Symbolic Software Model Validation

Cynthia Sturton,[1] Rohit Sinha,[2] Thurston H.Y. Dang,[2] Sakshi Jain,[2] Michael McCoyd,[2]
Wei Yang Tan,[2] Petros Maniatis,[3] Sanjit A. Seshia,[2] and David Wagner [2]

[1]University of North Carolina at Chapel Hill
[2]University of California, Berkeley
[3]Intel Labs

Abstract—**Modeling is the crucial first step in formal verification. Some models are constructed by humans from source code, while others are extracted automatically by tools. Regardless of how a model is constructed, verification is only as good as the model; therefore, it is essential to validate the model against the implementation it represents. In this paper we present two complementary approaches to software model validation. The first, data-centric model validation, checks that, for data structures relevant to the property being verified, all operations that update these data structures are captured in the model. The second, operation-centric model validation, checks that each operation being modeled is correctly simulated by the model. Both techniques are based on a combination of symbolic execution and satisfiability modulo theories (SMT) solving. We demonstrate the application of our methods on several case studies, including the address translation logic in the Bochs x86 emulator, the Berkeley Packet Filter, a TCAS benchmark suite, the FTP server from GNU Inetutils, and a component of the XMHF hypervisor.**

I. Introduction

In formal verification, one typically first creates a model of the system to verify, and then uses techniques such as model checking and theorem proving to prove properties about the model. A model may be constructed manually from source code or extracted automatically by a tool. In either case, successful verification means the property has been proven true of the model, but not necessarily of the original system. This weakness is clear for manually-constructed models, but also holds for tools operating directly on source code. Software model checkers construct and internally maintain a model of the code, on which the analysis is performed. Moreover, to scale to large code bases, tools often require human guidance (e.g., code annotations or modifications). Thus, again, one implicitly trusts that the model is correct: if model verification succeeds, one assumes the verification also holds for the original system.

In this work we present a framework for validating that assumption. We formalize the process of validating a model against the original source code implementation of the system to prove that a property proven true of the model is also true of the system. We concentrate solely on validating models that will be used for the verification of safety properties, and on systems implemented in imperative languages (e.g., C/C++). A special focus is on systems software that exports several "logical operations" that client programs can use (or misuse), such as an FTP server and its commands or a virtual machine monitor and its services to operating systems hosted on it. Such systems are best understood as being logically concurrent, even if they are single-threaded. Each logical operation operates on an associated collection of data. Therefore, the corresponding models are usually specified as transition systems with logical operations that can be interleaved in a specified manner to operate on their associated data structures based on client inputs. Data-structure complexity often necessitates the use of abstraction using first-order logic with suitable background theories.

To be sound, model validation must show that all states reachable in the original system are also reachable in the model. However, often a system has many equivalent states with respect to the property to be verified. Modelers use this fact to model only relevant portions of the system. For example, the Bochs CPU Emulator [21] is a large system comprising over 400 C/C++ files, several files in other languages, and over 200 KLoC.[1] A model of the entire system is daunting, and possibly unnecessary. A property about address translation, for example, might require a model of only a few of the system modules. Modules for CPU-instruction fetch and decode might safely be excluded from the model. The first phase of model validation, therefore, is to show that the modelers selected modules correctly: no portion of the system was excluded from modeling while being relevant to the property under consideration. This requires first identifying the set of relevant variables and, then, showing that those variables are never written by program code that has been excluded from modeling.

The second phase of model validation is to show that the model is an overapproximation of the code with respect to the property in question. We check that the model simulates the relevant code by checking that each logical operation in the code is overapproximated by a corresponding operation

Work done while Sturton was at the University of California, Berkeley.

[1]Calculated for Bochs 2.6.2 using David A. Wheeler's SLOCCount.

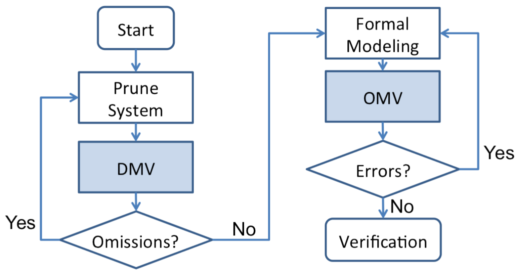

Figure 1: Model checking-based verification with model validation. Model validation consists of the DMV and OMV steps (shaded).

```
if (curr_privilege_level == 3)
   page_fault = 1;
else page_fault = 0;
...
update_access_dirty(...);
```

(a) Code.

```
page_fault := (cpl[0] & cpl[1]);
```

(b) Model. `curr_privilege_level` is abbreviated to `cpl`.

Figure 2: A code snippet and the corresponding model.

in the model. This is formulated as checking validity of a logical formula and discharged using an SMT solver [1].

We propose the following work flow (see Figure 1):

1. *System Pruning*: Before modeling, modelers prune irrelevant code. Given a system $\mathcal{S}$ and a property Φ, relevant code, $\mathcal{F}_{\mathcal{P}}$, is identified. This is typically a manual step that requires some domain expertise.
2. *Data-Centric Model Validation (DMV)*: The pruning done in the previous step is validated. If DMV finds that $\mathcal{F}_{\mathcal{P}}$ is missing relevant code, the process returns to Step 1.
3. *Model Construction*: Given $\mathcal{F}_{\mathcal{P}}$, modelers build a formal model $\mathcal{M}$ of the pruned system.
4. *Operation-Centric Model Validation (OMV)*: $\mathcal{M}$ is validated as a sound abstraction of $\mathcal{F}_{\mathcal{P}}$ with respect to Φ. If OMV finds discrepancies, the process returns to Step 3.
5. *Verification*: Model checking-based verification is done.

In this paper we focus on Steps 2 and 4 of the work flow; we assume pruning and modeling are manual, and that any standard model-checking technique can be used in Step 5. In our theoretical framework, DMV and OMV are sound, but our current implementation is not. We use symbolic execution [14] and SMT solving [1] to implement DMV and OMV. The former is used to explore all feasible code paths, and the latter to check the validity of queries generated during DMV and OMV. Due to limitations of symbolic execution tools, DMV, in some cases, fails to catch portions of $\mathcal{S}$ that were omitted from $\mathcal{F}_{\mathcal{P}}$. We rely on modeler expertise to identify the relevant code to model and use DMV for debugging. In that sense, our DMV implementation increases the modeler's confidence.

Our implementation currently uses KLEE [4] for symbolic execution, the UCLID [2] language for building our models, and the UCLID decision procedure for proving the SMT queries that check validation. The methodology can be made to work with other tools as well.

This paper makes the following contributions:

- A theoretical framework for software model validation.
- An implementation of the framework using symbolic execution and SMT solving.
- Evaluation of our procedure on software benchmarks. Our case studies include the Bochs address translation logic, a component of the Berkeley Packet Filter, a TCAS benchmark suite, the FTP server from GNU Inetutils, and a component of the XMHF hypervisor.

II. Running Example

We present here an example snippet of code and the corresponding model (Figure 2), which we use later to illustrate our algorithm. The code snippet comes from the Bochs CPU emulator. It tests, for memory pages marked as supervisor-only, whether the CPU has a current privilege level (CPL) of 3. If it does, a page fault will be raised (page-fault flag set to 1), indicating a failed protection check. Otherwise, the page-fault flag is set to 0. After a successful check for a page fault (not shown), the access and dirty bits of the page table entry are updated. In this example `curr_privilege_level` and `page_fault` are both integer variables.

The modeler asserts that `update_access_dirty` is irrelevant to the CPL, so she prunes it.

In the model, if bits 0 and 1 of the current privilege variable are both set, the page-fault flag is set to 1, otherwise, it is set to 0. `cpl` is modeled as a 32-bit bitvector, and `page_fault` as a 1-bit bitvector.

III. Theoretical Formulation and Approach

A. Notation and Background

Program: We use "system" or $\mathcal{S}$ to refer to the original software system to verify; Φ is the property to verify.

$\mathcal{S}$ is represented as a tuple $\mathcal{S} = (\mathcal{I}_{\mathcal{S}}, \mathcal{V}_{\mathcal{S}}, Init_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}})$, where

- $\mathcal{I}_{\mathcal{S}}$ is a finite set of input variables;
- $\mathcal{V}_{\mathcal{S}}$ is a finite set of state variables;
- $Init_{\mathcal{S}}$ is a predicate characterizing the set of initial valuations to variables in $\mathcal{I}_{\mathcal{S}}$ and $\mathcal{V}_{\mathcal{S}}$;

- $\mathcal{C}_{\mathcal{S}}$ is the code for the program (in an imperative programming language such as C or C++) describing how the system variables are updated.

The input variables, $\mathcal{I}_{\mathcal{S}}$, are the read-only variables of the system. The state variables, $\mathcal{V}_{\mathcal{S}}$, are all other variables of the program, including any global variables and return variables.

We make no assumptions about $\mathcal{C}_{\mathcal{S}}$ except the availability of a function σ that maps $\mathcal{C}_{\mathcal{S}}$ to a transition relation between pairs of system states, given a notion of a step. In other words, σ gives *relational semantics* to $\mathcal{S}$.

We term "annotated system" or $\mathcal{A}$ the representation of $\mathcal{S}$ that highlights particular code fragments as being "relevant" to the verification task at hand. More formally, we define $\mathcal{A}$ as a tuple: $\mathcal{A} = (\mathcal{I}_{\mathcal{A}}, \mathcal{V}_{\mathcal{A}}, \mathit{Init}_{\mathcal{A}}, \mathcal{F}_{\mathcal{A}})$ where

- $\mathcal{I}_{\mathcal{A}}$ is a finite set of input variables: $\mathcal{I}_{\mathcal{A}} = \mathcal{I}_{\mathcal{S}}$;
- $\mathcal{V}_{\mathcal{A}}$ is a finite set of state variables: $\mathcal{V}_{\mathcal{A}} = \mathcal{V}_{\mathcal{S}}$;
- $\mathit{Init}_{\mathcal{A}}$ is a predicate characterizing the set of initial valuations to variables in $\mathcal{I}_{\mathcal{A}}$ and $\mathcal{V}_{\mathcal{A}}$;
- $\mathcal{F}_{\mathcal{A}} = \{f_1, f_2, \ldots, f_N, f_{\text{misc}}, f_{\text{orc}}\}$ is a finite set of code fragments in $\mathcal{C}_{\mathcal{S}}$, where $f_1, f_2, \ldots, f_N$ are code fragments that capture the relevant parts of $\mathcal{C}_{\mathcal{S}}$, f_{misc}, which is disjoint from $f_1, f_2, \ldots, f_N$, represents the code in $\mathcal{C}_{\mathcal{S}}$ deemed irrelevant, and f_{orc} is the code that orchestrates how program execution interleaves between the code fragments $f_1, f_2, \ldots, f_N, f_{\text{misc}}$.

For the rest of the paper, we assume the code fragments $f_1, f_2, \ldots, f_N, f_{\text{misc}}$ to be terminating.[2] We do not make this assumption about the orchestration code f_{orc} since in general, for reactive systems like an operating system or emulator, f_{orc} is designed to run forever. Often, the code fragment f_i $(1 \leq i \leq N)$ is a single C/C++ function in $\mathcal{C}_{\mathcal{S}}$, but in general we only require that each code fragment have clearly specified entry and exit points. We also require that each of $f_1, f_2, \ldots, f_N$ executes atomically.

The code f_{orc} dictates how the code fragments $f_1, f_2, \ldots, f_N, f_{\text{misc}}$ are composed together. For example, f_{orc} might simply be the sequential composition of two code fragments. A more complicated orchestrator f_{orc} might iterate over a "while(1)" loop, repeatedly selecting a code fragment to execute based, for example, on a scheduling policy or on a sequence of external inputs. An example of the latter is a CPU emulator that repeatedly emulates instructions within a loop, where each instruction type has an associated function that emulates it. The form of the orchestrator depends heavily on the type of system under verification.

[2]Note that we may be able to lift the assumption about f_{misc}, the code deemed irrelevant to the verification task, provided that the technique for checking this irrelevance can handle non-terminating code. For simplicity, we retain this assumption in the present paper.

The function σ applies to $f_1, f_2, \ldots, f_N, f_{\text{misc}}$ in exactly the same way as it applies to $\mathcal{C}_{\mathcal{S}}$ — it characterizes the code fragments in terms of their underlying transition relation, given the notion of a step. In this paper, since we assume that $f_1, f_2, \ldots, f_N, f_{\text{misc}}$ are terminating, the notion of a step will be taken to be a single execution of a code fragment. Thus, $\sigma(f_i)$ is the set of (pre, post) state pairs of f_i. We will often abbreviate $\sigma(f_i)$ by δ_i.

From $\mathcal{A}$, we can create a "program" $\mathcal{P}$ by dropping f_{misc} from the set $\mathcal{F}_{\mathcal{A}}$ and treating any invocation of f_{misc} from f_{orc} as a no-op (stuttering step). We define $\mathcal{P}$ as a tuple analogously to $\mathcal{A}$: $\mathcal{P} = (\mathcal{I}_{\mathcal{P}}, \mathcal{V}_{\mathcal{P}}, \mathit{Init}_{\mathcal{P}}, \mathcal{F}_{\mathcal{P}})$ where

- $\mathcal{I}_{\mathcal{P}}$ is a finite set of input variables: $\mathcal{I}_{\mathcal{P}} \subseteq \mathcal{I}_{\mathcal{A}} \cup \mathcal{V}_{\mathcal{A}}$;
- $\mathcal{V}_{\mathcal{P}}$ is a finite set of state variables: $\mathcal{V}_{\mathcal{P}} \subseteq \mathcal{V}_{\mathcal{A}}$;
- $\mathit{Init}_{\mathcal{P}}$ is a predicate characterizing the set of initial valuations to variables in $\mathcal{I}_{\mathcal{P}}$ and $\mathcal{V}_{\mathcal{P}}$;
- $\mathcal{F}_{\mathcal{P}} = \{f_1, f_2, \ldots, f_N, f_{\text{orc}}\}$ is a finite set of code fragments. The f_i and f_{orc} are defined as in $\mathcal{A}$.

The transition relation of the overall program $\mathcal{P}$ is determined from those of its code fragments by the form of composition, i.e., by f_{orc}; we will denote it as $\delta^{\mathcal{P}}$.

As an example, consider the program in Figure 2a. This program is modeled as the tuple $\mathcal{P} = (\mathcal{I}_{\mathcal{P}}, \mathcal{V}_{\mathcal{P}}, \mathit{Init}_{\mathcal{P}}, \mathcal{F}_{\mathcal{P}})$, where $\mathcal{I}_{\mathcal{P}} = \{\mathit{curr\_privilege\_level}\}$, $\mathcal{V}_{\mathcal{P}} = \{\mathit{page\_fault}\}$, $\mathit{Init}_{\mathcal{P}} = \textbf{true}$ (allowing arbitrary values to *page_fault* and *curr_privilege_level*), and $\mathcal{F}_{\mathcal{P}} = \{f_1, f_{\text{orc}}\}$ where f_1 includes all the code except the function `update_access_dirty` and f_{orc} is the sequential composition $f_1; f_{\text{misc}}$. In the corresponding $\mathcal{A}$, f_{misc} was defined to include only `update_access_dirty`.

Symbolic Execution: One approach to computing the relational semantics of a code fragment f is to enumerate its paths, using symbolic execution [14] to compute path conditions which can then be combined to form the transition relation $\sigma(f)$.

A program is symbolically executed in the following way. Initially, the variables in $\mathcal{I}_{\mathcal{P}}$ and $\mathcal{V}_{\mathcal{P}}$ are each represented as a symbolic value. Each variable starts with a fresh, unconstrained symbolic value. As the program executes, operations are computed symbolically. At the first conditional branch, execution can continue down one of two paths. The symbolic execution engine forks execution, and for each path, creates a new *path condition*, which is a predicate $P(\mathcal{I}_{\mathcal{P}}, \mathcal{V}_{\mathcal{P}})$ on the input and state variables that is true along that path. At each subsequent conditional branch and execution fork, the path condition for the current path of execution is updated with the new predicate. If π is the current path condition and a conditional branch creates the two predicates $P, \neg P$, then after forking execution, the two new path conditions will be $\pi_1 = \pi \wedge P$ and $\pi_2 = \pi \wedge \neg P$. After execution has completed, for every path explored, there is a path condition π, which is a conjunction of predicates on the input and state variables. Restricting $Init_{\mathcal{P}}$ to values that satisfy the path condition and then executing the program will always force execution down the same path. Along with each π we can also take note of the final value of the state variables of the program, $\mathcal{V_P}'$. These values may be concrete or symbolic or some combination of the two. After any execution of the program along the path described by π, the state of the program will be consistent with $\mathcal{V_P}'$.

Once the symbolic execution engine has explored all possible paths through the program fragment, we have a set $R = \{(\pi, \mathcal{V_P}')\}$ of pairs of path conditions and their corresponding output state. This set effectively describes the input–output relation of the program.

Going back to the example in Figure 2a, note that there are two possible paths through the code, with corresponding path conditions:

$$\pi_1 : \mathit{curr\_privilege\_level} = 3$$
$$\pi_2 : \mathit{curr\_privilege\_level} \neq 3.$$

The corresponding next-states for each path are:

$$\mathcal{V_P}_1' : \mathit{page\_fault} = 1$$
$$\mathcal{V_P}_2' : \mathit{page\_fault} = 0.$$

Model: We now formalize the class of models considered in this work. Our models are constructed as transition systems where state variables have one of three possible types: Boolean, bit-vector, or memory. The last type can be used to model arrays or various data structures. Fig. 3 shows the grammar for expressions in our modeling language. Broadly speaking, the expressions are in a combination of the theories of uninterpreted functions, finite-precision bit-vector arithmetic, and arrays [1].

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

Figure 3: The syntax of expressions. 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*.

A formal model is represented as a tuple $\mathcal{M} = (\mathcal{I_M}, \mathcal{V_M}, Init_{\mathcal{M}}, \mathcal{O_M})$ where

- $\mathcal{I_M}$ is a finite set of input variables;
- $\mathcal{V_M}$ is a finite set of state variables;
- $Init_{\mathcal{M}}$ is a predicate characterizing the set of initial valuations to variables in $\mathcal{I_M}$ and $\mathcal{V_M}$;
- $\mathcal{O_M} = \{op_1, op_2, \ldots, op_N, main\}$ is a finite set of "operations" that determine how the state variables evolve, where *main* is a specially designated "top-level" operation that determines how the remaining operations are composed together. $\mathcal{O_M}$ defines the transition relation $\delta^{\mathcal{M}}$ of the model, as described below.

Each operation is a finite set of assignments to variables in $\mathcal{V_M}$. Assignments define how state variables are updated in a single step of the transition system. A *next-state assignment* α updates a state variable and is a rule with one of the following forms:

$$\texttt{next}(v) := e,$$
$$\texttt{next}(v) := \{e_1, e_2, \ldots, e_n\}, \text{or}$$
$$\texttt{next}(v) := *$$

where v is a signal in $\mathcal{V_M}$, $e, e_1, e_2, \ldots, e_n$ are expressions in the grammar of Fig. 3 that are a function of $\mathcal{V_M} \cup \mathcal{I_M}$, and "$*$" is a wildcard that is translated at each transition into a fresh symbolic constant of the appropriate type. The curly braces express non-deterministic choice.

The transition relation corresponding to an operation op_i, denoted $\delta_i^{\mathcal{M}}$, is computed as $\delta_i^{\mathcal{M}} = \bigwedge_{\alpha \in op_i} r(\alpha)$, where

$$r(\texttt{next}(v) := e) \triangleq (v' = e);$$
$$r(\texttt{next}(v) := \{e_1, e_2, \ldots, e_n\}) \triangleq \bigvee_{i=1}^{n} (v' = e_i); \text{ and}$$
$$r(\texttt{next}(v) := *) \triangleq (v' = *)$$

and v' denotes the next-state version of variable v.

The overall transition relation of the model $\mathcal{M}$ is determined by the form of composition expressed in *main*. For example, if *main* composes $\{op_1, op_2, \ldots, op_N\}$ asynchronously, then the overall transition relation is $\delta^{\mathcal{M}} = \bigvee_{i=1}^{N} \delta_i^{\mathcal{M}}$.

As an example, consider the model in Figure 2b. It is expressed as the tuple $(\mathcal{I_M}, \mathcal{V_M}, Init_{\mathcal{M}}, \mathcal{O_M})$, where $\mathcal{I_M} = \{cpl\}$, $\mathcal{V_M} = \{page\_fault\}$, $Init_{\mathcal{M}}$ is $\mathbf{true}$ (allowing arbitrary values to *cpl* and *page_fault*), and $\mathcal{O_M}$ has a single operation *op* which in turn contains only the following next-state assignment:

$$\texttt{next}(page\_fault) := \texttt{ITE}(cpl[0] = 1 \wedge cpl[1] = 1, 1, 0)$$

Environment: The environment $\mathcal{E}$ that provides inputs to the program is similarly modeled as a transition system, where the input variables in $\mathcal{M}$ are the state variables of $\mathcal{E}$. The final model that is to be verified is the composition of $\mathcal{M}$ and $\mathcal{E}$, written $\mathcal{M} \| \mathcal{E}$. The form of the composition depends on the context; both synchronous and asynchronous compositions are possible. However, for this paper, and in all of our examples, the environment $\mathcal{E}$ is stateless, generating completely arbitrary inputs to $\mathcal{M}$ at each step.

Transition Systems and Simulation: Notice that the original software system $\mathcal{S}$, the program $\mathcal{P}$, and its model $\mathcal{M}$ are all transition systems with the same basic form. Once composed with an environment model $\mathcal{E}$, the resulting transition system has the form $\mathcal{T} = (\mathcal{V}, \mathit{Init}, \delta)$, where $\mathcal{V}$ are the state variables, *Init* is the initial condition, and δ is the transition relation. A state s of $\mathcal{T}$ is a valuation to the variables in $\mathcal{V}$.

Given a transition system $\mathcal{T}$ and a set of variables $\mathcal{V}_L \subseteq \mathcal{V}$, each state s of $\mathcal{T}$ can be labeled with a valuation to variables in $\mathcal{V}_L$. We denote this labeling as $\mathcal{L}(s)$.

Given two transition systems $\mathcal{T}_1$ and $\mathcal{T}_2$ sharing a common labeling function $\mathcal{L}$, $\mathcal{T}_1$ is said to *simulate* $\mathcal{T}_2$ if there exists a *simulation relation* $\mathcal{H}$ relating states in $\mathcal{T}_1$ and $\mathcal{T}_2$ with the following properties:

(i) $\forall s_1, s_2.\mathcal{H}(s_1, s_2) \implies \mathcal{L}(s_1) = \mathcal{L}(s_2)$
(ii) $\forall s_1, s_2, s_2'.[\mathcal{H}(s_1, s_2) \wedge \delta_2(s_2, s_2')] \implies \exists s_1'.\delta_1(s_1, s_1') \wedge \mathcal{H}(s_1', s_2')$
(iii) $\forall s_2.\mathit{Init}_2(s_2) \implies (\exists s_1.\mathit{Init}_1(s_1) \wedge \mathcal{H}(s_1, s_2))$

We write "$\mathcal{T}_1$ simulates $\mathcal{T}_2$" as $\mathcal{T}_2 \preceq \mathcal{T}_1$.

A slight variation is the notion of *stuttering simulation*. We say that $\mathcal{T}_1$ *stutter simulates* $\mathcal{T}_2$ if there exists a relation $\mathcal{H}_{\text{st}}$ between states in $\mathcal{T}_1$ and $\mathcal{T}_2$ where the second condition above is modified as follows:

(ii) $\forall s_1, s_2, s_2'.[\mathcal{H}(s_1, s_2) \wedge \delta_2(s_2, s_2')] \implies [\mathcal{H}(s_1, s_2') \vee \exists s_1'.\delta_1(s_1, s_1') \wedge \mathcal{H}(s_1', s_2')]$.[3]

We write "$\mathcal{T}_1$ stutter simulates $\mathcal{T}_2$" as $\mathcal{T}_2 \preceq_{\text{st}} \mathcal{T}_1$.

B. Problem Definition and Approach

Models are always constructed based on the properties to be verified. Therefore, a model must be validated *with respect to a specific* property Φ.

In this paper, we focus on the verification of temporal safety properties of the form $\mathbf{G}\phi$, where $\mathbf{G}$ is the temporal operator "always" and ϕ is a state predicate, i.e., a Boolean expression over state variables with no temporal operators.

The overall model validation problem definition is as follows:

Definition 1 (Software Model Validation): Consider the transition systems $\mathcal{T}_\mathcal{S}$ formed by composing $\mathcal{S}$ and $\mathcal{E}$, and $\mathcal{T}_\mathcal{M}$ formed by composing $\mathcal{M}$ and $\mathcal{E}$. Determine whether $\mathcal{T}_\mathcal{M}$ satisfies Φ only if $\mathcal{T}_\mathcal{S}$ satisfies Φ.

[3]Note that this definition is a slight variant of the standard one which allows any finite number of stuttering steps in $\mathcal{T}_1$, but we chose this for simplicity and since it fits our problem context.

This paper takes a particular approach towards solving this problem, which can be formalized using the notion of (stuttering) simulation. Specifically, we use the following result (a proof of which one may find in any standard book on model checking, such as the one by Clarke et al. [6]).

Proposition 1: Given two transition systems $\mathcal{T}_1$ and $\mathcal{T}_2$, and a property Φ of the form $\mathbf{G}\phi$. If, using a labeling function $\mathcal{L}$ based on variables appearing in ϕ, $\mathcal{T}_2 \preceq \mathcal{T}_1$, then $\mathcal{T}_2$ satisfies Φ if $\mathcal{T}_1$ satisfies Φ.

The above result applies also when one uses stuttering simulation instead of "plain" simulation.

Let $\mathcal{V}^* \subseteq \mathcal{V}_\mathcal{S} \cup \mathcal{I}_\mathcal{S}$ be a set of variables deemed to be *relevant* to proving or disproving that $\mathcal{S}$ satisfies Φ. For soundness, $\mathcal{V}^*$ must contain all variables that influence the value of ϕ. However, this set of variables is difficult to determine exactly. Instead, we generate $\mathcal{V}^*$ based on the "relevant" code fragments left after pruning.

At a minimum, we include in $\mathcal{V}^*$ the set $\mathcal{V}_\phi$ of variables that syntactically appear in ϕ. Conservatively, the rest of $\mathcal{V}^*$ could be computed syntactically from the *cone of influence* of $\mathcal{V}_\phi$ on the entire code base [6]. However, this can lead to a highly over-approximate set $\mathcal{V}^*$. An alternative is to compute $\mathcal{V}^*$ as the syntactic cone-of-influence of $\mathcal{V}_\phi$ only on the code *excluding* f_{misc}. In this latter case, if we additionally verify that $\mathcal{V}^*$ is not modified in f_{misc}, we can conclude that there is no code in f_{misc} that influences the value of ϕ. This is the approach we adopt in the present paper.

Label the states of $\mathcal{T}_\mathcal{S}$ and $\mathcal{T}_\mathcal{M}$ using valuations to the variables in $\mathcal{V}^*$. Then, the approach of this paper can be outlined as follows:

1. Transform $\mathcal{S}$ to $\mathcal{A}$ by identifying fragments $\mathcal{F}_\mathcal{P}$ in $\mathcal{S}$;
2. From $\mathcal{A}$, obtain a new program $\mathcal{P}$ by dropping f_{misc} from the set $\mathcal{F}_\mathcal{A}$ and treating any invocation of f_{misc} from f_{orc} as a no-op (stuttering step);
3. Check that $\mathcal{S} \preceq_{\text{st}} \mathcal{P}$ using the labeling function from $\mathcal{V}^*$;
4. Create a model $\mathcal{M}$ of $\mathcal{P}$ (manually or automatically), where each operation op_i has a 1-1 correspondence with a code fragment f_i in $\mathcal{P}$, and
5. Validate $\mathcal{M}$ by checking that: (i) $\mathcal{I}_\mathcal{M} = \mathcal{I}_\mathcal{P}$; (ii) $\mathcal{V}_\mathcal{M} \supseteq \mathcal{V}_\mathcal{P}$; (iii) $\mathit{Init}_\mathcal{M}$ is equivalent to $\mathit{Init}_\mathcal{P}$ when projected on the common variables, and (iv) $\mathcal{P} \preceq \mathcal{M}$ using the labeling function based on $\mathcal{V}^*$.

The third step above is implemented as data-centric model validation (DMV), and the fifth step is operation-centric model validation (OMV). Note that we allow $\mathcal{M}$ to have more variables than $\mathcal{P}$ (and $\mathcal{S}$), since models often have extra "specification" variables for use in the proof. If the two simulation checking steps pass, then we can conclude that $\mathcal{M}$ stutter simulates $\mathcal{S}$ and therefore satisfies Φ only if $\mathcal{S}$ satisfies Φ. Note that this is only true for safety properties, which is what we consider in this paper. It is possible for a model that stutter simulates $\mathcal{S}$ to satisfy a liveness property, even though $\mathcal{S}$ does not.

The two simulation checking steps are the most important ones. We implement them as follows:

- *Checking if $\mathcal{S} \preceq_{st} \mathcal{P}$:* To do this, it is sufficient to verify that f_{misc} does not modify the values of variables in $\mathcal{V}^*$. There are several ways to do this, and we take the approach of using an assertion checking tool based on symbolic execution, described further in Sec. IV.
- *Checking if $\mathcal{P} \preceq \mathcal{M}$:* For this, we check that the transition relation $\delta_i^{\mathcal{M}}$ of each op_i overapproximates that of the corresponding code fragment f_i, denoted $\delta_i^{\mathcal{P}}$. The transition relations are computed using symbolic execution, and the check $\delta_i^{\mathcal{P}} \implies \delta_i^{\mathcal{M}}$ is discharged using SMT solving.

In theory, our methodology is *sound*, meaning that we never wrongly conclude that a model $\mathcal{M}$ satisfies Φ when the system $\mathcal{S}$ does not. However, in practice, our implementation has encountered significant practical limitations of symbolic execution tools, especially in the DMV step. Due to this, the implementation described in the following sections is a bug-finding tool rather than a verifier. Nevertheless, we demonstrate that it is useful in practice in weeding out various kinds of bugs in our models.

C. Example

Consider applying our problem definition to validate the model in Figure 2b against the program in Figure 2a. Assume that we have already performed the DMV step and determined that `update_access_dirty` does not modify any relevant state variables.

Our OMV methodology then produces the following two SMT queries, a query for each pair $(\pi, \mathcal{V_P}')$ in the program's symbolic summary. The model is valid iff both queries are proved correct by the backend solver.

1) $(page\_fault' = \texttt{ITE}(cpl[0] = 1 \wedge cpl[1] = 1, 1, 0)$ $\wedge\, cpl = 3) \Rightarrow (page\_fault' = 1)$
2) $(page\_fault' = \texttt{ITE}(cpl[0] = 1 \wedge cpl[1] = 1, 1, 0)$ $\wedge\, cpl \neq 3) \Rightarrow (page\_fault' = 0)$

The first query is proven valid by our SMT solver. However, the second query is invalid, indicating that our model does not correctly capture the behavior of the code. The returned counter-example is this: when $cpl = 7$, the two low-order bits are both set, and therefore, $page\_fault$ is set. The model makes an implicit assumption that cpl will never be higher than 6. This seems reasonable, considering a physical CPU's CPL is never higher than 3. However, the C code enforces no such assumption, and so the model is incorrect.

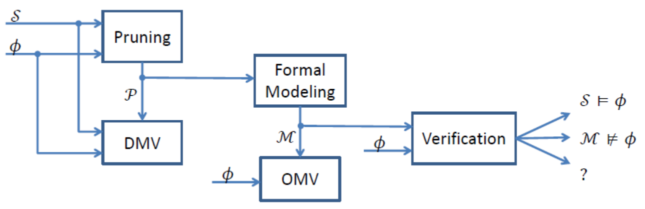

Figure 4: The five steps in our model validation process.

IV. Implementation

Section III-B outlined our formal approach to validating a model. This section describes our practical implementation. Figure 4 depicts the data flow of the process.

A. Pruning the System

The pruning step of our implementation corresponds to steps 1 and 2 of the outline given in Section III-B. In our implementation, the pruning stage is performed manually by a domain expert. Given the system $\mathcal{S}$ and the property Φ, the expert decides which code fragments are relevant and creates $\mathcal{A}$, with the set $\mathcal{F_P}$ identified. The expert then returns $\mathcal{P}$, in which code in f_{misc} has been replaced with no-ops.

B. DMV

Although pruning is performed by experts with knowledge of $\mathcal{S}$, it is error-prone. Some function or module in $\mathcal{S}$ that is relevant to Φ may get overlooked and be mislabeled as f_{misc} during annotation of $\mathcal{A}$. DMV takes as inputs $\mathcal{S}$, Φ, and $\mathcal{P}$ and returns a Yes/No answer, indicating whether $\mathcal{P}$ and $\mathcal{S}$ are equivalent with respect to Φ. If the answer is No, the domain expert repeats the pruning stage until DMV can return Yes. The step corresponds to step 3 of the theoretical approach given in Section III-B. In that step we prove $\mathcal{S} \preceq_{\text{st}} \mathcal{P}$ by checking that f_{misc} does not modify the values of variables in $\mathcal{V}^*$. In our implementation, we relax this requirement and check only for a subset of $\mathcal{V}^*$. As a result, we can not prove the simulation, only find instances when it fails.

We start by identifying the program variables $\mathcal{V}_\phi$ that the property Φ *directly* depends on. This can be done syntactically: the variables in $\mathcal{V}_\phi$ are those variables mentioned in ϕ. We wish to establish that any state change described in f_{misc} does not affect, directly or indirectly, the state described by $\mathcal{V}_\phi$; i.e., those pruned state changes are *stuttering steps* with respect to $\mathcal{V}_\phi$. To do this, we attempt to show variables in $\mathcal{V}_\phi$ do not change value within f_{misc}. Intuitively, we tackle this by keeping track of the last valuation of $\mathcal{V}_\phi$ and ensuring that it does not change after any statement that appears in f_{misc}.

More specifically, we create a system $\mathcal{S}'$ that is identical to $\mathcal{S}$, except for the following modifications. For each variable

$v \in \mathcal{V}_\phi$, we create a new shadow variable v' in $\mathcal{S}'$. After every (atomic) statement in $\mathcal{S}'$ that is also in $\mathcal{P}$, we add a statement that assigns the value of v to v'. In other words, we attempt to keep the value of v' synchronized with the value of v across $\mathcal{P}$ statements in $\mathcal{S}'$. In contrast, after every (atomic) statement in $\mathcal{S}'$ that is a statement in f_{misc}, we add an assertion that v equals v' for each $v \in \mathcal{V}_\phi$, which asserts that these statements do not write to any variable in $\mathcal{V}_\phi$, or equivalently, that every statement in f_{misc} is a stuttering step with respect to $\mathcal{V}_\phi$. Note that, since v' variables do not appear in $\mathcal{S}$ and do not alter the control flow, they do not alter the behavior of $\mathcal{S}'$ with respect to $\mathcal{S}$ state.

We then use KLEE to search for a path through $\mathcal{S}'$ where one of the inserted assertions fails. If KLEE is successful, we have found a f_{misc} statement that updates $\mathcal{V}_\phi$, so our DMV algorithm returns No. The path and inputs that cause the violation are passed back to the pruning stage, allowing the expert to refine the definition of $\mathcal{P}$ to incorporate the previously missed $\mathcal{V}_\phi$ update.

In practice, the extra statements in $\mathcal{S}'$ increase by $|\mathcal{V}_\phi|$ the size of the original system $\mathcal{S}$, which complicates and prolongs the execution time of the dynamic analysis performed by KLEE or other symbolic-execution engines. What is more, many of the added statements are ineffectual: in most practical systems, $\mathcal{V}_\phi$ will not be updated in every single statement, so checking even one assertion per statement can be overkill. To ease the burden on the symbolic execution engine, our prototype relaxes the strict definition of $\mathcal{S}'$, aiming to capture commonly missed $\mathcal{V}_\phi$ updates, but at the expense of soundness. Specifically, our prototype adds $v' := v$ statements into $\mathcal{S}'$ only after $\mathcal{P}$ statements that the domain expert knows to be updating v. Furthermore, our prototype adds $v' = v$ assertions into $\mathcal{S}'$ only before $\mathcal{P}$ statements known to be updating v (contrast that to the original definition, which adds assertions after every f_{misc} statement). The rationale for this last relaxation is that updates to v' are good enforcement points for the equality between v' and v, and this requires fewer changes to the program. Finally, our prototype checks the $v' = v$ assertion at all exit points from $\mathcal{S}'$, to capture any $\mathcal{V}_\phi$ updates on execution suffixes that do not include a known $\mathcal{P}$ update.

Our prototype's relaxation is unsound. It may miss successive updates to a v between known updates to it; if a statement reads such a missed v update, affecting the result of $\mathcal{P}$, our validation down the pipeline may be unsound as well. Since our prototype is a bug-finding tool inspired by our idealized methodology, this unsoundness is acceptable for our purposes. A more effective, sound relaxation would be to assert that $v = v'$ before every *read* of v, but finding all reads of v is, in itself, another hard problem, with similarities to alias analysis. A promising direction for future work would be to modify the KLEE symbolic execution engine to include watchpoints on symbolic state, effectively checking that $v = v'$ after every statement of f_{misc} (or some optimized version that is semantically equivalent but that adds the fewest extra statements and assertions necessary). The use of dynamic slicing [29] may also be useful. Initial experiments with a Hoare-style, deductive verifier indicate that symbolic execution may be better suited to this type of verification task, but we leave to future work a more rigorous comparison of the two methods.

C. Modeling the Program

From the selected $\mathcal{P}$, a model $\mathcal{M}$ is built using UCLID's modeling language. We are agnostic as to how the modeling is done; it can be a manual or an automated process. This corresponds to step 4 of the outline given in Section III-B.

D. Validating the Model

In the last step (step 5 of the outline in Section III-B), we validate the model $\mathcal{M}$ correctly simulates the program $\mathcal{P}$. We use KLEE for this step, as well. KLEE performs a combination of symbolic and concrete execution, based on which inputs and state bits are made symbolic. We use KLEE to learn $R = \{(\pi_\mathcal{P}, \mathcal{V}_\mathcal{P}')\}$, the set of path conditions and corresponding outputs describing the input–output relation of $\mathcal{P}$. First, we set all global state $\mathcal{V}_\mathcal{P}$ and input variables $\mathcal{I}_\mathcal{P}$ to be unconstrained, symbolic values. When execution reaches a conditional branch point, KLEE forks and follows all feasible branches. As KLEE explores these program paths, it maintains a path condition for each path that is a function of $\mathcal{V}_\mathcal{P} \cup \mathcal{I}_\mathcal{P}$. When a path terminates via exit statement or end of program, we note the path condition $\pi_\mathcal{P}$ for that path and the state of $\mathcal{V}_\mathcal{P}'$ at the point of termination. Once KLEE has explored all possible paths, we have R.

Once we have computed R, we use UCLID's decision procedure to verify that for every path condition describing a portion of the input space in $\mathcal{P}$, both $\mathcal{M}$ and $\mathcal{P}$ produce the same (symbolic) output. Our queries to the UCLID decision procedure take the form of $\texttt{decide}(\pi_\mathcal{P} \Rightarrow (\mathcal{V}_\mathcal{M}' = \mathcal{V}_\mathcal{P}'))$, and we have a separate query for each $(\pi_\mathcal{P}, \mathcal{V}_\mathcal{P}')$ pair in R. If one of the queries fails, UCLID will return a counterexample with concrete values for $\mathcal{I}_\mathcal{P}$ and $\mathcal{V}_\mathcal{P}$ that satisfy the path condition, but for which $\mathcal{V}_\mathcal{M}' \neq \mathcal{V}_\mathcal{P}'$.

Translating Path Conditions to SMT queries: In order to create each query, we must first translate the $\pi_\mathcal{P}$ generated by KLEE to a format suitable for UCLID's input language. KLEE's path conditions are written in KQuery[4], the input language to KLEE's backend constraint solver (Kleaver). The symbolic states of $\mathcal{V}_\mathcal{P}$ are given in a similar syntax. We built a tool to translate path conditions and symbolic state

[4]http://klee.llvm.org/KQuery.html

```
1 int loop_prog(int bound)  {
2    int retval = 0;            //E2L, lines 2,3,4
3    if (bound <= 0)
4       return −1;
5    for (int i = 0; i < bound; i++)
6       retval++;               //L2L, lines 5,6
7    return retval;             //L2E
8 }
```

Figure 5: A program with a dynamically-determined loop bound. Validation of the model is done in three parts.

from KLEE into SMT queries in UCLID's input language. Our tool requires the user to provide an input file that maps variable names used in $\mathcal{P}$ to variable names used in $\mathcal{M}$. All variables in KQuery are represented as an array of bits; these are easily translated to UCLID's bitvector type. UCLID allows two additional types: Boolean, and uninterpreted functions. Currently, we require UCLID models to not use Boolean types; bitvectors of length 1 can be used instead. Seamlessly converting between the two is functionality that can be added to future versions of our tool. UCLID models are free to use uninterpreted functions, but we do not handle verification with respect to properties that include universal quantification over an uninterpreted function. For example, if f is an uninterpreted function in a UCLID model and i is a bitvector variable in that model, our tool can validate a model with respect to the property $\phi(f(i))$, but would not be able to validate a model with respect to the property $\phi(f)$.

Handling Loops: For programs that contain loops, the number of paths possible through the code can explode quickly. In cases where the loop bound can be determined statically, this is not a problem, but when the loop bound is determined at run time, complete path coverage requires exploring all possible loop bounds, causing an exponential blow-up in the number of paths through the program.

In some cases we can handle this using a divide & conquer approach. Figure 5 illustrates the idea. For a program $\mathcal{P}$ with a single loop, we can divide the source code of the program into three parts: the code from program entry to the loop, the loop code itself, and the code from loop exit to program exit. In Figure 5, these sections are labled E2L, L2L, and L2E, respectively. We then explore each section of code independently of the other two. At the start of each exploration we set the current constraints on $\mathcal{I}_{\mathcal{P}}$ and $\mathcal{V}_{\mathcal{P}}$ to reflect the invariants of the previous section(s). These invariants are determined manually and care must be taken to not over-constrain the starting conditions of each section. At a minimum, the following weak invariants can safely be used: no constraints set for E2L, the loop guard forms the constraints for L2L, and no constraints set for L2E. These constraints will form part of the path conditions created during subsequent exploration of the section. For example, in Figure 5, execution of E2L starts with no constraints on *bound* or *retval* and the starting path condition is $\pi_{\text{E2L}} = \texttt{True}$; execution of L2L is started with the path condition (and corresponding constraints) $\pi_{\text{L2L}} = (i \geq 0) \wedge (i < \mathit{bound})$; and execution of L2E is started with no constraints.

We divide the model into three parts as well. We create a separate model for the E2L, L2L, and L2E program fragments, and separately validate each model against its corresponding program fragment. Because each fragment is explored starting from an over-approximation of reachable starting states, the explored paths through the fragments will constitute a superset of reachable paths in the original program. If the model is shown to be consistent with each of these paths, and the corresponding output, a property proven true of the model will be true of the program. However, this method is not complete, and a property that is disproven for the model may in fact be true of the program.

In some cases, it may be possible to strengthen the invariant through manual inspection of the code. For example, in Figure 5, the execution of L2L can be started with the path condition $\pi_{\text{L2L}} = (i \geq 0) \wedge (i < \mathit{bound}) \wedge (\mathit{bound} > 0) \wedge (\mathit{retval} \geq 0)$, and execution of E2L can be started with the path condition $\pi_{\text{L2E}} = (i \geq \mathit{bound}) \wedge (\mathit{bound} > 0) \wedge (\mathit{retval} \geq 0)$. This will increase the completeness of model validation for some programs, but still does not guarantee it. Furthermore, our divide & conquer approach is suitable to code with single loops, but is not applicable to code with nested loops or recursion.

V. Evaluation: Data-Centric Validation

A. BPF

The Berkeley Packet Filter (BPF) is a kernel module that filters network packets, sending only the desired ones to the user. The user provides a filter program, written in the BPF pseudo-machine language. The BPF kernel module contains an interpreter that runs the filter program on each network packet. The property Φ asserts that the BPF interpreter (*bpf_filter*) has a monotonically increasing program counter; this is desirable, since it implies that all filter programs must eventually halt. One challenge is that it is not practical to explore all reachable states by running the BPF interpreter on all filter programs of n fully-symbolic instructions. Therefore, we over-approximate its behavior: we run one iteration of the interpreter loop on a single fully-symbolic BPF instruction, starting from a fully symbolic state.

In particular, we made the accumulator, the index registers, the temporary variable k, and the memory fully symbolic. The inputs were fully symbolic two-instruction BPF programs that had passed the *bpf_validate* function (Figure 6).[5]

[5]A valid BPF program must end with a *return* instruction, so our input approximates a fully symbolic "one-instruction" program.

```
int bpf_validate(filter, len) {
    for (i = 0; i < len; ++i) {
        switch (BPF_CLASS(filter[i].code)) {
            case BPF_ST:
                if (filter[i]−>k >= BPF_MEMWORDS)
                    return 0;
                break;
            case BPF_DIV:
                // Check for division by 0.
                if (filter[i]−>k == 0)
                    return 0;
                break;
            ...
        }
    }
    return BPF_CLASS(filter[len−1].code) == BPF_RET;
}
```

Figure 6: Simplified code from *bpf_validate*, which checks that the BPF program cannot write out-of-bounds, cannot divide by zero, and ends with a *return*.

```
pass (const char ∗passwd) {
    if (cred.logged_in || askpasswd == 0)
        return;
    askpasswd = 0;
    cred.logged_in = 1;    }

user (const char ∗name) {
    if (cred.logged_in)
        end_login (&cred);
    askpasswd = 1;        }

end_login (struct credentials ∗pcred)    {
    memset (pcred, 0, sizeof (∗pcred)); }
```

Figure 7: The *PASS* and *USER* commands (simplified code).

We encountered no spurious counter-examples. We were able to efficiently explore all paths of *bpf_filter* using KLEE (in a few seconds) and confirm that we had identified all writes to the program counter. Thus, our data-centric model validation was successful.

B. *ftpd*

ftpd is GNU's File Transfer Protocol server. The main loop reads input from the network client, parses the command, and runs the appropriate block of code (which might be inline or in a separate function) to execute the command.

We sought to model the blocks of code that affect whether the user is considered to be logged in. There are many FTP commands, but we would expect very few to be relevant. A simple syntactic approach of searching for *logged_in =* identifies only the *PASS* function (Figure 7), so we used DMV to check that there are no other writes to *logged_in*.

We modified the *ftpd* software so that its input was partly symbolic: it could choose to explore any of *USER* and *PASS* (with a prespecified, known-good username and password), nine other commands that had no parameters, and nine commands with partly symbolic parameters (two alphanumeric characters). We excluded other commands because it was difficult to make their parameters symbolic while respecting validity constraints, because they were not implemented by *ftpd*, or because they modified the filesystem (e.g., *RMD*), which is not well handled with KLEE. In some cases, we also modified functions (e.g., *CWD*) to remove system calls or side effects. We then added the shadow credentials data structure, copying *cred* to *shadow_cred* after the known write to *logged_in*, and placed *assert (cred.logged_in == shadow_cred.logged_in);* before the write and in the body of the parser loop.

We started the analysis from a partly symbolic state, with the *askpasswd* global variable and all of the credentials data structure made symbolic (for each string, we stored two symbolic characters). KLEE produced a test case where the assertion failed: when the client is already logged in, and issues another *USER* command, the entire credentials data structure is zeroed out by *end_login()* (Figure 7). This is non-trivial to identify syntactically.

After adding the update to *shadow_cred.logged_in* after the write in the *user* function, we verified with DMV that we had not missed any writes. In a separate experiment, we confirmed that DMV would also have been able to identify a missing write to *cred.logged_in* in the *PASS* function. Each experiment required less than 3 minutes of run-time.

C. *XMHF hypervisor*

The eXtensible Modular Hypervisor Framework (XMHF) [30] is a hypervisor with a small trusted core (about 6000 LoC) that is amenable to formal verification, and Vasudevan et al. have proven a memory integrity property of XMHF [30]. A hypervisor runs at a higher privilege level, and supports the execution of guest software running at a lower privilege level. For that reason, whenever a guest executes a privileged instruction (e.g., to set the CR3 register), the CPU transfers control to the hypervisor, which uses an intercept handler (see Figure 8) to take the necessary action.

```
void interceptHandler(VCPU ∗vcpu, struct registers ∗regs) {
  switch(vcpu−>vmcs.vmexit_reason) {
    case CRX_ACCESS: handle_crx_access(vcpu, regs); break;
    case IO: handle_ioport_access(vcpu, regs); break;
    case WRMSR: handle_wrmsr(vcpu, regs); break;
    case EPT_VIOLATION: handle_pagetable(vcpu, regs); break;
    ...  }}
```

Figure 8: Simplified code for XMHF intercept handler.

Consider the following property: the extended page tables must remain constant after setup. This property requires us to model all program paths in *inter-*

ceptHandler that might update relevant state such as the *vcpu→vmx_vaddr_ept_pml4_table* field, which shadows the address pointing to the base of the extended page tables. Instead of modeling all intercept handlers, we prune away handlers that do not appear to update the *vcpu→vmx_vaddr_ept_pml4_table* field and then use DMV to check that we did not miss any updates. Forcing the input data structures *vcpu* and *regs* to be symbolic, we use KLEE to symbolically explore paths in *interceptHandler*. DMV found no writes to the *vcpu→vmx_vaddr_ept_pml4_table* field in each handler that the expert decided to prune away. This allows us to prune away the following handlers: set CR0 register, set CR4 register, read MSR register, write MSR register, read CPU id, access IO ports. This experiment used KLEE to evaluate 300 LoC (in C) in 5 seconds.

Note that the intercept handlers rely on inlined assembly instructions to perform certain operations such as reading the MSR register. Since we do not model the x86 hardware, we assume that none of the inlined assembly operations impact the *vcpu→vmx_vaddr_ept_pml4_table* field. While we are confident of this assumption, we do recognize this as a potential weakness of this case study.

VI. Evaluation: Operation-Centric Validation

A. Bochs address translation function

Bochs [21] is an open-source C++ x86 emulator. The code base is large and previous research has shown that manual analysis and testing, while useful, are not enough to guarantee correctness [19], [20]. At the same time, new techniques have shown that model checking can be a practical approach to verifying the correctness of the Bochs CPU emulator and other virtualization technologies like it [27]. In this case study, we validate the model of the Bochs address translation function used by Sinha et al. [27], who verified whether the function returned the correct physical address when the TLB optimization was used.

The address translation function in Bochs (f_1) is 98 LoC and the corresponding UCLID model is about 300 LoC. There are three input variables $\mathcal{I}_{\mathcal{P}}$: the linear address to be translated, the read or write permissions requested of the address, and the emulated CPU's current privilege level. We make symbolic the input variables, the variables $\mathcal{V}_{\mathcal{P}}$ (which consist of the two page table entries and the single TLB entry that the translation could access), and all global variables.

During symbolic execution, KLEE explored 219 paths through the code in roughly nine seconds. KLEE achieves full code coverage of the function, as seen in Table I. There is one branch not taken by KLEE; manual analysis reveals that taking this branch would not cover any new state in the code.

UCLID discharged the queries comparing each path condition to the model in under a minute and found seven discrepancies between the behavior of the source code and the behavior of the model. Each discrepancy was ultimately due to a bug in the model. For this and the other case studies, while running UCLID was quick, finding the root cause of each discrepancy was a manual process that took anywhere from a few minutes to close to an hour, per bug. We categorize these bugs by the likely cause of the modeling error: there was one typo, two implicit assumptions, and four logic errors. Our running example (Section II) corresponds to one of these bugs in the model, in which the modeler assumed the CPU privilege level was never higher than 3. The extended version of this paper [28] also shows the source and model code for six of the bugs.

B. TCAS

Traffic Collision Avoidance Software (TCAS) [9], [18] is an aircraft collision avoidance system. The TCAS software is complex and safety-critical, and verification of its correctness is well-suited to model checking. In this case study, we used a simplified and publicly available version of TCAS that the software engineering research community has used to evaluate the efficacy of various test case generation techniques [23]. Though only 133 LoC (in C), the TCAS code includes 9 functions and complicated control flow.

From the TCAS code, we created 23 different models, each injected with a different fault developed by Hutchins et al. [13]. We validated each model against the TCAS source code, checking whether model validation was able to find the injected fault.

We made symbolic all 12 inputs to the TCAS program. During symbolic execution, KLEE explored 29 paths through the code and completed in less than a second. There was one line of code not covered by KLEE (see Table I), and manual analysis revealed this line of code to be unreachable. There were eight branches not taken by KLEE and these were either infeasible or redundant branches.

Our model validation found the errors in all 23 models, and took roughly 5 seconds per model. It is difficult to categorize these bugs since they did not naturally arise during the translation from code to model. However, they included: using a less-than-or-equal ($\leq$) sign in place of a less-than sign ($<$), or vice versa; using a logical AND (&) in place of a logical OR (|), or vice versa; leaving an entire clause out of a series of conjunctions and disjunctions; and typographical errors such as setting a variable to the wrong value. In addition to the 23 injected faults, model validation also revealed two model bugs that were introduced unintentionally during the modeling phase (see Table II).

Program	Lines of Code	Lines Executed	Branches	Branches Taken
Bochs	98	98	30	29
TCAS	60	59	68	60
bpf_validate	71	69	28	24

Table I: Code and path coverage achieved during symbolic exploration of the code by KLEE. The total LoC given refers only to the pruned program, not the original system.

Program	Typo	Logic Error	Assumption about System Invariants	Total
Bochs	1	4	2	7
TCAS	1	1	0	2
bpf_validate	0	4	0	4

Table II: Types of bugs in the model found during operation-centric model validation. For TCAS, only bugs introduced unwittingly during the modeling phase are categorized.

C. BPF

Prior to running a filter, BPF runs a validator (Figure 6) to ensure the filter is valid and satisfies three properties: 1) all jumps in the program move forward (no loops are allowed), 2) all jumps move to a legitimate place in the program (i.e., do not jump past the end the program), and 3) the program always terminates with either an accept or reject. Validation before use is a common strategy for security-critical code.

In this case study, we show how one might verify Φ, that filter programs are marked valid if and only if they satisfy the three properties above. This case study demonstrates validation of non-statically bounded loops, which are not handled by model validation on hardware designs.

We use $\mathcal{F}_{\mathcal{P}} = \{\mathit{bpf\_validate}, f_{\text{misc}}, f_{\text{orc}}\}$. *bpf_validate* takes two inputs, $\mathcal{I}_{\mathcal{P}} = \{$filter program, filter program length$\}$. It has two local variables, $\mathcal{V}_{\mathcal{P}} = \{$current program instruction pointer, loop iteration counter$\}$. As *bpf_validate* has a single loop, it can be structured in three blocks: E2L, L2L, and L2E. Because the for-loop does not have a bound that can be determined statically, we use the divide & conquer technique for model validation outlined in Section IV-D.

Each iteration of the loop reads and checks a single instruction, and does not propagate any state between iterations. Therefore, we run a single iteration of the loop on a fully symbolic instruction, with the *len* parameter symbolic (but constrained to be less than or equal to the length of the program) as well.

Symbolic execution completed in less than 0.2 seconds, and the verification of path conditions against the model took less than 2 seconds. Model validation found four modeling bugs, all in the for-loop. Two of the bugs involved an inverted logic statement. The third and fourth were errors in calculating the return values, *return* and *return_happened*: both are computed using switch statements in the model, and in both cases there was a case missing.

VII. Related Work

There has been previous work on model validation in both the hardware and software verification communities. Much of the work focuses on checking that the model is functionally equivalent to the code (or at least overapproximate), similar to our OMV step. Many efforts are also focused on the case when the models and code are in the same programming language. Our work is distinguished from the previous work on two counts: (i) we combine OMV and DMV, checking both that the model does not miss relevant code and that it is functionally overapproximate, and (ii) we handle cases where the model and the code are in different languages with different underlying formalisms.

Several efforts [5], [7], [12], [15]–[17] have focused on validating a design written in a high-level language like C against a register transfer level (RTL) implementation written in a language such as Verilog or VHDL. The basic idea is similar to our operation-centric model validation: derive an expression, for both the C program and the RTL program, describing the input-output transition relation of the program and use symbolic execution and satisfiability solving to check equivalence between the two expressions. Past work assumes that the C function is written for eventual implementation in hardware and therefore some restrictions can be placed on the expressiveness of the C code that reduce the problem to one of checking equivalence of combinational circuits. These approaches typically do not have a step similar to our DMV step.

Model validation has not been studied as widely in the software community. Heitmeyer et al. present a manual, theorem-proving based approach to partitioning code for a separation kernel, so one can concentrate verification efforts only on those portions of code that are relevant to the property [10]. This is similar in spirit to our pruning phase, although our approach is more automated and our implementation is more focused on bug-finding. Their manual approach might not scale well to software systems as large as Bochs or XMHF. Static analysis techniques such as alias analysis, Mod/Ref analysis, and program slicing [8], [11], [29] might be helpful for performing DMV on large code bases, but at the cost of lower precision, potentially resulting in a high false alarm rate.

Caballero et al. [3] use a combination of concrete and symbolic exploration of a function at the binary level to find cases where the behavior of security-sensitive functions deviates from those of manually written models of those functions. Symbolic execution has also been used to compare two versions of the same code fragment or function, e.g., for checking code optimizations, version tracking, regression testing, and code refactorization [24], [26]. Similarly, compiler translation validation [22], [25] has similar objectives to OMV, although in translation validation there are significant similarities between the programs before and after compilation that can be exploited in the validation process; such similarities cannot be relied upon in our model validation setting.

VIII. Conclusion

We have proposed a formal framework for validating a model of a software system against the code it was created from. Using an implementation based on symbolic execution and SMT solving, we have demonstrated the utility of our approach on several case studies. Among the many directions for future work, we believe the most important is to improve the implementation of DMV, potentially by combining scalable static analysis methods with selective assertion checking.

Acknowledgment: This work was funded in part by the Intel Science and Technology Center for Secure Computing, by SRC grants 2045.001 and 2460.001, by NSF grant 0644436, and by the AFOSR under MURI award FA9550-09-1-0539.

References

[1] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, volume 4. IOS Press, 2009.

[2] 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 *CAV*, 2002.

[3] J. Caballero, S. McCamant, A. Barth, and D. Song. Extracting Models of Security-Sensitive Operations using String-Enhanced White-Box Exploration on Binaries. Technical Report UCB/EECS-2009-36, EECS Department, University of California, Berkeley, Mar 2009.

[4] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, 2008.

[5] E. Clarke and D. Kroening. Hardware Verification using ANSI-C Programs as a Reference. In *ASP-DAC*, 2003.

[6] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[7] X. Feng and A. J. Hu. Early Cutpoint Insertion for High-Level Software vs. RTL Formal Combinational Equivalence Verification. In *DAC*, 2006.

[8] B. Hardekopf and C. Lin. Flow-Sensitive Pointer Analysis for Millions of Lines of Code. In *CGO*, 2011.

[9] W. Harman. TCAS – A System for Preventing Midair Collisions. *The Lincoln Laboratory Journal*, 2, 1989.

[10] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean. Formal Specification and Verification of Data Separation in a Separation Kernel for an Embedded System. In *ACM CCS*, 2006.

[11] M. Hind and A. Pioli. Which pointer analysis should I use? In *ACM SIGSOFT Software Engineering Notes*, 2000.

[12] A. J. Hu. High-Level vs. RTL Combinational Equivalence: An Introduction. In *ICCD*, 2007.

[13] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the Effectiveness of Dataflow- and Control Flow-Based Test Adequacy Criteria. In *ICSE*, 1994.

[14] J. C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7), 1976.

[15] A. Koelbl, J. R. Burch, and C. Pixley. Memory Modeling in ESL-RTL Equivalence Checking. In *DAC*, 2007.

[16] A. Koelbl, R. Jacoby, H. Jain, and C. Pixley. Solver Technology for System-Level to RTL Equivalence Checking. In *DATE*, 2009.

[17] A. Koelbl and C. Pixley. Constructing efficient formal models from high-level descriptions using symbolic simulation. *Int. J. of Par. Prog.*, 33(6), 2005.

[18] J. Kuchar and A. C. Drumm. The Traffic Alert and Collision Avoidance System. *Lincoln Laboratory Journal*, 16(2), 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] D. Mihocka and S. Shwartsman. Virtualization without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure. In *ISCA WAMSBT*, 2008.

[22] G. C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5), 2000.

[23] T. Ostrand. TCAS. Software-artifact Infrastructure Repository. http://sir.unl.edu/portal/bios/tcas.php, Last checked May, 2013.

[24] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *FSE*, 2008.

[25] A. Pnueli, M. Siegel, and O. Shtrichman. The Code Validation Tool (CVT) – Automatic Verification of a Compilation Process. *STTT*, 2(2), 1998.

[26] D. A. Ramos and D. R. Engler. Practical, Low-Effort Equivalence Verification of Real Code. In *CAV*, 2011.

[27] R. Sinha, C. Sturton, P. Maniatis, S. A. Seshia, and D. Wagner. Verification with Small and Short Worlds. In *FMCAD*, 2012.

[28] C. Sturton, R. Sinha, T. H. Dang, S. Jain, M. McCoyd, W. Y. Tan, P. Maniatis, S. A. Seshia, and D. Wagner. Symbolic Software Model Validation. Technical report, EECS Department, University of California, Berkeley, 2013.

[29] F. Tip. A Survey of Program Slicing Techniques. *J. of Prog. Lang.*, 3(3), 1995.

[30] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. In *IEEE S&P*, 2013.