G}\Phi$. □

This problem is known to be undecidable in general since a two-counter machine can be encoded in our formalism using applications of uninterpreted functions [16]. Hence, we can only devise a semi-decision procedure for the problem. In the next section, we describe such a procedure that is based on abstraction.

IV. Methodology

S^{2}W is based on a combination of abstraction and Bounded Model-Checking (BMC). We tackle state-space explosion by abstracting away all but a small subset of the space of the system. We call this mostly-abstracted system our "small world." The abstracted portion of the system can be considered as being updated with an arbitrary value ("$*$") at each step of execution. All other parts of the system are modeled precisely. Thus, this abstraction is a form of localization abstraction [17], where the localization is to small, finite portions of large data structures.

We check the safety property on the small world using BMC. To make BMC sound, we first find and prove the length of the diameter D of our small world to use as the bound – i.e., D is an integer such that every state reachable in $D+1$ steps is also reachable in D or fewer steps. Proving that a conjectured diameter D is correct is undecidable in our formalism [10]. The key to our approach is a set of heuristics that are effective in our chosen application domain of emulators and hypervisors. For our examples, the diameter of the mostly-abstracted system is typically small; we therefore term this the "short world."

If BMC runs for D steps and does not find a violation of the safety property in our small world, then the original model is safe. If BMC finds a counter-example, we cannot say whether the property holds for the original model: BMC can return a spurious counter-example. Choosing the small world well reduces the likelihood of finding spurious counter-examples.

To summarize, there are two crucial pieces to our approach: choosing the right small world and proving the length of the short world. We discuss both of these in more detail below.

As an optimization, we prefix the above approach with an attempt to prove the safety property using one-step induction (on the original, non-abstract model, $\mathcal{M}$). If that succeeds, there is no need to continue on to S^{2}W's abstraction. (This step can be generalized to perform k-step induction as needed.)

For the presentation in this section, it is convenient to represent the system under verification $\mathcal{S}$ as a transition system $(\mathcal{I}, \mathcal{V}, \mathcal{R}, \mathit{Init})$ where the elements of the tuple have the same meanings as in Sec. III. The environment $\mathcal{E}$ sets the values of the input variables in $\mathcal{I}$ at each step; in all our case studies, the inputs from $\mathcal{E}$ are completely unconstrained. Verification (using induction or BMC) is performed on the composition of $\mathcal{S}$ and $\mathcal{E}$.

A. Induction

First, S^{2}W attempts to prove the safety property using simple one-step induction on the non-abstract model $\mathcal{M}$. We check the validity of the following two formulas, as per standard practice:

$$\mathit{Init}_{\mathcal{M}}(\mathcal{V}_{\mathcal{M}}) \rightarrow \Phi(\mathcal{V}_{\mathcal{M}}) \tag{3}$$

$$\Phi(\mathcal{V}_{\mathcal{M}}) \wedge \mathcal{R}_{\mathcal{M}}(\mathcal{V}_{\mathcal{M}}, \mathcal{V}'_{\mathcal{M}}) \rightarrow \Phi(\mathcal{V}'_{\mathcal{M}}) \tag{4}$$

If both checks pass, the verification is complete. We report "Property valid" and exit. If check (3) fails, the property is invalid. We report "Property invalid in initial state" and exit. If check (4) fails, we continue with S^{2}W, to find the small world.

B. Small World

The objective of this step is to identify a small portion of system state that we should model precisely during BMC. Everything else will be allowed to take on arbitrary values at each step of execution.

It is important to note that the soundness of S^{2}W does *not* depend on the choice we make for the small world; we could randomly select some portion of the state to model precisely, abstract everything else away, and if our three steps complete and verify the property, the property would be true of the original, non-abstracted system. However, choosing the small world wisely ensures that the short world is indeed short, which allows BMC to complete in a reasonable amount of time. A well-chosen small world also reduces the amount of spurious counter-examples returned by the BMC step.

We present here a heuristic for choosing the small world when dealing with systems involving large or unbounded data structures. In our case studies, the heuristic found a small world whose short world was reasonable in length and for which no spurious counter-examples were returned by the BMC.

To select those state variables to model precisely, S^{2}W starts with the property $\mathbf{G}\ \Phi$, where Φ is of the form $\forall x_1, x_2, \ldots, x_n.\ \phi(x_1, x_2, \ldots, x_n)$. If we prove Φ by instantiating the quantifier with a completely arbitrary, symbolic parameter vector $(a_1, a_2, \ldots, a_n)$, that suffices to prove the original property. Thus, starting with the symbolic vector $(a_1, a_2, \ldots, a_n)$, we compute a *dependence set* ($\mathcal{U}$) for the instantiated property $\phi(a_1, a_2, \ldots, a_n)$. $\mathcal{U}$ is a *set of expressions* involving state variables and the parameters $a_1, \ldots, a_n$ such that fixing the values of the expressions in this set fixes the value of the instantiated property. For variable M modeling a memory, these expressions typically involve indexing into M at a finite number of (symbolic) addresses. For a Boolean or bit-vector variable, either the variable is in $\mathcal{U}$ or not.

Typically, this set of expressions is derived syntactically by traversing the expression graph of the formula ϕ represented in terms of state and input variables (after performing certain simplifications).

Example 3: In our running example, recall that the property is $\mathbf{G}\ \Phi_2$ where:

$$\begin{aligned}\Phi_2 \doteq\ & \forall x.\ (\mathit{addr} = x) \rightarrow \\ & \qquad ((\mathit{cache.addr} = \mathit{addr} \wedge \mathit{cache.addr} \neq 0) \rightarrow \\ & \qquad\quad \mathit{cache.data} = \mathit{mem}[\mathit{addr}])\end{aligned}$$

Φ_2 has the form $\forall x.\ \phi(x)$. Instantiating x with a, a fresh symbolic constant, we can drop the quantifer and get $\phi(a)$, for which, by propagating the equality $\mathit{addr} = a$, we see that its value is determined by the expressions $\mathit{mem}[a]$ and *cache*. Thus, we use $\mathcal{U} = \{\mathit{mem}[a], \mathit{cache}\}$ as our dependence set.

Once we have computed $\mathcal{U}$, using the above heuristic or some other method, we can define our small world. Recall that $\mathcal{S}$ is represented as a symbolic transition system $(\mathcal{I}, \mathcal{V}, \mathcal{R}, \mathit{Init})$. Let $\hat{\mathcal{R}}$ be a transition relation that differs from $\mathcal{R}$ by setting all state not in $\mathcal{U}$ to a non-deterministic value and leaving all others unchanged. Abusing notation slightly to use $\mathcal{U}$ wherever we use $\mathcal{V}$, this means that $\hat{\mathcal{R}}(\mathcal{U}, \mathcal{I}, \mathcal{U}') = \mathcal{R}(\mathcal{U}, \mathcal{I}, \mathcal{U}')$, and $\hat{\mathcal{R}}(\mathcal{W}, \mathcal{I}, \mathcal{W}') = \mathbf{true}$ for $\mathcal{W} = \mathcal{V} \setminus \mathcal{U}$. Similarly, $\hat{Init}(\mathcal{U}) = \mathit{Init}(\mathcal{U})$ and $\hat{Init}(\mathcal{W}) = \mathbf{true}$.

Then the abstracted small world is $\hat{\mathcal{S}} = (\mathcal{I}, \mathcal{V}, \hat{\mathcal{R}}, \hat{Init})$. $\hat{\mathcal{S}}$ is an overapproximate (abstract) version of $\mathcal{S}$ that precisely tracks only the state in $\mathcal{U}$, and allows all other variables to change arbitrarily at each step of execution. Thus, the composition of $\hat{\mathcal{S}}$ and $\mathcal{E}$ is an overapproximate model $\hat{\mathcal{M}}$. (Note that if $\mathcal{M}$ was infinite-state, $\hat{\mathcal{M}}$ is too.)

The next step is proving a short world for $\hat{\mathcal{M}}$ and using BMC on $\hat{\mathcal{M}}$ to verify the property.

C. Short World

The objective of this phase is to determine a bound on the diameter D of the abstract model $\hat{\mathcal{M}}$. For this section, we will assume that $\mathcal{E}$ is stateless, as is the case for all of our case studies; the approach extends in a straightforward manner for the general case. Thus, the diameter of $\hat{\mathcal{M}}$ is the same as that of $\hat{\mathcal{S}}$.

Suppose we believe the diameter to be $\leq k$. To verify this bound, we check the validity of the following logical formula:

$$\begin{aligned}& \forall \mathcal{V}_0, \mathcal{V}_1, \ldots, \mathcal{V}_{k+1}, \mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_{k+1}. \\ & \quad \big[\mathit{Init}(\mathcal{V}_0) \wedge \bigwedge_{i=0}^{k} \hat{\mathcal{R}}(\mathcal{V}_i, \mathcal{I}_{i+1}, \mathcal{V}_{i+1})\big] \\ & \rightarrow \big[\exists \mathcal{V}'_0, \mathcal{V}'_1, \ldots, \mathcal{V}'_k, \mathcal{I}'_1, \mathcal{I}'_2, \ldots, \mathcal{I}'_k. \\ & \quad \mathit{Init}(\mathcal{V}'_0) \wedge \bigwedge_{i=0}^{k-1} \hat{\mathcal{R}}(\mathcal{V}'_i, \mathcal{I}'_{i+1}, \mathcal{V}'_{i+1}) \wedge \bigvee_{i=0}^{k} \mathcal{V}_{k+1} = \mathcal{V}'_i\big]\end{aligned} \tag{5}$$

Since $\hat{\mathcal{R}}$ modifies state expressions outside $\mathcal{U}$ arbitrarily on each step, we can replace $\mathcal{V}$ everywhere in the above formula with $\mathcal{U}$, and obtain the actual convergence criterion that must be checked.

Nevertheless, checking the convergence criterion is undecidable for the class of systems we are interested in, due to the presence of uninterpreted functions, memories, and possibly parameters with unbounded bitwidth [10]. The quantified formula in (5) is also very hard to solve in practice. Therefore, quantifier instantiation heuristics must be devised to perform the convergence check. In this section, we present two such heuristics that have worked well for the range of case studies considered in this paper.

The Sub-Sequence Heuristic: The first heuristic checks that for any state reachable in $k+1$ steps using $k+1$ symbolic inputs to $\hat{\mathcal{S}}$, one can also reach that state using *some sub-sequence of length* $\leq k$ of those $k+1$ symbolic inputs. We can express the sub-sequence heuristic as performing a particular instantiation of the existential quantifiers in criterion (5), and checking the validity of the following formula that results:

$$\begin{aligned}& \forall \mathcal{U}_0, \mathcal{U}_1, \ldots, \mathcal{U}_{k+1}, \mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_{k+1}, \mathcal{U}'_0, \mathcal{U}'_1, \ldots, \mathcal{U}'_k. \\ & \quad \big[\mathit{Init}(\mathcal{U}_0) \wedge (\mathcal{U}_0 = \mathcal{U}'_0) \wedge \bigwedge_{i=0}^{k} \hat{\mathcal{R}}(\mathcal{U}_i, \mathcal{I}_{i+1}, \mathcal{U}_{i+1})\big] \rightarrow \\ & \bigvee_{(\mathcal{I}'_1, \ldots, \mathcal{I}'_k) \prec (\mathcal{I}_1, \ldots, \mathcal{I}_{k+1})} \left[\bigwedge_{i=0}^{k-1} \hat{\mathcal{R}}(\mathcal{U}'_i, \mathcal{I}'_{i+1}, \mathcal{U}'_{i+1}) \wedge \bigvee_{i=0}^{k} \mathcal{U}_{k+1} = \mathcal{U}'_i\right]\end{aligned} \tag{6}$$

Here the symbol $\prec$ denotes that $(\mathcal{I}'_1, \ldots, \mathcal{I}'_k)$ is a sub-sequence of $(\mathcal{I}_1, \ldots, \mathcal{I}_{k+1})$.

The intuition for the sub-sequence heuristic is that in many systems with large arrays and tables, locations in those tables are updated destructively based on the current input, meaning that past updates do not matter. The address translation logic in the emulators we have studied has this nature. Thus, for such systems, it is possible to drop from the input sequence inputs that have no effect on the $k+1$-st step.

Observe the quantifier alternation in criterion (5) has been eliminated in the stronger criterion (6). Thus, we can simply perform a validity check of an SMT formula in the combination of theories required by our model. If the sub-sequence criterion (6) holds, then so does (5). However, it is possible that criterion (6) is too strong, even when a short diameter exists. This scenario necessitates an alternative semi-automatic approach, described next.

The Gadget Heuristic: The *gadget heuristic* is an approach to instantiating the existential quantified variables in criterion (5) that is particularly useful for systems in which some state in $\mathcal{U}$ depends on the past history of state updates in a non-trivial manner. A gadget is a small sequence of state transitions manually constructed to generate some subset of all reachable system state. A *universal gadget set* is a set of such sequences that, in concert, can generate any reachable system state[1]. The length k of the longest gadget in the universal gadget set is then an upper bound on the diameter of the system.

In terms of the formula expressed as criterion (5), a gadget is a particular guess for a set of initial states $\mathcal{V}'_0$ (expressed symbolically) and a sequence $\mathcal{I}'_1, \mathcal{I}'_2, \ldots, \mathcal{I}'_l$ (for $l \leq k$) of symbolic input expressions to use. For a finite number of gadgets, the inner existential quantifier in criterion (5) can be replaced as a disjunction over all the formulas obtained by substituting the gadget expressions for $(\mathcal{V}'_0, \mathcal{I}'_1, \mathcal{I}'_2, \ldots, \mathcal{I}'_l)$. If this instantiated formula is valid, then so is the original formula (5).

We defer further discussion about gadget construction to Sec. IV-D, where we discuss its use on our running example.

Performing BMC: Once we have proven k is an upper bound on the length of the diameter of $\hat{\mathcal{S}}$, we run BMC on $\hat{\mathcal{S}}$ for k steps. If $\phi(a_1, \ldots, a_n)$ holds at each step of the simulation, then it follows that $\hat{\mathcal{S}}$ satisfies $\mathbf{G}\ \Phi$. Because $\hat{\mathcal{S}}$ is an overapproximation of all states reachable by $\mathcal{S}$, it follows that $\mathcal{S}$ satisfies $\mathbf{G}\ \Phi$.

If BMC fails, we return a "short" counter-example. The counter-example will be no longer than k. If this is a valid counter-example, the property does not hold. If it is a spurious counter-example, we can return to step two of S^{2}W and expand our set $\mathcal{U}$ to include more state variables and inputs. Such a strategy would be an instance of counterexample-guided abstraction refinement.

Restricted State Spaces: In some systems, we are interested in proving a safety property over a restricted state space, where the restriction can be captured by a predicate over state variables. The restriction predicate is often specified as an antecedent in the temporal safety property. Examples of such a restriction can be found in Sections V-A and V-B. In such cases, we note that it is enough to compute a bound on the reachability diameter — the short world bound — under that restriction. It also sufficient to perform model checking under this restriction.

D. Example

To illustrate the above approach, we apply it to our example. In step one we attempt to prove property $\mathbf{G}\ \Phi_2$ by induction. For this, we perform the following two checks:

$$\mathit{Init}(\mathit{mem}, \mathit{cache}) \rightarrow \Phi_2 \tag{7}$$

$$\Phi_2 \wedge \mathcal{R}_T(\mathcal{V}, \mathcal{V}') \rightarrow \Phi_2 \tag{8}$$

Check (7) passes, since the cache is initially empty. However, the induction step (check (8)) does not pass. Starting from a state in which Φ_2 holds, it is possible to transition to a state in which Φ_2 is violated. To see why this is so, consider the following state for the cache and two particular entries of *mem*:

$$\mathit{mem}[i] := a,\ \mathit{mem}[j] := b,\ \mathit{cache.addr} := i,\ \mathit{cache.data} := z$$

where $z \neq a$, the last read was for address j, and the output was b. (This state is not reachable in our model, but one-step induction does not take this into account.) Note that Φ_2 holds in this state: for every $x \neq j$, the antecedent ($\mathit{addr} = x$) of the property is false and therefore the property is true; when $x = j$, the nested antecedent ($\mathit{cache.addr} = \mathit{addr} \wedge \mathit{cache.addr} \neq 0$) is false and therefore the property is true. In this state, a $\mathit{read}(i)$ command will hit in the cache and the output will be z, making the property evaluate to false in the next state.

Since simple induction failed for our toy example, we move to the next step, identifying the small world $\hat{\mathcal{S}}_T$. As described in Section IV-B, we introduce a fresh symbolic constant a for x, removing the $\forall x$ quantifier from the property. We then select $\mathcal{U}$ syntactically from the property to be the set of expressions $\mathcal{U} = \{\mathit{mem}[a], \mathit{cache}\}$. In $\hat{\mathcal{S}}_T$ the variables in $\mathcal{U}$ are updated according to the original model ($\mathcal{S}_T$). All other state variables (all entries of *mem* other than $\mathit{mem}[a]$) are made to be fully abstract: they are allowed to update to non-deterministic values on every step. The same symbolic constant is used throughout the following short world checks.

The last step of our verification is to identify a short world and then run BMC on the abstract model for the length of the short world. We describe the gadget heuristic here; the sub-sequence heuristic would also work, although it finds a slightly looser bound on the length of the diameter. To build the gadgets we enumerate the possible end-state valuations for the system's state variables (*cache*, $\mathit{mem}[a]$) and for each, determine how to get there from a possible starting state. Notice that we only need to consider $\mathit{mem}[a]$ and not all of *mem*. This is because, in our small world $\hat{\mathcal{S}}_T$, all entries of *mem* other than $\mathit{mem}[a]$ receive new arbitrary values at the end of each step, so we know they can hold any possible value at every step of any trace. In theory there are 2^{34} end-states: one for each possible value of $\mathit{cache.addr}$ times the two possible values of $\mathit{cache.data}$ and $\mathit{mem}[a]$ each. However, for our property, we do not really care about the precise valuation of $\mathit{cache.addr}$, rather, we care about whether $\mathit{cache.addr} = a$ and whether $\mathit{cache.addr} = 0$. So we can abstract away the details of $\mathit{cache.addr}$ and consider the following 16 ending states:

$$\begin{aligned}&\{\mathit{cache.addr} = a, \mathit{cache.addr} \neq a\}\\ &\quad \times \{\mathit{cache.addr} = 0, \mathit{cache.addr} \neq 0\}\\ &\quad \times \{\mathit{cache.data} = 0, \mathit{cache.data} = 1\}\\ &\quad \times \{\mathit{mem}[a] = 0, \mathit{mem}[a] = 1\}\end{aligned}$$

Not all of the above 16 states are reachable, and in the end four gadgets are enough to reach all reachable states. Each gadget uses either one or two read commands. We build the gadgets with the appropriate values for *addr* and show they form a universal gadget set and therefore, that the short world

[1] Our gadgets are inspired by "state-generation gadgets," used for automated testing of CPU emulators from arbitrary but reachable initial states [19], and by gadgets identified for return-oriented programming, used to produce a Turing-complete command set for malicious exploits [26].

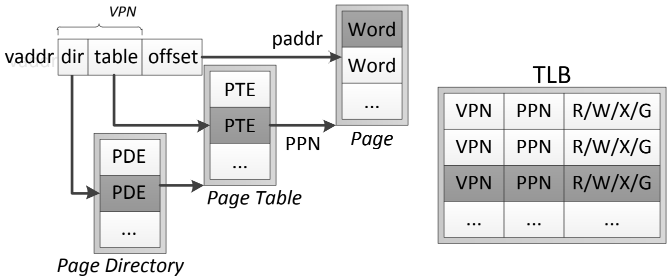

Fig. 3: On the left, we show a two-level page walk translating VPN to PPN addresses. The TLB caches VPN to PPN translations along with read/write/execute/global permission bits.

has length two. We then perform BMC and verify that the property holds.

V. Evaluation

We have evaluated S^{2}W on six case studies and describe them here: the TLB of the Bochs x86 emulator, a set-associative cache, shadow paging in a hypervisor, hypervisor integrity for SecVisor [12], the Chinese Wall access-control policy in sHype [12], and separation in the original version of ShadowVisor [11]. We describe the first three in detail; the last three were verified using one-step induction and we describe them only briefly. The code for all of our models, along with their verification, is available online.[2] All experiments were performed using UCLID [3] (with the Plingeling SAT solver [1] backend) on a machine with 8 Intel Xeon cores and 4 GB RAM.

A. Bochs' TLB

Bochs [23] is an open source x86 emulator (in C++) for emulating CPU, BIOS, and I/O peripherals. Bochs emulates virtual memory using paging, which includes logic to translate a virtual address (VPN) to a physical address (PPN). Figure 3 illustrates the steps of a page walk. The input virtual address *vaddr* is partitioned into 3 sets of bits $(\mathit{vaddr}_{\mathit{dir}}, \mathit{vaddr}_{\mathit{table}}, \mathit{vaddr}_{\mathit{offset}})$. First, the $\mathit{vaddr}_{\mathit{dir}}$ bits index a page directory entry (PDE) within the page directory region. The PDE contents, along with the $\mathit{vaddr}_{\mathit{table}}$ bits, index into the page tables to retrieve a page table entry (PTE). The PTE contents identify a 4KB physical page, and when concatenated with the 12 bit $\mathit{vaddr}_{\mathit{offset}}$ index a particular byte within this page. Since the above page walk includes two memory lookups, most x86 processors implement a TLB to cache VPN to PPN translations. The TLB also caches permission bits (r/w/x/g) checked during memory accesses. With this optimization, Bochs' address translation logic first checks its TLB for an entry describing the wanted VPN. If no such entry exists, Bochs performs a page walk to compute the corresponding PPN, and then stores that translation in its TLB for future accesses. We would like to prove that the optimized paging unit (with Bochs TLB) is functionally equivalent to the original paging unit (without TLB).

The Bochs TLB + page table system is modeled as a tuple $\mathcal{S}_{\mathit{Bochs}}$ = ($\mathcal{I}$, $\mathcal{O}$, $\mathcal{V}$, Init, $\mathcal{A}$) where

- $\mathcal{I} = \{\mathit{vaddr}, \mathit{data}, \mathit{pl}, \mathit{rwx}, \mathit{command}\}$. *vaddr* is the virtual address to translate. *data* is used to update the page table memory. *pl* indicates the CPU's current privilege level (either user or supervisor mode). *rwx* indicates whether this memory access writes and/or executes this address.
- $\mathcal{O} = \{\mathit{paddr\_TLB}, \mathit{pagefault\_TLB}, \mathit{paddr\_noTLB}, \mathit{pagefault\_noTLB}\}$. *paddr_TLB* is the result of address translation with TLB. *paddr_noTLB* is the result of address translation without the TLB. *pagefault_TLB* indicates a page fault occurred (due to insufficient permission) during translation with TLB. *pagefault_noTLB* indicates a page fault during translation without the TLB.
- $\mathcal{V} = \{\mathit{mem}, \mathit{TLB}, \mathit{legal}\}$, where *mem* is a 32-bit addressable memory containing both the page directory and page tables. *TLB* is an array (2^{10} entries in Bochs) of structs, where each struct is 160 bits wide and has 5 32-bit fields: *vpn*, *ppn*, access bits (*ab*), etc. *legal* is a Boolean variable denoting whether the system reached the current state via a legal sequence of transitions.
- $\mathit{Init} = (\mathrm{mem}_0, \mathit{TLB}_0, \mathbf{true})$, where $\mathit{TLB}_0[i].\mathit{vpn} :=$ 0xffffffff for all i and mem$_0$ is an uninterpreted function from 32 bit addresses to arbitrary 32 bit values. The Bochs TLB is initialized with its *vpn* field set to 0xffffffff in all entries, thus making it empty. *legal* is initialized to true.
- $\mathcal{A}$: $\mathcal{V}$ evolves via operations *write_pde*, *write_pte*, *invlpg*, *invlpg_all*, *setcr3*, and *translate*, and the environment non-deterministically chooses one of these operations at each step. Table I describes each of these commands.

Command	Modifies	Guard
write_pte	*mem*	**true**
write_pde	*mem*	**true**
translate	*TLB*	$\neg\mathit{present} \vee \neg\mathit{permission}$
$\mathit{set\_cr}3$	*TLB*	**true**
invlpg	*TLB*	$\mathit{TLB}[\mathit{vaddr}_{\mathit{table}}].\mathit{vpn}_{31:12} = (\mathit{vaddr}_{\mathit{dir}} \circ \mathit{vaddr}_{\mathit{table}})$
invlpg_all	*TLB*	$\mathit{TLB}[\mathit{vaddr}_{\mathit{table}}].\mathit{vpn}_{31:22} = \mathit{vaddr}_{\mathit{dir}}$

TABLE I: The allowable operations in our model of the Bochs TLB.

Each command is implemented in distinct functions within Bochs (src/cpu/paging.cc). Since Bochs executes on a single thread, we can safely model each function as an atomic operation, i.e., a single step in the state transition system. The commands *write_pde* and *write_pte* are used to update the page directory and page tables respectively, typically to modify access permissions or page mapping. *translate* performs address translation and assigns the result to variables in $\mathcal{O}$. Furthermore, if a page walk was deemed necessary, then *translate* updates a TLB entry with the results of that page walk. If global pages are enabled, then a *setcr3* (which switches to a new page table, typically during a context switch) flushes all non-global entries in the TLB. Otherwise if global pages are disabled, all TLB entries are flushed on a *setcr3*. The x86 instruction *invlpg* flushes a specific TLB entry containing the translation for *vaddr*; *invlpg* is needed to invalidate the TLB entry following a write to the page table. *invlpg_all* atomically flushes all TLB entries that have $\mathit{vaddr}_{\mathit{table}}$ in their *vpn* (bits 31 to 22); *invlpg_all* is needed to invalidate a set of TLB entries following a write to the page directory.

We check equivalence of both the physical address and whether a page fault occurred. Since the x86 manual only guarantees cache coherency when the TLB is flushed properly, we only require equivalence on traces where each *write_pde*

[2] http://uclid.eecs.berkeley.edu/s2w/

is followed by a *invlpg_all* and each *write_pte* is followed by *invlpg*. This constraint is enforced by *legal*, which is true only if the sequence of operations abides by these constraints. Any state that satisfies *legal* is guaranteed to be reachable from the initial state via a legal sequence of state transitions.

The property that we check is:

$$\Phi_9 \doteq \forall v, p, r.\ (vaddr = v \wedge pl = p \wedge rwx = r) \rightarrow \\ legal \rightarrow ((pagefault\_TLB \Leftrightarrow pagefault\_noTLB) \wedge \\ (\neg pagefault\_noTLB \rightarrow (paddr\_noTLB = paddr\_TLB))) \quad (9)$$

1) Induction: The one-step induction check consists of proving (10) and (11) using the UCLID verifier.

$$Init(\mathcal{V}_{Bochs}) \rightarrow \Phi_9(\mathcal{V}_{Bochs}) \qquad (10)$$

$$\Phi_9(\mathcal{V}_{Bochs}) \wedge \mathcal{R}(\mathcal{V}_{Bochs}, \mathcal{V}'_{Bochs}) \rightarrow \Phi_9(\mathcal{V}'_{Bochs}) \qquad (11)$$

Our initial state satisfies Φ_9 because the TLB is initialized to be empty, thereby forcing both optimized and unoptimized designs to undergo the two-level page walk. However, one-step induction (check (11)) fails because the back-end SMT engine cannot solve the formula, which has a quantifier alternation. Consequently, we proceed onto the small and short world steps.

2) Small World: We syntactically derive the dependence set $\mathcal{U}_{\Phi_9}$ by traversing the expression graph of Φ_9. After introducing a fresh 32-bit symbolic constant $v = (v_{dir}, v_{table}, v_{offset})$, the dependence set is

$$\mathcal{U}_{\Phi_9} = \{legal, TLB[v_{table}], mem[cr3_{31:12} \circ v_{dir}], \\ mem[mem[cr3_{31:12} \circ v_{dir}]_{31:12} \circ v_{table}]\}$$

The last three expressions represent the TLB entry, page directory entry, and page table entry pointed to by v, respectively. (Here $cr3_{31:12}$ refers to the upper 20 bits of the cr3 control register and is modeled as a symbolic constant.) Our abstract model $\hat{\mathcal{S}}_{Bochs}$ precisely tracks only the variables in $\mathcal{U}_{\Phi_9}$; other state elements get updated with arbitrary values at each step.

3) Short World: We use the sub-sequence heuristic to find an upper bound on the diameter of $\hat{\mathcal{S}}_{Bochs}$. We find a bound of 9 steps. Finally, we perform bounded model checking for 9 steps, proving that all reachable states of $\hat{\mathcal{S}}_{Bochs} \| \mathcal{E}_{Bochs}$ satisfy Φ_9. The sub-sequence check and BMC took about 45 minutes and 25 minutes respectively.

B. Content Addressable Memory

While the TLB functions as a direct-mapped cache (each concrete logical address is associated with a single TLB entry), S^{2}W also applies to systems with set-associative caches and Content Addressable Memories (CAMs). A CAM stores associations between keys and data. The key is typically stored as part of the data, and is used for comparison during lookups.

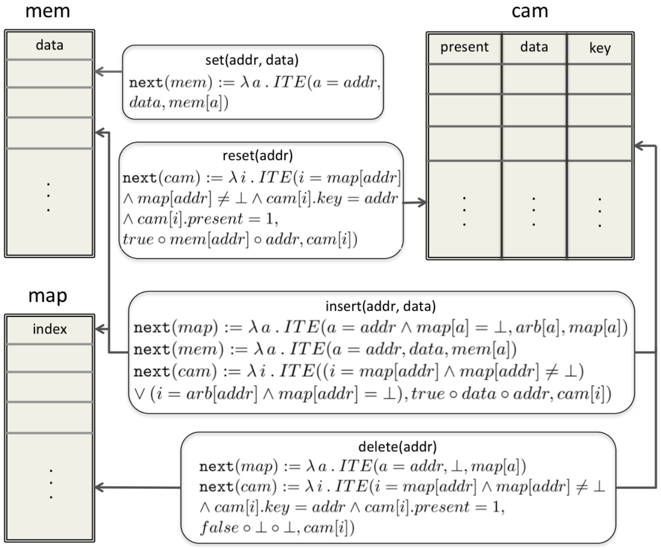
mem
data
cam
present
data
key
set(addr, data)
$\texttt{next}(mem) := \lambda a\,.\,ITE(a = addr, data, mem[a])$
reset(addr)
$\texttt{next}(cam) := \lambda i\,.\,ITE(i = map[addr] \wedge map[addr] \neq \perp \wedge cam[i].key = addr \wedge cam[i].present = 1, true \circ mem[addr] \circ addr, cam[i])$
map
index
insert(addr, data)
$\texttt{next}(map) := \lambda a\,.\,ITE(a = addr \wedge map[a] = \perp, arb[a], map[a])$
$\texttt{next}(mem) := \lambda a\,.\,ITE(a = addr, data, mem[a])$
$\texttt{next}(cam) := \lambda i\,.\,ITE((i = map[addr] \wedge map[addr] \neq \perp) \vee (i = arb[addr] \wedge map[addr] = \perp), true \circ data \circ addr, cam[i])$
delete(addr)
$\texttt{next}(map) := \lambda a\,.\,ITE(a = addr, \perp, map[a])$
$\texttt{next}(cam) := \lambda i\,.\,ITE(i = map[addr] \wedge map[addr] \neq \perp \wedge cam[i].key = addr \wedge cam[i].present = 1, false \circ \perp \circ \perp, cam[i])$

Fig. 4: Our model of a slow memory and its CAM-based cache.

Figure 4 shows a system containing slow memory and a CAM-based cache. We would like to prove that a lookup in slow memory yields the same data as lookup in the CAM-based cache (if the data is present in the CAM). We model the CAM's state using a variable *cam* that maps a CAM index to its contents, a 65-bit vector containing fields *present*, *key*, and *data*. The $cam[i].present$ bit indicates whether the key $cam[i].key$ and data $cam[i].data$ are valid entries. $cam[i].key$ and $cam[i].data$ contain the 32-bit key and 32-bit data stored at CAM index i (if $cam[i].present$ is true). Memory is modeled as an variable *mem* mapping a 32-bit address to a 32-bit vector. That is, $mem[a]$ refers to the 32-bit data at address a.

We also maintain a state variable *map* that maps an address a to a 32-bit CAM index. *map* is updated when data is added or deleted from the CAM. A $read(addr)$ command checks the contents of the CAM at index $map[addr]$. If $cam[map[addr]].key = addr$ and $cam[map[addr]].present$ is true, then $\mathcal{S}_{CAM}$ assigns $cam[map[addr]].data$ to output variable out_cam_{data} and true to output variable $out\_cam_{present}$. Otherwise, $\mathcal{S}_{CAM}$ assigns false to $out\_cam_{present}$. *read* also assigns $mem[addr]$ to output variable out_mem_{data}. The $insert(addr, data)$ command checks *map* for an existing mapping of address $addr$. If $map[addr] \neq \perp$, then $\mathcal{S}_{CAM}$ updates the CAM at index $map[addr]$ with data *data* and key *addr*.[1] Otherwise, we arbitrarily choose a new location $arb[addr]$ to insert *data* and *addr*, and update $map[addr]$ with this new location. The $set(addr, data)$ command updates $mem[addr]$ with *data*, possibly making the CAM contents stale. The $reset(addr)$ command resynchronizes *cam* with *mem* at address *addr* (if the CAM contains a valid entry for address *addr*). These commands are implemented using atomic operations, and they update *map*, *mem* and *cam* in parallel.

The CAM + memory system is modeled as a tuple $\mathcal{S}_{CAM}$ = ($\mathcal{I}$, $\mathcal{O}$, $\mathcal{V}$, *Init*, $\mathcal{A}$) where

- $\mathcal{I} = \{addr, data, command\}$
- $\mathcal{O} = \{out\_cam_{data}, out\_cam_{present}, out\_mem_{data}\}$
- $\mathcal{V} = \{cam, mem, map, legal\}$: These are all modeled as bit-vector functions. *cam* returns a 65-bit vector: 1-bit *present*, 32-bit *data*, 32-bit *addr*. Both *mem* and *map* return a 32-bit vector. *legal* is a Boolean variable denoting whether the system legally reached the current state.
- $Init = (c_0, m_0, map_0, \mathbf{true})$, where $map_0[a] = \perp$ for all a, the present field of each CAM entry is initialized to false, memory is initialized to arbitrary values, and *legal* is initialized to true.

[1] Note that $\perp$ in our model equals 0x00000000; it is an acceptable design choice to not cache that address.

- $\mathcal{A}$: The state evolves via commands *insert*, *delete*, *set*, *reset* and *read*. The environment non-deterministically chooses one of these commands at each step. Figure 4 defines the state transition relation for each command.

The safety property Φ_{12} checks that the CAM and memory have the same data for all keys present in CAM. Note that the CAM only guarantees cache coherency if it is resynchronized with memory after each *set*. The state variable *legal* enforces this constraint: it is true if every *set* is followed by a *reset*.

$$\begin{aligned}\Phi_{12} \doteq\ & \forall a.\ (\mathit{addr} = a) \rightarrow \mathit{legal} \rightarrow \\ & (\mathit{out\_cam}_{\mathit{present}} \rightarrow (\mathit{out\_cam}_{\mathit{data}} = \mathit{out\_mem}_{\mathit{data}})) \end{aligned} \tag{12}$$

Since Φ_{12} is expressed over output variables, we check if a state s satisfies Φ_{12} by performing a *read* operation on state s with a fresh symbolic constant for a.

1) Induction: We first try one-step induction on this system to prove Φ_{12}. The initial state satisfies Φ_{12} because both the CAM and *map* are empty. However, the inductive check fails because of the quantifier alternation, similar to the TLB case study. Hence, we continue onto the small world step of our approach.

2) Small World: We syntactically derive the dependence set by traversing the expression graph of Φ_{12}. We introduce a fresh 32-bit symbolic constant a for the address that we precisely track in *map* and *mem*. We precisely track *legal*, $\mathit{map}[a]$, $\mathit{mem}[a]$, and $\mathit{cam}[\mathit{map}[a]]$.

$$\mathcal{U}_{\Phi_{12}} = \{\mathit{legal}, \mathit{cam}[\mathit{map}[a]], \mathit{mem}[a], \mathit{map}[a]\} \tag{13}$$

Our abstract model $\hat{\mathcal{S}}_{CAM}$ precisely tracks updates to only these variables.

3) Short World: Using the sub-sequence heuristic, we find an upper bound on the reachability diameter of 5 steps. Finally, we perform bounded model checking for 5 steps, proving that all reachable states of $\hat{\mathcal{S}}_{CAM} \| \mathcal{E}_{CAM}$ satisfy Φ_{12}. The sub-sequence check and BMC takes about 15 seconds and 5 seconds respectively.

C. Shadow Paging

For our third case study, we model a shadow page table system. A hypervisor may use shadow page tables to assure address space separation between the guest and host. The guest page tables can be updated arbitrarily by the guest operating system, while the shadow page tables are updated only by the hypervisor. The hardware uses the shadow page tables for address translation, so it is the hypervisor's responsibility to make sure the shadow page tables stay synchronized with the guest tables, while at the same time ensuring no translation will ever allows the guest to access memory outside its allocated sandbox. We model the synchronization process and verify that the physical address returned by translation never exceeds some constant limit, LIMIT.

Our shadow paging model (Figure 5) is as follows. There are two page table structures: guest and host. Each is a two-level structure: a page directory table (PDT) and a page table (PT). We refer to the guest and shadow page tables as gPDT, gPT and sPDT, sPT, respectively. Entries in the PDT have three fields: present (*p*), page-size-extension (*pse*), and address (*addr*). Entries in each nested PT have two fields: present (*p*) and address (*addr*).

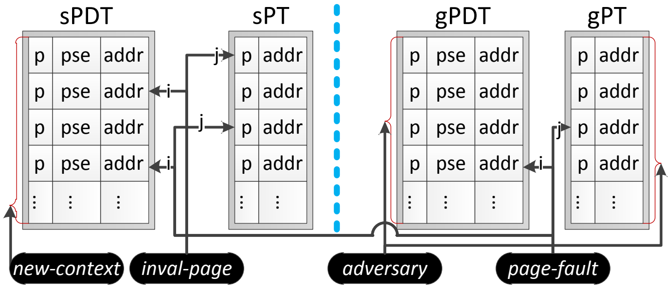

Fig. 5: An illustration of the shadow page table model.

Let $\mathcal{S}_{SP}$ = ($\mathcal{I}$, $\mathcal{V}$, *Init*, $\mathcal{A}$) be the shadow paging model with

- $\mathcal{I} = \{i, j, \mathit{command}\}$: i and j index into the PDT and PT, respectively; *command* is one of *page-fault*, *inval-page*, *new-context*, or *adversary*.
- $\mathcal{V} = \{\text{gPDT}, \text{gPT}, \text{sPDT}, \text{sPT}, \text{LIMIT}\}$. gPDT, gPT, sPDT, and sPT are modeled as functions that map indices to bit-vectors. gPDT and sPDT return 34-bit vectors: (1-bit *p*, 1-bit *pse*, 32-bit *addr*). gPT and sPT return 33-bit vectors: (1-bit *p*, 32-bit *addr*). LIMIT is a constant 32-bit vector.
- $\mathit{Init} = (\text{sPDT}_0, \text{sPT}_0, \text{gPDT}_0, \text{gPT}_0)$, where sPDT and sPT are both initialized with the *p* bit cleared in all entries. gPDT and gPT are initialized to arbitrary values.
- $\mathcal{A}$: The four commands update state in the following way: *page-fault* synchronizes the shadow tables with the guest tables. *inval-page* conditionally invalidates (zeros out) entries in sPDT and sPT. *new-context* unconditionally invalidates entries in sPDT. *adversary* writes to gPDT and gPT. The assignments to gPDT, gPT, sPDT, and sPT are summarized in Table II.

Command	Modifies	Guard
page-fault(i, j)	sPDT$[i]$	gPDT$[i]$.*pse* $\wedge$ gPDT$[i]$.*p*$\wedge$ (gPDT$[i]$.*addr* $<$ LIMIT)
	sPDT$[i]$	gPDT$[i]$.*pse* $\wedge$ $\neg$(gPDT$[i]$.*p*$\wedge$ (gPDT$[i]$.*addr* $<$ LIMIT))
	sPDT$[i]$, sPT$[j]$	$\neg$gPDT$[i]$.*pse* $\wedge$ gPDT$[i]$.*p*$\wedge$ (gPDT$[i]$.*addr* $<$ LIMIT)$\wedge$ gPT$[j]$.*p* $\wedge$ (gPT$[j]$.*addr* $<$ LIMIT)
	sPDT$[i]$	$\neg$gPDT$[i]$.*pse* $\wedge$ gPDT$[i]$.*p*$\wedge$ (gPDT$[i]$.*addr* $<$ LIMIT)$\wedge$ $\neg$(gPT$[j]$.*p* $\wedge$ (gPT$[j]$.*addr* $<$ LIMIT))$\wedge$ $\neg$(sPDT$[i]$.*p* $\wedge$ $\neg$sPDT$[i]$.*pse*)
	sPDT$[i]$	$\neg$gPDT$[i]$.*pse* $\wedge$ $\neg$(gPDT$[i]$.*p*$\wedge$ (gPDT$[i]$.*addr* $<$ LIMIT))
	sPT$[j]$	$\neg$gPDT$[i]$.*pse* $\wedge$ gPDT$[i]$.*p*$\wedge$ (gPDT$[i]$.*addr* $<$ LIMIT)$\wedge$ $\neg$(gPT$[j]$.*p* $\wedge$ (gPT$[j]$.*addr* $<$ LIMIT))$\wedge$ (sPDT$[i]$.*p* $\wedge$ $\neg$sPDT$[i]$.*pse*
inval-page(i, j)	sPDT$[i]$	(sPDT$[i]$.*p* $\wedge$ $\neg$gPDT$[i]$.*p*)$\vee$ (sPDT$[i]$.*p* $\wedge$ gPDT$[i]$.*p*$\wedge$ (sPDT$[i]$.*pse* $\vee$ gPDT$[i]$.*pse*))
	sPT$[j]$	sPDT$[i]$.*p* $\wedge$ gPDT$[i]$.*p*$\wedge$ $\neg$gPDT$[i]$.*pse* $\wedge$ $\neg$sPDT(i).*pse*
new-context	gPDT	`true`
adversary	gPDT	`true`
	gPT	`true`

TABLE II: Next-state assignments for the shadow paging model.

Our model is based on the ShadowVisor model [11], but has been extended to introduce pointers. The safety properties we verify are similar to those of ShadowVisor. We verify that a translation using the shadow page tables will never return an address above a fixed limit.

$$\begin{aligned}\Phi_{14} = \forall i.\ & (\text{sPDT}[i].p \wedge \text{sPDT}[i].\mathit{pse}) \rightarrow \\ & \text{sPDT}[i].\mathit{addr} < \text{LIMIT}\end{aligned} \tag{14}$$

$$\Phi_{15} = \forall i,j.\ (\text{sPDT}[i].p \wedge \neg\text{sPDT}[i].\mathit{pse} \wedge \text{sPT}[j].p) \rightarrow \text{sPT}[j].\mathit{addr} < \text{LIMIT} \quad (15)$$

1) Induction: The property $\mathbf{G}\ \Phi_{14}$ is proven by induction. However, one-step induction fails to prove $\mathbf{G}\ \Phi_{15}$. If sPT$[j]$ is marked as present, but has an address greater than LIMIT, Φ_{15} is still true as long as sPDT$[i]$ is marked not present. From that state, it is possible to update sPDT$[i]$ to present without updating sPT$[j]$, so that in the next state the sPDT$[i]$ and sPT$[j]$ entry are both marked present and the data in the sPT$[j]$ entry is greater than LIMIT, violating Φ_{15}. Therefore, we move on to the small and short world steps for $\mathbf{G}\ \Phi_{15}$.

2) Small World: The properties are concerned with a single entry i in sPDT and a single entry j in sPT. Therefore we use a fresh symbolic constant to choose an arbitrary entry from each. In particular, our conditional dependency set is $\mathcal{U}_{\phi_{15}} = \{\text{sPDT}[a_i], \text{sPT}[a_j]\}$. In our abstract symbolic transition system, $\hat{\mathcal{S}}_{SP}$, we track precisely only the state in $\mathcal{U}_{\phi_{15}}$. a_i, a_j stay constant.

3) Short World: The sub-sequence short-world heuristic does not work for page tables, because page-table entry updates can depend on previous writes to the entry or to other entries; it is not always possible to drop one step of a trace to achieve an equivalent final state. Instead, we use gadgets to find and prove the short world. We manually construct a universal gadget set to prove the length of the diameter of $\hat{\mathcal{S}}_{SP}$.

To build the gadgets we case split on the possible end-state valuations for the variables in $\mathcal{U}_{\phi_{15}}$, and for each, determine how to get there from a valid starting state. The model has only four commands and it was usually obvious which commands were needed to get to a particular state. The gadgets must also specify the parameters (i, j) to the command, and, in the case of the *adversary* command, the value that gets written to the guest tables. Figuring out the correct parameters to use for each command was more difficult. In this case, the parameters were always either $i := a_i$ or $i := a'_i$ (where $a'_i \neq a_i$ is arbitrarily chosen), and similarly for j. The *adversary* data that gets written to the guest tables was a combination of the *addr* field of the (symbolic) end-state valuation we were trying to achieve and a particular value for the *p* and *pse* bits, which we chose according to the particular gadget we were building.

We needed a total of thirteen gadgets, each four commands or less, to prove the short world has length four. We then ran BMC on $\hat{\mathcal{S}}_{SP}$ for four steps and verified the property held at each step. The verification of the short world took less than a minute; BMC took approximately five seconds.

D. Other Hypervisor Models

We applied our abstraction technique to three additional hypervisor models. All were verified using one-step induction, each within 5 seconds.

SecVisor: SecVisor [12], [25] is a small hypervisor that supports a single guest OS. It virtualizes the memory management unit by implementing shadow page tables and synchronizing them with the guest page tables. The model assumes an adversary can write arbitrary values to the guest page tables. SecVisor aims to execute only approved code in kernel mode; therefore, the page-table synchronization must prevent adversary-provided code from having execute permissions while in kernel mode. We verified the security property using one-step induction on a model given by Franklin et al. [12].

sHype: sHype [12] is an access control system used by the Xen [2] hypervisor. Based on the Chinese Wall policy, it establishes “conflict of interest” classes and guarantees each virtual machine will never access two pieces of data from the same conflict of interest class. We verified the security property using a model presented by Franklin et al. [12].

ShadowVisor: ShadowVisor [11] served as the starting point for our shadow page table model. It models the page tables of a simple hypervisor that assures address space separation between guest and host by maintaining separate guest and shadow page tables. Like our shadow page table model, ShadowVisor guarantees that if an address is marked as present, it will never exceed a certain fixed limit. We model ShadowVisor in our modeling language and use one-step induction to verify the property.

VI. Related Work

Verification of infinite-state or parametrized systems has been well-studied. Here we present the most closely-related work.

Franklin et al. [11], [12] present a small-model approach to verifying systems with parametrized data structures (arrays). In essence, they present a formal language such that if the system can be modeled in their language, then a small-model theorem applies, stating that the unbounded arrays can be reduced to arrays with one designated element alone. Finite-state model checking can then be employed on the resulting system. While this approach is very elegant, there are some important differences with the approach in this paper. First, our modeling language is more expressive, allowing us to model the Bochs TLB, CAM, and shadow paging examples which cannot be modeled in their language. Second, our approach is different: we compute an abstraction based on localization within the large data structures, and we use bounded model checking.

The use of inductive invariant checking is common in this problem domain. The method of *invisible invariants* [6], [22] is an example of an inductive verification technique applied to systems of N identical finite-state processes. The core idea in this method is to generalize from the reachable states of a small number of processes into a quantified inductive assertion of the form $\forall i.\phi(i)$, where the index i ranges over process IDs. Namjoshi [24] also discusses the so-called *cutoff method*, which is a small-model approach for such systems of parametrized processes, drawing connections between inductive methods, small-model approaches and compositional reasoning. In general, these approaches are not easily applied to our examples since they are not naturally decomposed into a system of N identical finite-state processes. Instead, in our problem domain, the number of interacting processes is usually finite and small, but the shared data structures are large and complicated.

Abstraction-based approaches have also been presented for infinite-state or parametrized systems similar to those studied in this paper. Lahiri and Bryant [18] presented an approach for verifying universally-quantified invariants on parametrized systems using predicate abstraction. While predicate abstraction can be quite effective for verifying control-related properties, especially when one can guess suitable predicates, it is not suitable for verifying equivalence of two code versions (such as in the Bochs TLB case study), which is a highly data-dependent property. McMillan [21] presents a semi-automatic approach to compositional reasoning using an abstraction similar to ours. Given a system with large arrays, he uses a form of localization abstraction (guided by case splitting performed by the user) to only model a few entries in the arrays precisely, allowing all other entries to be updated by an arbitrary value $\perp$. This abstraction is used to compute a finite-state abstract model on which reachability analysis is performed. In contrast, our method computes an abstraction based on index terms derived from the property, and uses a BMC-based approach to verify the system. Bjesse [8] describes an automatic approach for verifying sequential circuits with large memories, if the memories are "remodellable" (a notion made formal in [8]). The work takes a "small-world" approach, transforming an initial netlist into another with memories with precise updates only to a small number of entries in memories, and uses counterexample-guided abstraction-refinement. Our work does not require the "remodellable" restriction. German [14] presents a novel approach for constructing sound and complete abstractions for similar systems. The approach is based on performing static analysis on the system model, and (in contrast to [8]) can handle unbounded delays between the time the array is read and when the read value propagates to the output. While the approach is completely automatic, it cannot handle certain data structures such as CAMs, in which a read, in principle, requires scanning the entire array. Our approach, while not always automatic, does handle structures such as CAMs (see Sec. V-B).

Efficient memory modeling is a technique used for bounded verification problems, either symbolic simulation (e.g. [27]) or bounded model checking (e.g. [13]). In contrast, our approach focuses on unbounded verification based on abstraction and a sound application of BMC based on heuristics to find the reachability diameter.

There has also been prior work on verifying emulators and hypervisors. Alkassar et al. [5] presented the verification of the TLB logic in the Hyper-V hypervisor. They verify invariant properties of the TLB using the VCC verifier. Their approach, like ours, is not fully automatic. While our approach uses abstraction-based model checking, assuming a particular model of atomicity of operations, theirs is a classic deductive verifier for C code (using VC generation and theorem proving) that operates at a somewhat lower level of abstraction.

VII. Conclusion

We have presented S^{2}W, a new approach to verifying systems with large or unbounded data structures that combines induction and abstraction-based model checking. Experimental results have been presented on several examples of emulators and hypervisors. In ongoing work, we are investigating how to make the technique more automated, to automatically generate abstract, term-level models from C/C++ code, and to validate these models.

Acknowledgments. This research was supported in part by SRC contract 2045.001, by Intel through the ISTC for Secure Computing, by the AFOSR under MURI award FA9550-09-1-0539, and by a generous gift from Intel. We thank Randal E. Bryant, Orna Kupferman, and Anupam Datta for their valuable feedback.

References

[1] Plingeling SAT Solver. http://fmv.jku.at/lingeling.

[2] The Xen Hypervisor. http://www.xen.org/.

[3] UCLID Verification System. Available at http://uclid.eecs.berkeley.edu.

[4] VMware Security Advisory vmsa-2009-0015. http://www.vmware.com/security/advisories/VMSA-2009-0015.html, 2009.

[5] E. Alkassar, E. Cohen, M. A. Hillebrand, M. Kovalev, and W. J. Paul. Verifying shadow page table algorithms. In *FMCAD*, 2010.

[6] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. D. Zuck. Parameterized Verification with Automatically Computed Inductive Assertions. In *CAV*, 2001.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *SOSP*, 2003.

[8] P. Bjesse. Word-Level Sequential Memory Abstraction for Model Checking. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2008.

[9] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In *Computer-Aided Verification (CAV'02)*, 2002.

[10] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Convergence Testing in Term-Level Bounded Model Checking. In *Correct Hardware Design and Verification Methods (CHARME)*, 2003.

[11] J. Franklin, S. Chaki, A. Datta, J. M. McCune, and A. Vasudevan. Parametric Verification of Address Space Separation. In *POST*, 2012.

[12] J. Franklin, S. Chaki, A. Datta, and A. Seshadri. Scalable Parametric Verification of Secure Systems: How to Verify Reference Monitors without Worrying about Data Structure Size. In *IEEE Security & Privacy*, 2010.

[13] M. K. Ganai, A. Gupta, and P. Ashar. Efficient Modeling of Embedded Memories in Bounded Model Checking. In *Proc. Computer-Aided Verification (CAV)*, 2004.

[14] S. German. A theory of abstraction for arrays. In *FMCAD*, 2011.

[15] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical Taint-Based Protection using Demand Emulation. In *EuroSys*, 2006.

[16] A. J. Isles, R. Hojati, and R. K. Brayton. Computing reachable control states of systems modeled with uninterpreted functions and infinite memory. In *Computer-Aided Verification (CAV '98)*, 1998.

[17] R. Kurshan. Automata-Theoretic Verification of Coordinating Processes. In *11th International Conference on Analysis and Optimization of Systems – Discrete Event Systems*, volume 199. Springer Berlin / Heidelberg, 1994.

[18] S. K. Lahiri and R. E. Bryant. Predicate abstraction with indexed predicates. *ACM Trans. Comput. Log.*, 9(1), 2007.

[19] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis. Path-Exploration Lifting: Hi-Fi Tests for Lo-Fi Emulators. In *ASPLOS*, 2012.

[20] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing CPU Emulators. In *ISSTA*, 2009.

[21] K. L. McMillan. Verification of Infinite State Systems by Compositional Model Checking. In *Correct Hardware Design and Verification Methods (CHARME)*, 1999.

[22] K. L. McMillan and L. D. Zuck. Invisible Invariants and Abstract Interpretation. In *SAS*, 2011.

[23] D. Mihocka and S. Shwartsman. Virtualization Without Direct Execution or Jitting : Designing a Portable Virtual Machine Infrastructure. *AMASBT*, 2008.

[24] K. S. Namjoshi. Symmetry and Completeness in the Analysis of Parameterized Systems. In *VMCAI*, 2007.

[25] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *SOSP*, 2007.

[26] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-Into-Libc Without Function Calls (on the x86). In *ACM CCS*, 2007.

[27] M. N. Velev, R. E. Bryant, and A. Jain. Efficient Modeling of Memory Arrays in Symbolic Simulation. In *Proc. Computer-Aided Verification (CAV)*, 1997.

[28] Q. Zhang, J. McCullough, J. Ma, N. Schear, M. Vrable, A. Vahdat, A. C. Snoeren, G. M. Voelker, and S. Savage. Neon: System Support for Derived Data Management. In *VEE*, 2010.