Using Symbolic Formal Verification to Identify State Continuity vulnerabilities in Enclave Programs

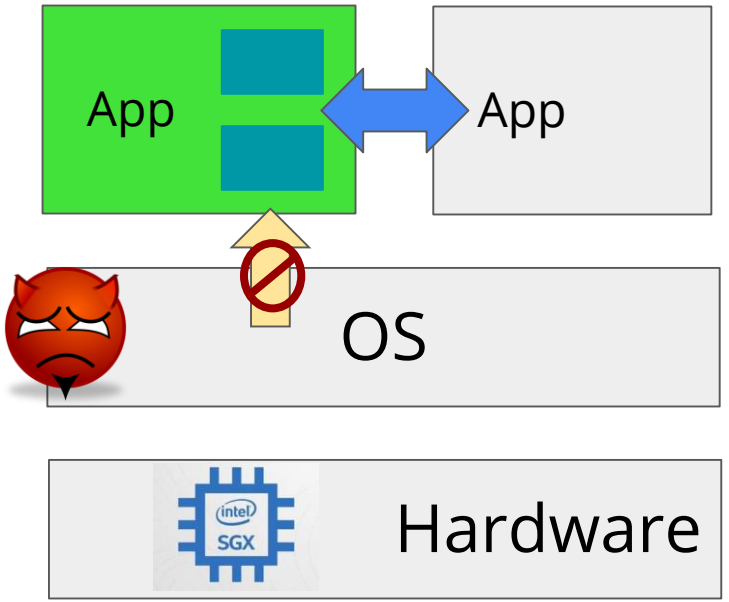
Zhiqiang Lin
zlin@cse.ohio-state.edu

Joint work with Mohit K. Jangid, Guoxing Chen, and Yinqian Zhang

Feb 3rd, 2022

What is Enclave Program

- Program executed inside hardware protected trusted execution environments (TEE) such as Intel SGX against untrusted privileged software from confidentiality and integrity attacks

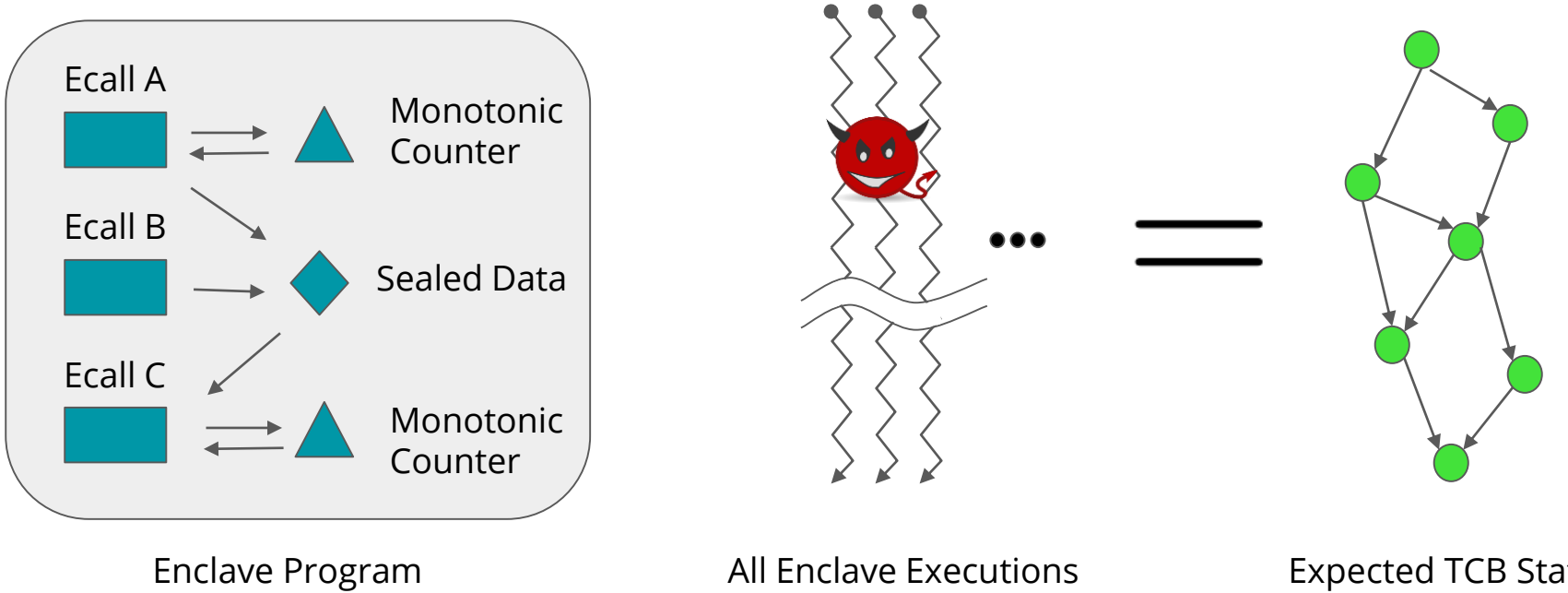


What is State Continuity

- Classic definition: ***protected module must resume from the same execution state after TCB (Trusted Computing Base) interrupts***
- **New TCB modules** in TEE:
  - Enclave memory (local/heap/global variables)
  - Non-volatile memory (monotonic counters)
  - Persistent storage (sealed data)
- **New threat model** in TEE:
  - Controls the privileged code (e.g., OS, or hypervisors)
  - Arbitrary thread and process instantiation
  - Permute, reorder enclave calls
  - Access to ecall or ocall arguments and returns
  - Replay, modify the data in untrusted code

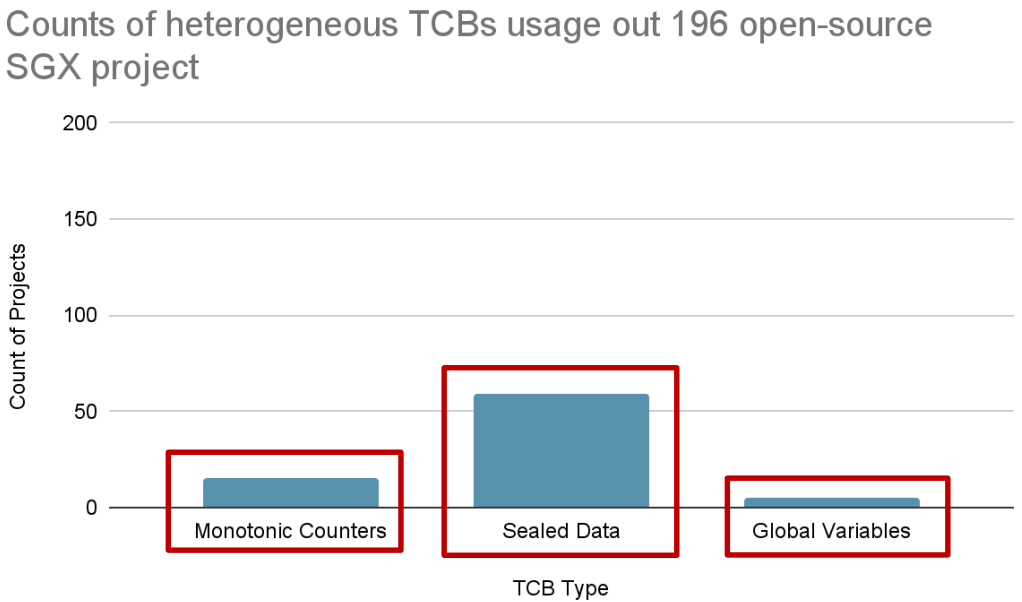
State Continuity for Enclave Programs

- Enclave program states ***always execute on the expected TCB state*** under the SGX threat model and TCB interrupts



Maintaining State Continuity is Important

State continuity TCB modules are prevalent in many open SGX applications.



The Research Question

State continuity properties are difficult to verify in the SGX environment.

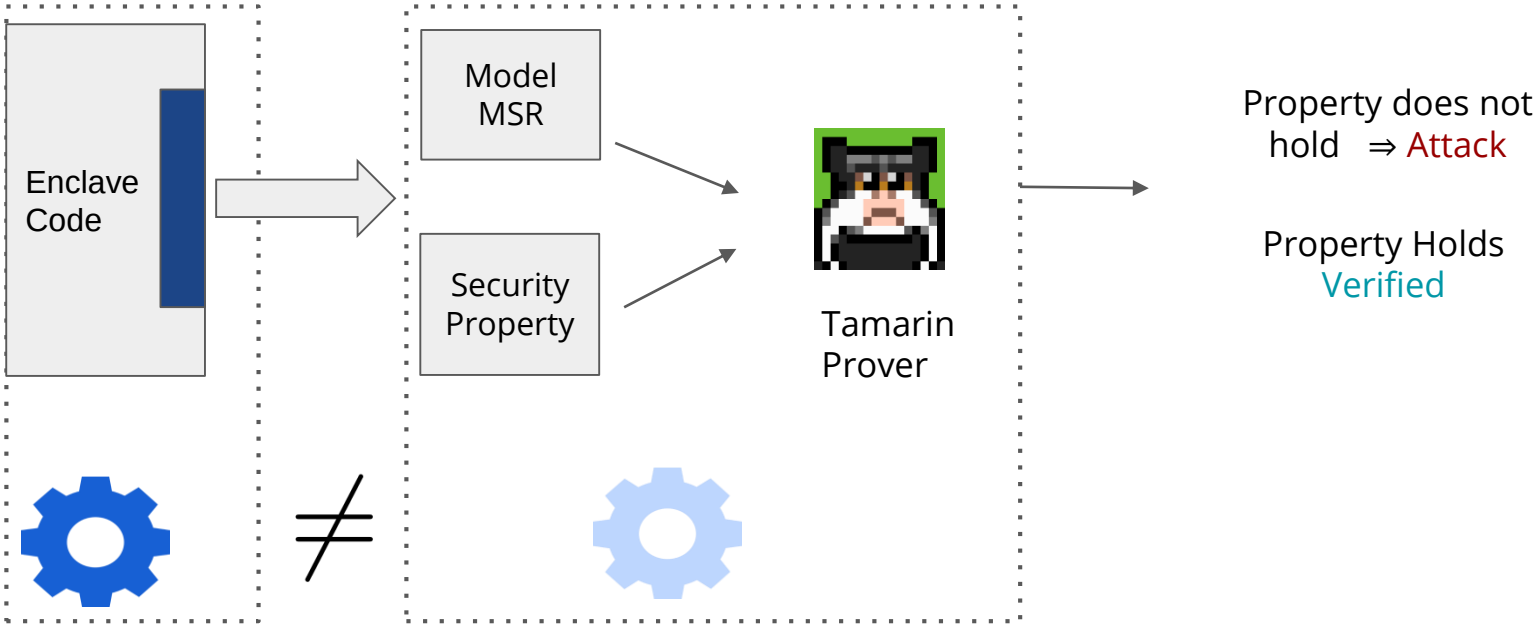
Manual efforts is tedious and error prone

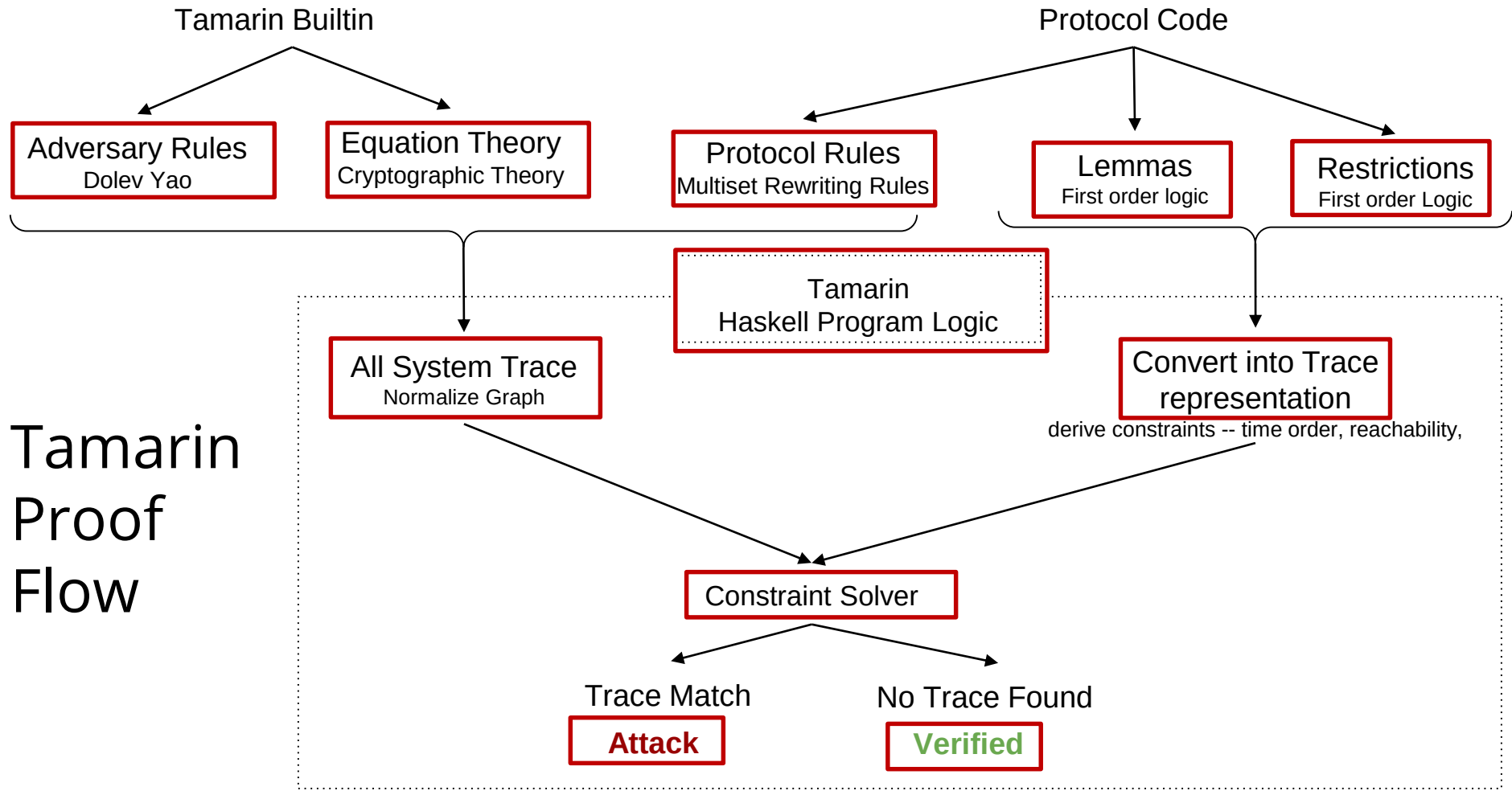
1. Clearly understand trusted & untrusted boundary
2. Correct coordination of heterogeneous TCB modules
3. Carefully apply thread synchronization and locks

Is there a systematic approach to verify state continuity ?

Potential Solution

- Use Symbolic Verification Tool -- Tamarin, to verify state continuity property

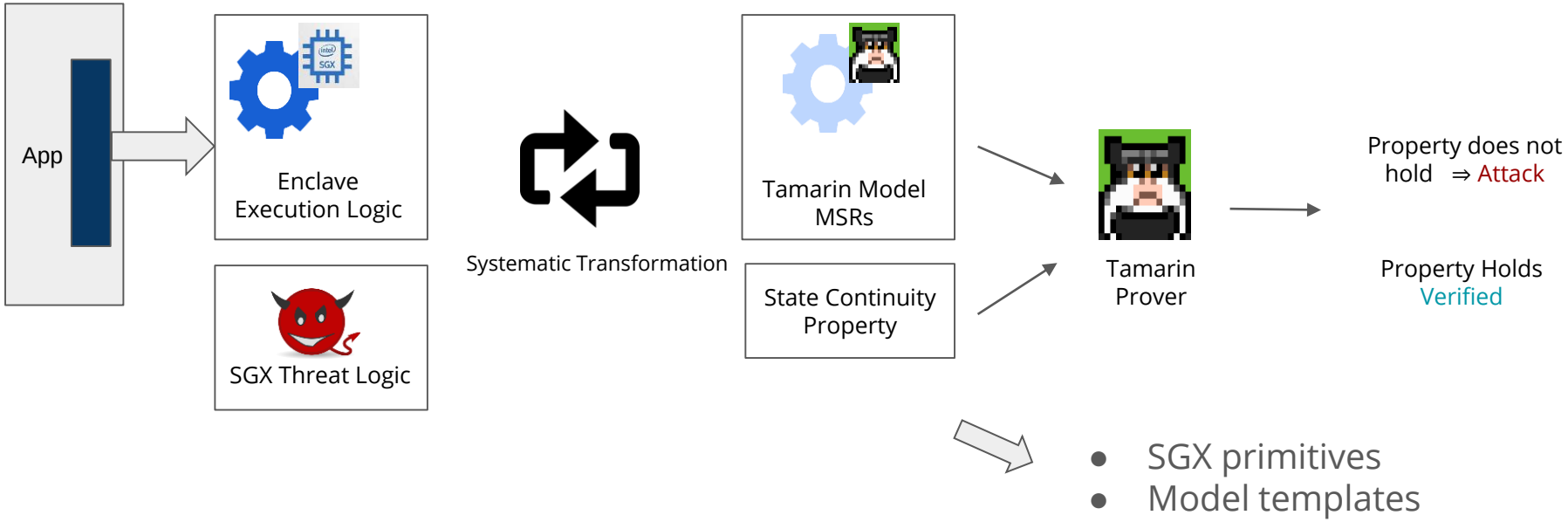



Tamarin Builtin
Protocol Code
Adversary Rules
Dolev Yao
Equation Theory
Cryptographic Theory
Protocol Rules
Multiset Rewriting Rules
Lemmas
First order logic
Restrictions
First order Logic
Tamarin
Haskell Program Logic
All System Trace
Normalize Graph
Convert into Trace representation
derive constraints -- time order, reachability,
Tamarin Proof Flow
Constraint Solver
Trace Match
Attack
No Trace Found
Verified

Outline

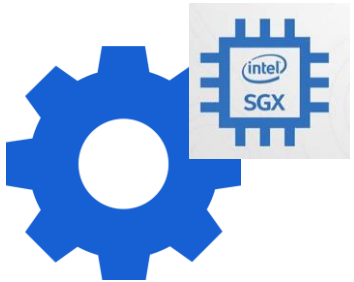
- Background
- **Our Approach**
- Example Sawtooth
- Modelling
- Conclusion

Our Approach



Challenge

Execution model of Tamarin MSRs differs significantly from enclave code execution.



Enclave Execution Model
Programming Paradigm

SGX Adversary

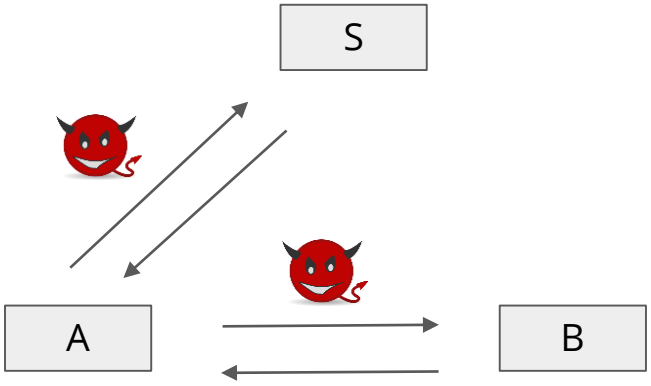
≠

Tamarin Execution Model
Multiset Rewriting Rules
(MSR)

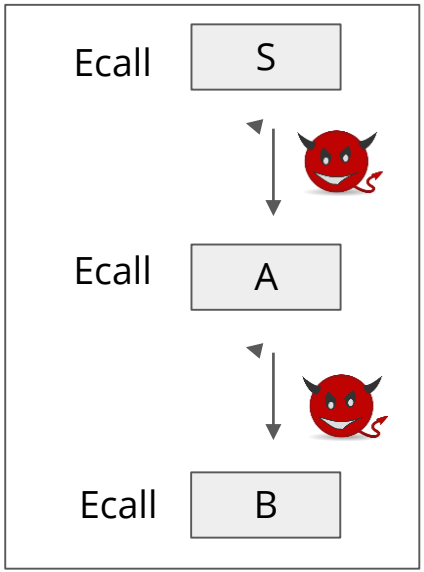
Dolev-Yao Adversary

Key Observations

Cryptographic Protocols and SGX Environment share common features



Key Exchange Protocols



SGX Environment

Tamarin MSR and query language

State Cont. Properties
SGX Thread Model

Outline

- Background
- Our Approach
- **Example Sawtooth**
- Modelling
- Conclusion

Example -- Sawtooth

- Permissioned Blockchain Framework
- Consensus algorithm: Proof-of-Elapsed-Time (PoET)
- Leverages Intel SGX for fair node participation
- Each node workflow
  - Signup and register into the blockchain network
  - Participate in the block leader election

Proof of Elapsed Time (PoET)

- Each node waits for a random time duration
- After the wait interval, each node broadcast it's **PoET certificate**

- The node with the certificate of the smallest wait duration wins the round and is allowed to publish a block in the ledger.

This algorithm is implemented using two ecalls E1 and E2

Ecall E1

- Generate random wait duration
- Create reference monotonic counter (**MC_ref**)
- Seal the duration and MC_ref

```
poet_err_t ecall_CreateWaitTimer(...)
{
        ...

        // Get the current sgx time (as a time basis)
        sgx_time_source_nonce_t timeSourceNonce;
        double sgx_request_time =
            static_cast<double>(GetCurrentTime(&timeSourceNonce));
        double duration =
            GenerateWaitTimerDuration(
                validatorAddress,
                previousCertificateId,
                inLocalMean);

        // Get the sequence ID (prevent replay) for this timer
        uint32_t sequenceId = 0;
        ret = sgx_increment_monotonic_counter(
            &validatorSignupData.counterId,
            &sequenceId);
        ...

        // Sign the serialized timer using the PoET secret key. The handle
        // will close automatically for us.
        Intel::SgxEcc256StateHandle eccStateHandle;

        ret = sgx_ecc256_open_context(&eccStateHandle);
        sp::ThrowSgxError(ret, "Failed to create ECC256 context");

        ret =
            sgx_ecdsa_sign(
                reinterpret_cast<const uint8_t *>(outSerializedTimer),
                static_cast<int32_t>(strlen(outSerializedTimer)),
                const_cast<sgx_ec256_private_t *>(
                    &validatorSignupData.privateKey),
                outTimerSignature,
                eccStateHandle);


    return result;
} // ecall_CreateWaitTimer
```

Ecall E2

- Unseal and verify the sealed object
- Verify elapsed time
- Compare **MC_ref**
- Generate PoETCertificate
- MC Increment

```
poet_err_t ecall_CreateWaitCertificate(...)
{   ...
    ret =
        sgx_ecdsa_verify(
            ...,
            inSerializedWaitTimer
            ...);
    ...

    // Verify that another wait timer has not been created after this
    // one and before the wait certificate has been requested.
    uint32_t sequenceId = 0;
    ret = sgx_read_monotonic_counter(
        &validatorSignupData.counterId,
        &sequenceId);
    if (sequenceId != waitTimer.SequenceId) {
        ...
        throw sp::ValueError("WaitTimer out of sequence.  (Attempted replay "
                "attack?)");
    }

    // PoET certificate generation
    uint8_t nonce[WAIT_CERTIFICATE_NONCE_LENGTH];
    ret = sgx_read_rand(nonce, sizeof(nonce));
    sp::ThrowSgxError(ret, "Failed to generate wait certificate nonce");
    std::string nonceHexString =
        sp::BinaryToHexString(nonce, sizeof(nonce));

    ...

    // Increment the counter to prevent creating another
    // wait certificate from the same timer.
    ret = sgx_increment_monotonic_counter(
        &validatorSignupData.counterId,
        &sequenceId);
    ...

    return result;
} // ecall_CreateWaitCertificate
```

Continuity of Certificate Generation State

- **Only one certificate should be generated per election round by a node**
- The **continuity** of the certificate generation state is preserved by using the monotonic counter

Sawtooth Block Leader Election

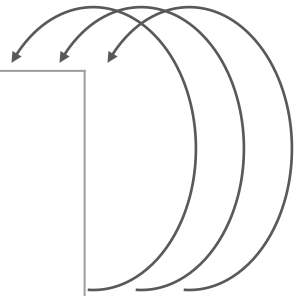
Ecall E1

Election 1 Create Reference Objects	• Generate random wait duration • Create reference monotonic counter (**MC_ref**) • Seal the duration and MC_ref

Wait random duration

Ecall E2

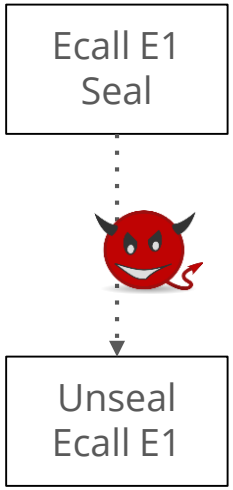
Election 2 Verify Proof of Elapsed Time	• Unseal and verify the sealed object • Verify elapsed time • Compare **MC_ref** **X X X** • PoETCertificate **Monotonic Counter ++**

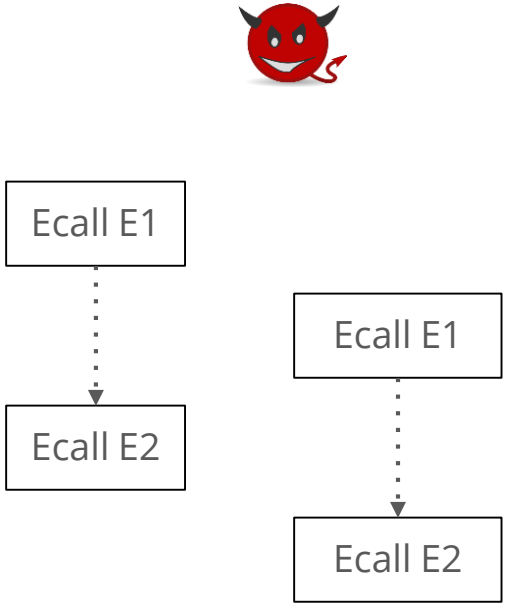


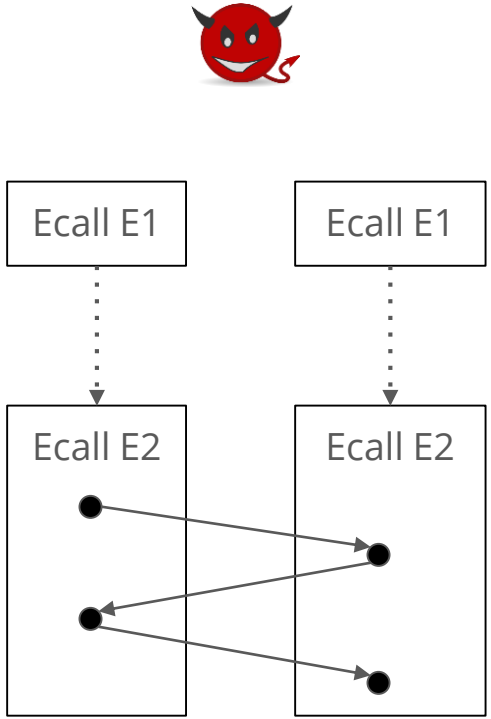
Sawtooth Expected TCB States

1. Monotonic Counter Value < MC_Ref — **PoETCertificate** 
2. Monotonic Counter Value = MC_Ref — **PoETCertificate** 
3. Monotonic Counter Value > MC_Ref — **Abort**

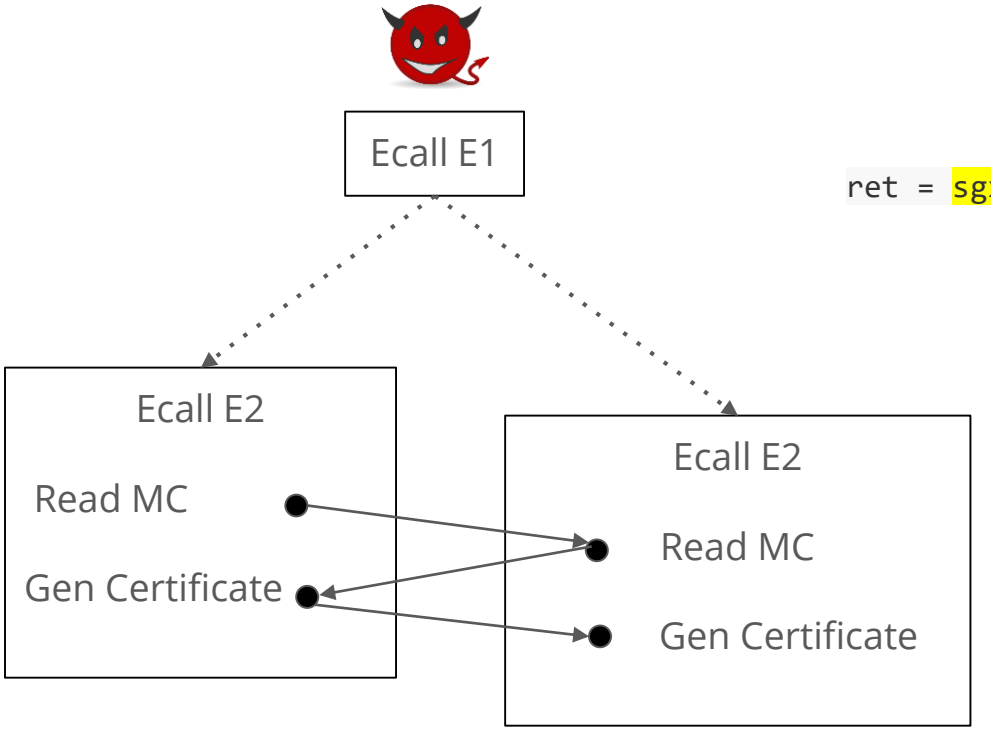
What Could Go Wrong?







What Goes Wrong?




```
ret = sgx_read_monotonic_counter()
```

Outline

- Background
- Our Approach
- Example Sawtooth
- **Modelling**
- Conclusion

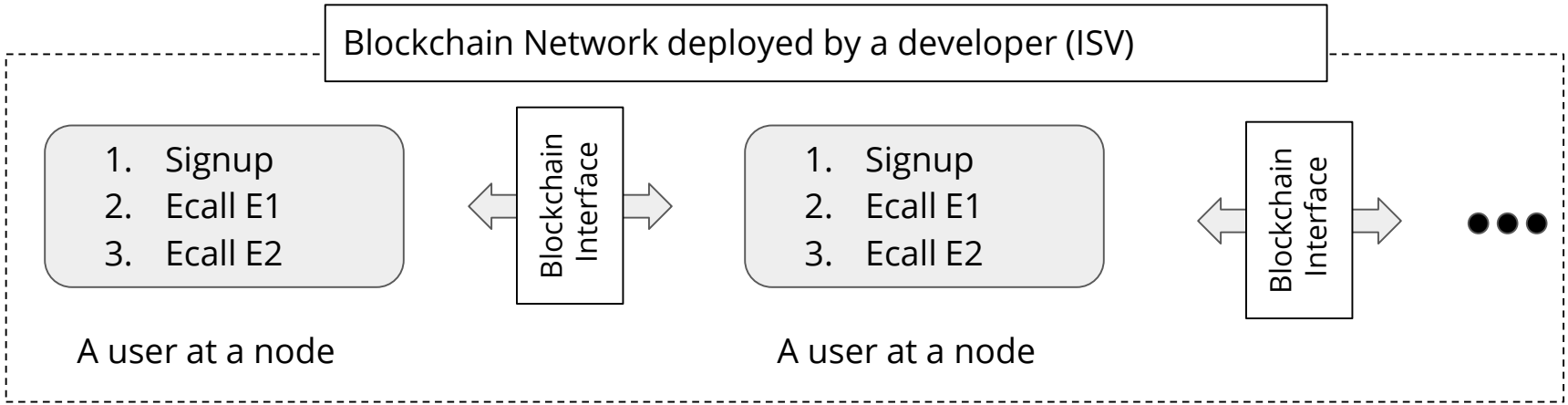
Tamarin Model for Sawtooth

- What components of the workflow do we need?
  - SGX entities -- independent software vendor (ISV), User, Nodes, Processes
  - Entity association network
  - Enclave threads
  - Sealed sign-up information
  - Monotonic Counter

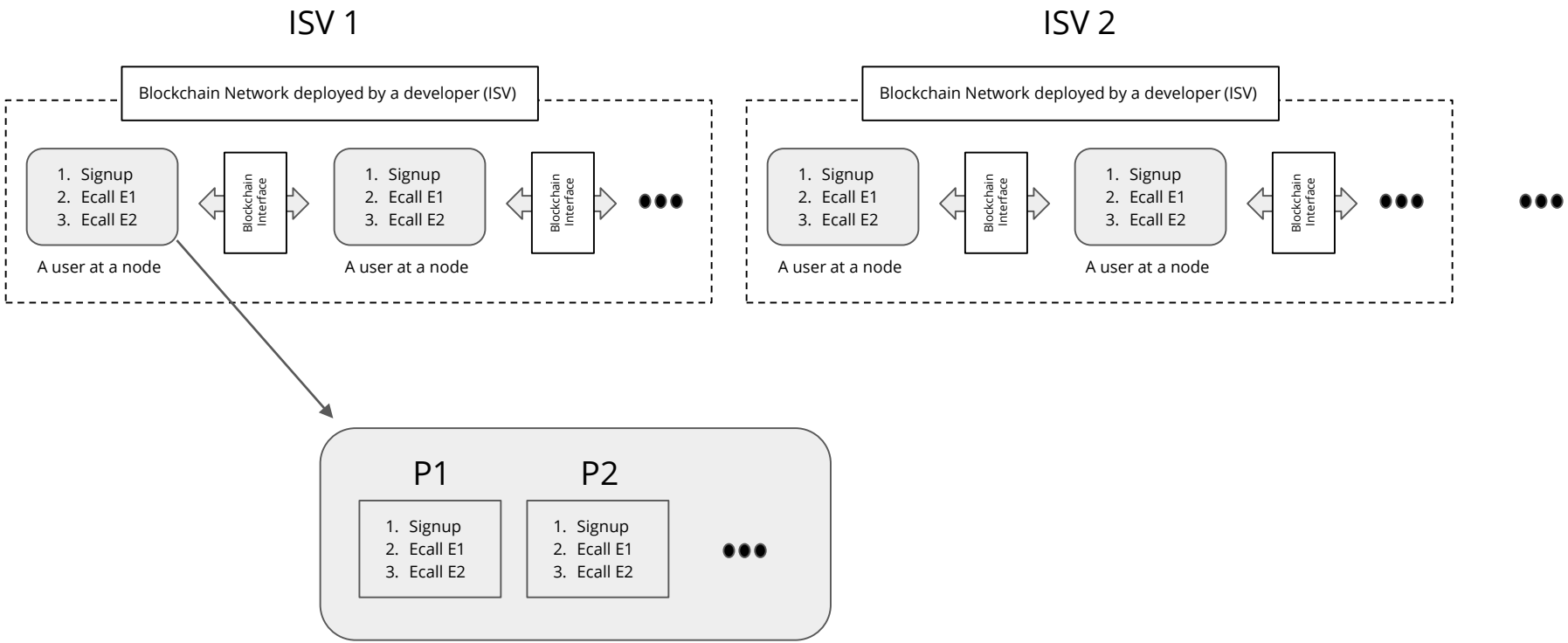
Tamarin Model for Sawtooth

- What components of the workflow do we need?
  - **SGX entities -- independent software vendor (ISV), User, Nodes, Processes**
  - Entity association network
  - Enclave threads
  - Sealed sign-up information
  - Monotonic Counter

SGX Entities



SGX Entities

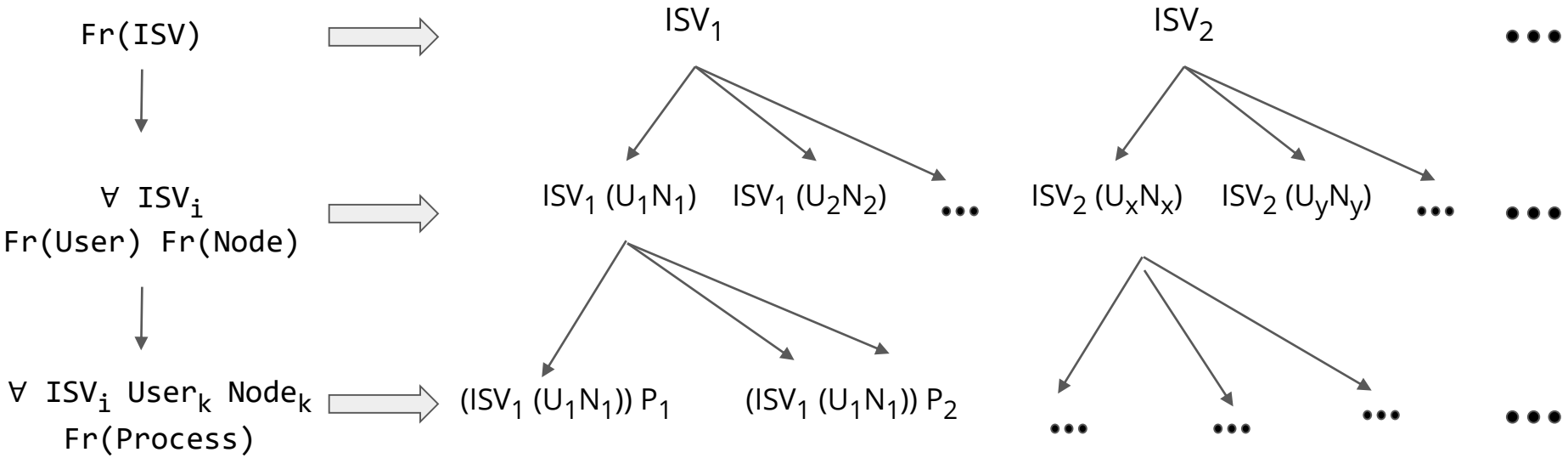


Tamarin Model for Sawtooth

- What components of the workflow do we need?
  - SGX entities -- ISV, User, Nodes, Processes
  - **Entity association network**
  - Enclave threads
  - Sealed sign-up information
  - Monotonic Counter

Entity Association Network

- Tamarin Fr(*) ***Fact*** produces unique variables
- Tamarin ***Rules*** can in instantiated unbounded times
- Variables can be passed on through ***Rules*** using Tamarin ***Facts***

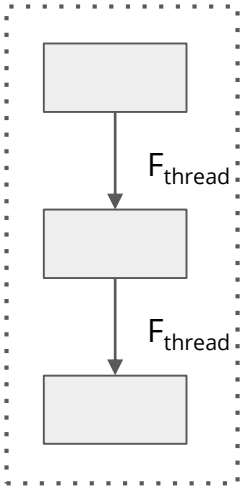


Tamarin Model for Sawtooth

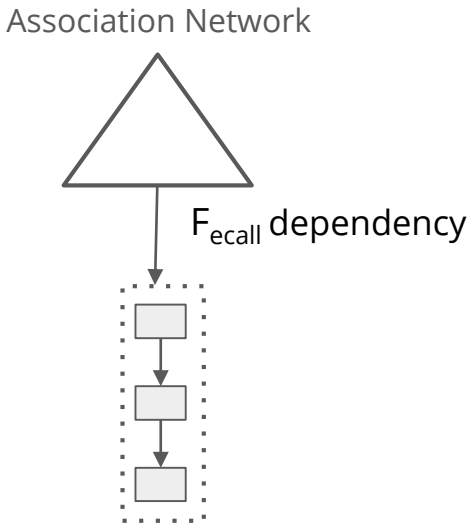
- What components of the workflow do we need?
  - SGX entities -- ISV, User, Nodes, Processes
  - Entity association network
  - **Enclave threads**
  - Sealed sign-up information
  - Monotonic Counter

Enclave Thread Construction

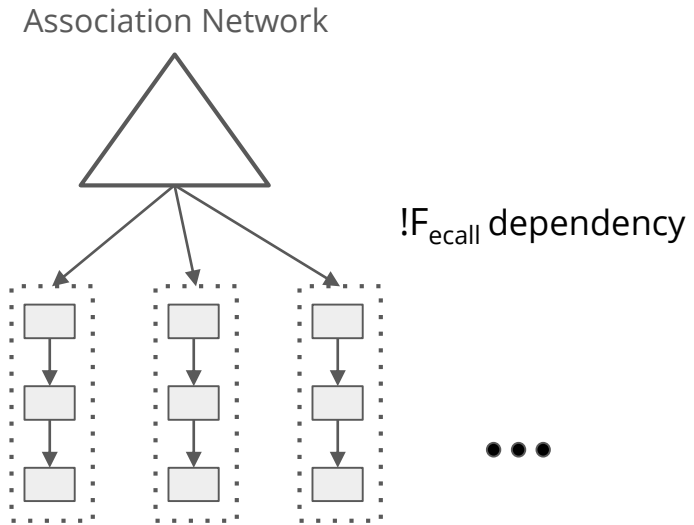
- ***Linear Fact (F)*** can be consumed only once.
- ***Persistent Fact (!F)*** can be consumed unbounded times.
- Linear and persistent **Fact dependencies** allows configuration of single and multiple thread



Enclave Thread



Single Thread



Multiple Threads

Tamarin Model for Sawtooth

- What components of the workflow do we need?
  - SGX entities -- ISV, User, Nodes, Processes
  - Entity association network
  - Enclave threads
  - Sealed sign-up information
  - Monotonic Counter

State Continuity Property

Fair election participation of each node in the blockchain requires that

a node must not generate two certificates with same MC_ref

First Order Logic Query

All

PoETCertificate (node , MC_ref) @t1 & **PoETCertificate** (node , MC_ref) @t2

== > # t1 = # t2

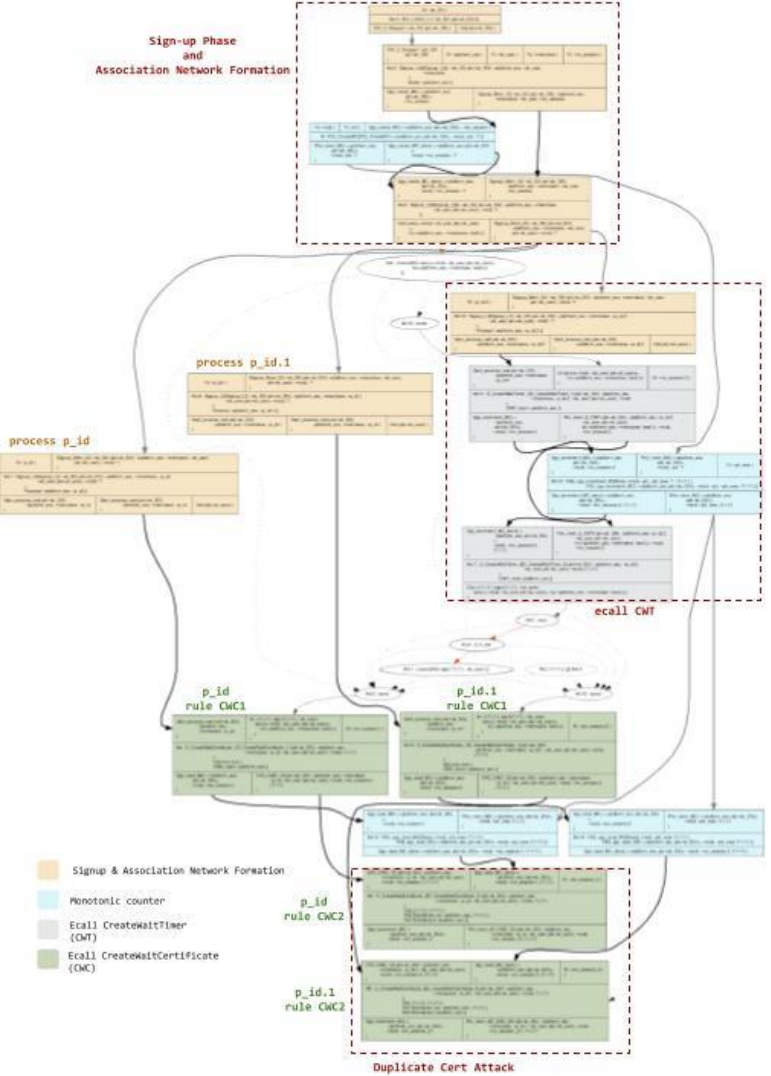
Verification

Apply patch in the mo

Sawtooth Model
Multiset Rewriting Rules

State Cont. Property
First Order Logic

Tamarin Prover



Other Case Studies

App	Attack Discovery Time	Verification Time	# Rules	Model LOC
Sawtooth	1m 18s	25s	11	300
Heartbeat	7s	2h 4m 7s	11	250
BI-SGX	36s	37s	18	450

Model Primitives used in our work

SGX primitives	Programming primitives
1. Enclave threads 2. Association network of SGX entities 3. Monotonic counters 4. Local/Global variables 5. SGX threat model 6. Key derivation 7. Sealing	1. Locks 2. Loops 3. Branching 4. Database (Read only)

SGX Threat Model

SGX Threat Model Construction	**Realized by**
Thread and process instantiation	Using the thread policy based on the ecall facts F_{ecall} in the first enclave thread rule and binding ecall sequences of rules using thread facts F_{thread}
Permute or reorder ecalls	Modeling the first enclave thread rule open to executability without order dependencies of timepoints and facts
Pause enclave execution at instruction level	Modeling instructions in individual rules and utilizing atomic rule executability
Read access to ecall returns; Read/Modify access to ecall or ocall arguments and returns	Arguments and returns pass through public channel
Replay, modify of sealing, ecall or arguments and returns	Public channel use in combination Tamarin's inbuilt *Dolev Yao adversary capabilities*

Limitations

- Refine Model
- Partial Deconstructions
- Helper Lemmas
- ...

Manual Encoding

Sawtooth model MSR

Security Property in First Order Logic

Tamarin Prover

Non Termination

Outline

- Background
- Our Approach
- Example Sawtooth
- Modelling
- **Conclusion**

Conclusion

- First attempt towards using symbolic verification tools to verify the state continuity for SGX enclave programs.
- We demonstrate our approach using open-source SGX applications, resulting into reusable SGX primitives and model templates.
- Tamarin Prover can effectively model SGX-specific semantics and operations; and state continuity properties.
- Our Tamarin code is released at Github: https://github.com/OSUSecLab/SGX-Enclave-Formal-Verification, The paper was published at USENIX Security 2021 and available at https://www.usenix.org/system/files/sec21-jangid.pdf

Using Symbolic Formal Verification to Identify State Continuity vulnerabilities in Enclave Programs

Zhiqiang Lin
zlin@cse.ohio-state.edu

Joint work with Mohit K. Jangid, Guoxing Chen, and Yinqian Zhang

Feb 3rd, 2022