Symbolic Software Model Validation

Cynthia Sturton\textsuperscript{1}  \textbf{Rohit Sinha}\textsuperscript{2}  Thurston Dang\textsuperscript{2}
Sakshi Jain\textsuperscript{2}  Michael McCoyd\textsuperscript{2}  Wei Yang Tan\textsuperscript{2}
Petros Maniatis\textsuperscript{3}  Sanjit Seshia\textsuperscript{2}  David Wagner\textsuperscript{2}

\textsuperscript{1}University of North Carolina at Chapel Hill
\textsuperscript{2}University of California, Berkeley
\textsuperscript{3}Intel Labs

MEMOCODE 2013
Outline

1 Introduction
   - Motivation
   - Running example

2 Problem Definition
   - Work Flow
   - Theoretical Formulation
   - DMV
   - OMV

3 Methodology
   - DMV Algorithm
   - OMV Algorithm

4 Evaluation
   - DMV Case Studies
   - OMV Case Studies

5 Related Work
Motivation

Model Validation

- Modeling is a first step in formal verification. Model is either
  - hand written
  - automatically generated
    - e.g. HAVOC (C to Boogie),
    - e.g. C2UCL (C to UCLID)
Motivation

Model Validation

- Modeling is a first step in formal verification. Model is either
  - hand written
  - automatically generated
    - e.g. HAVOC (C to Boogie),
    - e.g. C2UCL (C to UCLID)

- Verification useless if model is incorrect
  - A valid model must allow all behaviors allowed by the modeled program.
Motivation
Model Validation

- Modeling is a first step in formal verification. Model is either
  - hand written
  - automatically generated
    e.g. HAVOC (C to Boogie),
    e.g. C2UCL (C to UCLID)

- Verification useless if model is incorrect
  - A valid model must allow all behaviors allowed by the modeled program.

- Want to prove that all states reachable in the original program are reachable in the model.
Motivation

Model Validation Subgoals

- Modelers typically only model portions of the system relevant to the property.
  - Prove that pruning step is correct
    i.e. unmodeled code has no effect on state variables relevant to the property.

- The model may contain incorrect or missing behaviours.
  - Prove that model encodes a superset of behaviours allowed in the original system.
Motivation

Model Validation Subgoals

- Modelers typically only model portions of the system relevant to the property.
  - Prove that pruning step is correct
    i.e. unmodeled code has no effect on state variables relevant to the property.

- The model may contain incorrect or missing behaviours.
  - Prove that model encodes a superset of behaviours allowed in the original system.
Running (Toy) Example

This example is extracted from the Address Translation module of Bochs CPU emulator. Our property requires us to model updates to page_fault.

C code

```c
u32 page_fault;
u32 translate(u32 curr_privilege_level)
{
    if (curr_privilege_level == 3)
        page_fault = 1;
    else
        page_fault = 0;
    ...
    update_access_dirty(...);
}
```
This example is extracted from the Address Translation module of Bochs CPU emulator. Our property requires us to model updates to `page_fault`.

**C code**

```c
u32 page_fault;
u32 translate(u32 curr_privilege_level)
{
    if (curr_privilege_level == 3)
        page_fault = 1;
    else
        page_fault = 0;
    ...
    update_access_dirty(...);
}
```

**UCLID model**

```plaintext
INPUT
cpl : BITVEC[32];
VAR
    page_fault : BITVEC[32];
ASSIGN
    next[page_fault] := (cpl[0] & cpl[1]);
```
Running (Toy) Example

This example is extracted from the Address Translation module of Bochs CPU emulator. Our property requires us to model updates to `page_fault`.

**C code**

```c
u32 page_fault;
void translate(u32 curr_privilege_level)
{
    if (curr_privilege_level == 3)
        page_fault = 1;
    else
        page_fault = 0;
    ...
    update_access_dirty(...);
}
```

**UCLID model**

```plaintext
INPUT
cpl : BITVEC[32];
VAR
page_fault : BITVEC[32];
ASSIGN
next[page_fault] := (cpl[0] & cpl[1]);
cpl = 0x00000007
```
Outline

1 Introduction
   • Motivation
   • Running example

2 Problem Definition
   • Work Flow
   • Theoretical Formulation
   • DMV
   • OMV

3 Methodology
   • DMV Algorithm
   • OMV Algorithm

4 Evaluation
   • DMV Case Studies
   • OMV Case Studies

5 Related Work
**Problem Definition**

**Work Flow**

1. **Prune system**: Before modeling program \( \mathcal{P} \), identify program fragments \( \mathcal{F}_\mathcal{P} \) relevant to property \( \Phi \).

2. **Data-Centric Model Validation (DMV)**: Is \( \mathcal{F}_\mathcal{P} \) missing any relevant code?

3. **Model Construction**: Build a formal model \( \mathcal{M} \) of \( \mathcal{F}_\mathcal{P} \).

4. **Operation-Centric Model Validation (OMV)**: Is \( \mathcal{M} \) a sound abstraction of \( \mathcal{F}_\mathcal{P} \)?

5. **Verification**: Does \( \mathcal{M} \) preserve \( \Phi \)?
Definition (Software Model Validation)
Consider the transition system $T_P$ formed by composing program $P$ and its environment $E$, and $T_M$ formed by composing $M$ and $E$. Determine whether $T_M$ satisfies $\Phi$ only if $T_P$ satisfies $\Phi$. 
Theoretical Formulation: Pruning

Expert decomposes $\mathcal{P}$ into a finite set of code fragments $\mathcal{F} = \{f_1, f_2, \ldots, f_N, f_{\text{misc}}, f_{\text{orc}}\}$. 
Theoretical Formulation: Pruning

Expert decomposes $\mathcal{P}$ into a finite set of code fragments $\mathcal{F} = \{f_1, f_2, \ldots, f_N, f_{\text{misc}}, f_{\text{orc}}\}$. $\mathcal{F}_\mathcal{P} = \{f_1, f_2, \ldots, f_N\}$ are code fragments that capture the relevant parts of $\mathcal{P}$. 

Rohit Sinha
Symbolic Software Model Validation
MEMOCODE 2013 9 / 29
Expert decomposes $\mathcal{P}$ into a finite set of code fragments $\mathcal{F} = \{ f_1, f_2, \ldots, f_N, f_{\text{misc}}, f_{\text{orc}} \}$.

- $\mathcal{F}_\mathcal{P} = \{ f_1, f_2, \ldots, f_N \}$ are code fragments that capture the relevant parts of $\mathcal{P}$.
- $f_{\text{misc}}$ contains code in $\mathcal{P}$ deemed irrelevant to $\Phi$. 
Theoretical Formulation: Pruning

Expert decomposes $\mathcal{P}$ into a finite set of code fragments $\mathcal{F} = \{ f_1, f_2, \ldots, f_N, f_{\text{misc}}, f_{\text{orc}} \}$.

- $\mathcal{F}_\mathcal{P} = \{ f_1, f_2, \ldots, f_N \}$ are code fragments that capture the relevant parts of $\mathcal{P}$.
- $f_{\text{misc}}$ contains code in $\mathcal{P}$ deemed irrelevant to $\Phi$.
- $f_{\text{orc}}$ orchestrates how program execution interleaves between the code fragments $f_1, f_2, \ldots, f_N, f_{\text{misc}}$.
  - e.g. sequential composition of code fragments
  - e.g. while(1) loop repeatedly selecting the next code fragment to run
Theoretical Formulation: Pruning Running Example

Bochs Address Translation

```c
if (curr_privilege_level == 3)
    page_fault = 1;
else
    page_fault = 0;
...
update_access_dirty(...);
```
Theoretical Formulation: Pruning Running Example

Bochs Address Translation

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

- Inputs $I_P = \{ curr\_privilege\_level \}$
Bochs Address Translation

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

- Inputs $I_P = \{\text{curr}_\text{privilege}_\text{level}\}$
- State Variables $V_P = \{\text{page}_\text{fault}\}$
Theoretical Formulation: Pruning Running Example

Bochs Address Translation

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

- Inputs $I_P = \{curr\_privilege\_level\}$
- State Variables $V_P = \{page\_fault\}$
- Initial state constraint $Init_P = true$
Theoretical Formulation: Pruning Running Example

Bochs Address Translation

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

- Inputs $I_P = \{ curr\_privilege\_level \}$
- State Variables $V_P = \{ page\_fault \}$
- Initial state constraint $Init_P = true$
- $F = \{ f_1, f_{misc}, f_{orc} \}$
  - $F_P = \{ f_1 \}$ includes the if-then-else block.
  - $f_{misc}$ includes all the code outside of $f_1$, including the function `update_access_dirty`.
  - $f_{orc}$ is the sequential composition $f_1; f_{misc}$.
Theoretical Formulation: Pruning

Expert decomposes $\mathcal{P}$ into a finite set of code fragments $\mathcal{F} = \{f_1, f_2, \ldots, f_N, f_{\text{misc}}, f_{\text{orc}}\}$.

- $\mathcal{F}_\mathcal{P} = \{f_1, f_2, \ldots, f_N\}$ are code fragments that capture the relevant parts of $\mathcal{P}$
- $f_{\text{misc}}$ contains code in $\mathcal{P}$ deemed irrelevant to $\Phi$.
- $f_{\text{orc}}$ orchestrates how program execution interleaves between the code fragments $f_1, f_2, \ldots, f_N, f_{\text{misc}}$
  - e.g. sequential composition of code fragments
  - e.g. while(1) loop repeatedly selecting the next code fragment to run

**Assumption**: each of $f_1, f_2, \ldots, f_N$ has a unique entry and exit point, executes atomically, and is terminating.

**Assumption**: $f_{\text{misc}}$ contains terminating code.
Let $\mathcal{V}^* \subseteq \mathcal{V}_P \cup \mathcal{I}_P$ be a set of variables deemed to be relevant to proving or disproving that $P$ satisfies $\Phi$. 
Theoretical Formulation: DMV

- Let $\mathcal{V}^* \subseteq \mathcal{V}_P \cup \mathcal{I}_P$ be a set of variables deemed to be relevant to proving or disproving that $P$ satisfies $\Phi$.
- At the least, $\mathcal{V}^*$ contains the set $\mathcal{V}_\phi$ of variables that syntactically appear in $\phi$. 

Theoretical Formulation: DMV

Let $\mathcal{V}^* \subseteq \mathcal{V}_P \cup \mathcal{I}_P$ be a set of variables deemed to be relevant to proving or disproving that $P$ satisfies $\Phi$.

- At the least, $\mathcal{V}^*$ contains the set $\mathcal{V}_\phi$ of variables that syntactically appear in $\phi$.
- Conservatively, $\mathcal{V}^*$ includes the cone of influence of $\mathcal{V}_\phi$ on the entire code base.
Let $\mathcal{V}^* \subseteq \mathcal{V}_P \cup \mathcal{I}_P$ be a set of variables deemed to be relevant to proving or disproving that $P$ satisfies $\Phi$.

- At the least, $\mathcal{V}^*$ contains the set $\mathcal{V}_\phi$ of variables that syntactically appear in $\phi$.
- Conservatively, $\mathcal{V}^*$ includes the cone of influence of $\mathcal{V}_\phi$ on the entire code base.

Verify that $f_{misc}$ does not modify variables in $\mathcal{V}^*$. 
Theoretical Formulation: Model Construction

Model each relevant program fragment in $f_1, \ldots, f_N$.

- For each $f_i$, generate $op_i$ modeling the next-state transition relation.
- We also model a scheduler process based on $f_{orc}$. It determines the allowed interleavings of operations $\{op_1, \ldots, op_N\}$. 
Theoretical Formulation: Model Construction

Model each relevant program fragment in $f_1, \ldots, f_N$.

- For each $f_i$, generate $op_i$ modeling the next-state transition relation.
- We also model a scheduler process based on $f_{orc}$. It determines the allowed interleavings of operations $\{op_1, \ldots, op_N\}$.

We use the UCLID modeling language:

- State variables of Boolean, bit-vector, or memory types.
- Expressions combine theories of uninterpreted functions, finite-precision bit-vector arithmetic, and arrays.
Theoretical Formulation: Modeling Running Example

Bochs Address Translation

\[ f_1 \]

```c
u32 page_fault;
u32 translate(u32 curr_privilege_level)
{
    if (curr_privilege_level == 3)
        page_fault = 1;
    else
        page_fault = 0;
}
```
Theoretical Formulation: Modeling Running Example

Bochs Address Translation

\( f_1 \)

\[
\begin{align*}
\text{u32 } & \text{ page_fault;} \\
u32 & \text{ translate(u32 curr_privilege_level)} \\
\{ & \text{ if (curr_privilege_level } = 3) \notag \\
& \text{ page_fault } = 1; \notag \\
& \text{ else} \notag \\
& \quad \text{ page fault } = 0; \notag \\
\}
\end{align*}
\]

Bochs Address Translation

\( op_1 \)

\[
\begin{align*}
\text{INPUT} & \notag \\
& \text{ cpl : BITVEC[32];} \notag \\
\text{VAR} & \notag \\
& \text{ page_fault : BITVEC[32];} \notag \\
\text{ASSIGN} & \notag \\
& \text{ next[page_fault] := (cpl[0] & cpl[1]);} \notag \\
\end{align*}
\]
Theoretical Formulation: Modeling Running Example

Bochs Address Translation

$f_1$

u32 page_fault;
u32 translate(u32 curr_privilege_level)
{
    if (curr_privilege_level == 3)
        page_fault = 1;
    else
        page_fault = 0;
}

Bochs Address Translation

$op_1$

INPUT
    cpl : BITVEC[32];
VAR
    page_fault : BITVEC[32];
ASSIGN
    next[page_fault] := (cpl[0] & cpl[1]);

_main schedules $op_1$ at each step. Execution of $f_{misc}$ is considered a stuttering step.
Theoretical Formulation: OMV

1. Model’s initial state is equivalent to program’s initial state.
Theoretical Formulation: OMV

1. Model’s initial state is equivalent to program’s initial state.

2. $M$ simulates $P$ using the labeling function based on $V^*$
   - Check that the transition relation $\delta^M_i$ of each $op_i$ overapproximates transition relation $\delta^P_i$ of $f_i$ by proving $\delta^P_i \Rightarrow \delta^M_i$.
   - $\delta^P_i$ is computed using symbolic execution, and $\delta^P_i \Rightarrow \delta^M_i$ is discharged using SMT solving.
Outline

1. Introduction
   - Motivation
   - Running example

2. Problem Definition
   - Work Flow
   - Theoretical Formulation
   - DMV
   - OMV

3. Methodology
   - DMV Algorithm
   - OMV Algorithm

4. Evaluation
   - DMV Case Studies
   - OMV Case Studies

5. Related Work
DMV Algorithm

- **DMV Goal**: Prove that $f_{misc}$ does not modify variables in $\mathcal{V}^*$. 
DMV Algorithm

- **DMV Goal**: Prove that $f_{\text{misc}}$ does not modify variables in $\mathcal{V}^*$.  

- **Strategy**: Keep track of the last valuation of $\mathcal{V}^*$. Prove that it does not change after any statement in $f_{\text{misc}}$. 
DMV Algorithm

- **DMV Goal**: Prove that $f_{misc}$ does not modify variables in $\mathcal{V}^*$.
- **Strategy**: Keep track of the last valuation of $\mathcal{V}^*$. Prove that it does not change after any statement in $f_{misc}$.
- **Steps**: Instrument the program
  For each variable $v$ in $\mathcal{V}^*$:
  1. Introduce shadow variable $v_s$.
  2. After any statement in $f_1, f_2, \ldots, f_N$, synchronize $v_s$ via assignment $v_s := v$.
  3. After any statement in $f_{misc}$, assert $(v == v_s)$.

Use KLEE to find a counter-example that violates an assertion.
DMV Algorithm

- **DMV Goal**: Prove that $f_{misc}$ does not modify variables in $\mathcal{V}^*$. 
- **Strategy**: Keep track of the last valuation of $\mathcal{V}^*$. Prove that it does not change after any statement in $f_{misc}$.
- **Steps**: Instrument the program
  For each variable $v$ in $\mathcal{V}^*$:
  1. Introduce shadow variable $v_s$. 
DMV Algorithm

- **DMV Goal:** Prove that $f_{misc}$ does not modify variables in $\mathcal{V}^*$.  
- **Strategy:** Keep track of the last valuation of $\mathcal{V}^*$. Prove that it does not change after any statement in $f_{misc}$.  
- **Steps:** Instrument the program  
  For each variable $v$ in $\mathcal{V}^*$:  
  1. Introduce shadow variable $v_s$.  
  2. After any statement in $f_1, f_2, \ldots, f_N$, synchronize $v_s$ via assignment $v_s := v$.  

Rohit Sinha  
Symbolic Software Model Validation  
MEMOCODE 2013
DMV Algorithm

- **DMV Goal**: Prove that $f_{\text{misc}}$ does not modify variables in $\mathcal{V}^*$.
- **Strategy**: Keep track of the last valuation of $\mathcal{V}^*$. Prove that it does not change after any statement in $f_{\text{misc}}$.
- **Steps**: Instrument the program
  For each variable $v$ in $\mathcal{V}^*$:
  1. Introduce shadow variable $v_s$.
  2. After any statement in $f_1, f_2, \ldots, f_N$, synchronize $v_s$ via assignment $v_s := v$.
  3. After any statement in $f_{\text{misc}}$, assert($v == v_s$).
DMV Algorithm

**DMV Goal:** Prove that $f_{misc}$ does not modify variables in $V^*$.

**Strategy:** Keep track of the last valuation of $V^*$. Prove that it does not change after any statement in $f_{misc}$.

**Steps:** Instrument the program
For each variable $v$ in $V^*$:
1. Introduce shadow variable $v_s$.
2. After any statement in $f_1, f_2, \ldots, f_N$, synchronize $v_s$ via assignment $v_s := v$.
3. After any statement in $f_{misc}$, assert($v == v_s$).

Use KLEE to find a counter-example that violates an assertion.
DMV Algorithm: Running Example

```c
if (curr_privilege_level == 3)
    page_fault = 1;
    page_fault_s = page_fault;
else
    page_fault = 0;
    page_fault_s = page_fault;
...
update_access_dirty(...);
assert(page_fault_s == page_fault);
```
OMV Algorithm

- **OMV Goal:** Prove that the model $M$ correctly simulates program $P$. 
OMV Algorithm

- **OMV Goal**: Prove that the model $M$ correctly simulates program $P$.
- **Strategy**: Symbolically execute $P$ to generate path condition and input-output relation of each path. Invoke SMT solver to prove that both $M$ and $P$ produce the same symbolic output on each path.
OMV Algorithm

- **OMV Goal**: Prove that the model $\mathcal{M}$ correctly simulates program $\mathcal{P}$.

- **Strategy**: Symbolically execute $\mathcal{P}$ to generate path condition and input-output relation of each path. Invoke SMT solver to prove that both $\mathcal{M}$ and $\mathcal{P}$ produce the same symbolic output on each path.

- **Steps**:
  1. Setup: Initialize global state and inputs to unconstrained, symbolic values.
OMV Algorithm

**OMV Goal**: Prove that the model $M$ correctly simulates program $P$.

**Strategy**: Symbolically execute $P$ to generate path condition and input-output relation of each path. Invoke SMT solver to prove that both $M$ and $P$ produce the same symbolic output on each path.

**Steps**:
1. **Setup**: Initialize global state and inputs to unconstrained, symbolic values.
2. **Symbolic execution**: Use KLEE to generate path condition $\pi_P$ and its corresponding input-output relation $\nu_P$. 
OMV Algorithm

- **OMV Goal**: Prove that the model $\mathcal{M}$ correctly simulates program $\mathcal{P}$.
- **Strategy**: Symbolically execute $\mathcal{P}$ to generate path condition and input-output relation of each path. Invoke SMT solver to prove that both $\mathcal{M}$ and $\mathcal{P}$ produce the same symbolic output on each path.
- **Steps**:
  1. **Setup**: Initialize global state and inputs to unconstrained, symbolic values.
  2. **Symbolic execution**: Use KLEE to generate path condition $\pi_\mathcal{P}$ and its corresponding input-output relation $\nu_\mathcal{P}'$.
  3. **SMT queries**: Use UCLID to decide $(\pi_\mathcal{P} \Rightarrow (\nu_\mathcal{M}' = \nu_\mathcal{P}'))$
OMV Algorithm: Running Example

\textbf{UCLID model}

INPUT
\begin{align*}
\text{cpl} & : \text{BITVEC}[32]; \\
\text{page\_fault} & : \text{BITVEC}[32];
\end{align*}

VAR
\begin{align*}
\text{page\_fault} & \triangleright \text{BITVEC}[32]; \\
\text{init}[\text{page\_fault}] & := \text{arbitrary\_cpl}; \\
\text{next}[\text{page\_fault}] & := (\text{cpl}[0] \& \text{cpl}[1]);
\end{align*}

\textbf{C code}

\begin{verbatim}
#include <stdio.h>

int main()
{
    u32 curr\_privilege\_level, page\_fault;
    klee\_make\_symbolic(curr\_privilege\_level);
    klee\_make\_symbolic(page\_fault);
    if (curr\_privilege\_level == 3)
        page\_fault = 1;
    else
        page\_fault = 0;
    return 0;
}
\end{verbatim}
OMV Algorithm: Running Example

\textbf{C code}

\begin{verbatim}
    u32 curr_privilege_level, page_fault;
    klee_make_symbolic(curr_privilege_level);
    klee_make_symbolic(page_fault);
    if (curr_privilege_level == 3)
        page_fault = 1;
    else
        page_fault = 0;
\end{verbatim}

\textbf{UCLID model}

\begin{verbatim}
    INPUT
    cpl : BITVEC[32];
    VAR
    page_fault : BITVEC[32];
    ASSIGN
    init[page_fault] := arbitrary_cpl;
    next[page_fault] := (cpl[0] & cpl[1]);
\end{verbatim}

\textbf{Symbolic Execution:}

\[(\pi_1, \nu_{P_1}) : (\text{curr\_privilege\_level} = 3, \text{page\_fault} = 1)\]
\[(\pi_2, \nu_{P_2}) : (\text{curr\_privilege\_level} \neq 3, \text{page\_fault} = 0).\]
OMV Algorithm: Running Example

\[ f_1 \text{ C code} \]

```c
u32 curr_privilege_level, page_fault;
klee_make_symbolic(curr_privilege_level);
klee_make_symbolic(page_fault);
if (curr_privilege_level == 3)
    page_fault = 1;
else
    page_fault = 0;
```

\[ \text{op}_1 \text{ UCLID model} \]

```
INPUT
    cpl : BITVEC[32];
VAR
    page_fault : BITVEC[32];
ASSIGN
    init[page_fault] := arbitrary_cpl;
    next[page_fault] := (cpl[0] & cpl[1]);
```

**Symbolic Execution:**

\[ (\pi_1, V_{P_1'}) : (curr\_privilege\_level = 3, page\_fault = 1) \]
\[ (\pi_2, V_{P_2'}) : (curr\_privilege\_level \neq 3, page\_fault = 0). \]

**SMT queries:** \( M \) is valid iff both queries are proved correct

1. \( (page\_fault' = ITE(cpl[0] = 1 \land cpl[1] = 1, 1, 0) \land cpl = 3) \Rightarrow (page\_fault' = 1) \)
2. \( (page\_fault' = ITE(cpl[0] = 1 \land cpl[1] = 1, 1, 0) \land cpl \neq 3) \Rightarrow (page\_fault' = 0) \)

Counter-example: \( cpl = 0x00000007 \)
Outline

1. Introduction
   - Motivation
   - Running example

2. Problem Definition
   - Work Flow
   - Theoretical Formulation
   - DMV
   - OMV

3. Methodology
   - DMV Algorithm
   - OMV Algorithm

4. Evaluation
   - DMV Case Studies
   - OMV Case Studies

5. Related Work
DMV Case Studies: XMHF Intercept Handling

```c
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 EPT_VIOLATION:
            handle_pagetable(vcpu, regs); break;
    ...
    }
    ...
}
```
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 EPT_VIOLATION:
        handle_pagetable(vcpu, regs); break;
    ...
    }
    ...
}

- $I_P = \{vcpu, regs\}$, $V_P = \{vcpu\}$, $Init_P = \text{true}$
- $\phi$: PML4 field contains a valid address.
- $F = \{f_1, f_{misc}, f_{orc}\}$ where $f_1$ includes handle_pagetable, $f_{misc}$ includes handle_crx_access and handle_ioport_access, and $f_{orc}$ is the switch-case composition.
DMV Case Studies: XMHF Intercept Handling

```c
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 EPT_VIOLATION:
            handle_pagetable(vcpu, regs); break;
        ...
    }
    ...
}
```

- $I_P = \{vcpu, regs\}$, $V_P = \{vcpu\}$, $Init_P = true$
- $\phi$: PML4 field contains a valid address.
- $\mathcal{F} = \{f_1, f_{misc}, f_{orc}\}$ where $f_1$ includes $handle\_pagetable$, $f_{misc}$ includes $handle\_crx\_access$ and $handle\_ioport\_access$, and $f_{orc}$ is the switch-case composition.

- **goal**: $f_{misc}$ does not update PML4 field.
- **runtime**: 8-10 seconds
DMV Case Studies: Other benchmarks

   - runtime: 3-5 seconds

2. GNU ftpd: File Transfer Protocol server. DMV check on parts of the authentication module.
   - runtime: 3 minutes
Bochs is an open-source x86 emulator written in C++
address translation function: emulates MMU
  - approx. 200 LoC
  - input variables: virtual address, read/write request permission, CPU current privilege level
  - state variables: page table, translation lookaside buffer

KLEE explored all 219 paths in 9 seconds, UCLID proved all 219 queries in 55 seconds.
Sample of 7 Bochs bugs

Error: first and second cases in Model are in wrong order

<table>
<thead>
<tr>
<th>Code</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>init[paddress] := ucl_32b_0;</td>
</tr>
<tr>
<td>if (present) {</td>
<td>next[paddress] := case</td>
</tr>
<tr>
<td>paddress = tlbEntry-&gt;ppf</td>
<td>poffset;</td>
</tr>
<tr>
<td>if (permission) }</td>
<td>ucl_32b_0;</td>
</tr>
<tr>
<td>return paddress; //TLB hit</td>
<td>(~permission</td>
</tr>
<tr>
<td>}</td>
<td>(ppf # [31:12])</td>
</tr>
<tr>
<td>}</td>
<td>@ poffset; (* page walk *)</td>
</tr>
<tr>
<td>...</td>
<td>default :</td>
</tr>
<tr>
<td>if (!(pde &amp; 0x1)</td>
<td></td>
</tr>
<tr>
<td>pagefault = 1;</td>
<td>@ poffset; (* TLB hit *)</td>
</tr>
<tr>
<td>return 0; }</td>
<td>esac;</td>
</tr>
<tr>
<td>if (!priv_check[priv_index]) {</td>
<td></td>
</tr>
<tr>
<td>pagefault = 1;</td>
<td></td>
</tr>
<tr>
<td>return 0; }</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>return ppf</td>
<td>poffset; //page walk</td>
</tr>
</tbody>
</table>
OMV Case Studies: Other benchmarks

1. Berkeley Packet Filter: \texttt{bpf\_validate} module for validating the filter program.
   - inputs: filter program, filter program length
   - runtime: 2 seconds, bugs found: 4
OMV Case Studies: Other benchmarks

   - inputs: filter program, filter program length
   - runtime: 2 seconds, bugs found: 4

2. TCAS: Traffic Collision Avoidance Software.
   - created 23 models, each injected with a fault developed by Hutchins et al.
   - runtime: 5 seconds per model, all bugs were found
Outline

1 Introduction
   - Motivation
   - Running example

2 Problem Definition
   - Work Flow
   - Theoretical Formulation
   - DMV
   - OMV

3 Methodology
   - DMV Algorithm
   - OMV Algorithm

4 Evaluation
   - DMV Case Studies
   - OMV Case Studies

5 Related Work
Related Work

OMV:
- Equivalence checking between C and RTL: Clarke et al., Hu et al.

DMV:
- Alias Analysis: Hind et al., Tip et al.
Summary

Contribution:

- A theoretical framework for software model validation.
- An implementation of this framework using KLEE and UCLID.
- Evaluation on software benchmarks.
Summary

Contribution:

- A theoretical framework for software model validation.
- An implementation of this framework using KLEE and UCLID.
- Evaluation on software benchmarks.

Future Work:

- Experimenting with other techniques for DMV
- Scaling symbolic execution tools