Learning to Produce Direct Tests for Security Verification Using Constrained Process Discovery

Kuo-Kai Hsieh[1], Li-C. Wang[1], Wen Chen[2], and Jayanta Bhadra[2]
[1]University of California, Santa Barbara
[2]NXP Semiconductors, Inc.
kuokai@umail.ucsb.edu, licwang@ece.ucsb.edu, {wen.chen, jayanta.bhadra}@nxp.com

ABSTRACT

Security verification relies on using direct tests manually prepared. Test preparation often requires intensive efforts from experts with in-depth domain knowledge. This work presents an approach to learn from direct tests written by an expert. After the learning, the learned model acts as a surrogate for the expert to produce new tests. The learning software comprises a database for accumulating and sharing security verification knowledge. The learning approach uses process discovery to build an upper-bound model and continuously adds constraints to refine it. We demonstrate the feasibility and effectiveness of the learning approach in a commercial SoC verification environment.

Keywords

Secure Hardware; Functional Verification; Grammatical Inference; Process Discovery

1. INTRODUCTION

Ubiquitous computing devices now contain an increasing portion of credential and private information referred to as security *assets* on chip. The emergence of Internet-of-Things (IoT) requires computing devices to be securely connected to each other, which imposes challenging requirements on protection of the security assets. Among the most commonly seen requirements are: (1) confidentiality, e.g., to protect manufacturing and OEM secrets (keys and codes) from unauthorized access, (2) integrity, e.g., to ensure that memory location can only be modified by an authorized agent, and (3) availability, e.g. to ensure that an asset is available to an authorized agent if the agent requires it. Authorization can be achieved through authentication which verifies the identity of an agent before enabling any access.

In System-on-Chip (SoC) design, security requirements are accommodated by defining and enforcing a set of security policies. Security policies specify the authentication, access, and protection requirements for various assets under different security execution levels during the chip life cycle [13]. One example could be the protection of a memory region during runtime where the protection policy is managed by the on-chip system controller.

The security polices are intrinsically complex and thus correct implementation of an SoC with compliance to them is challenging. A complication to the secure SoC design arises from the need to support a variety of different resource domains with different privileges. Moreover, the security requirements are often at conflict with design constraints regarding performance and power, where inappropriate tradeoff decisions can introduce security vulnerabilities.

Verifying that a hardware design is secure and trustworthy, especially at the system level, has become a major challenge. Conceptually, security verification is fundamentally different from traditional verification. Traditional verification is in view of the potential use of the hardware by a customer who has every intention to make their product work. Security verification, on the other hand, is to verify against malicious intents such as attacks.

In practice, security verification relies on two main approaches: formal verification and testbench simulation. Formal methods such as information flow analysis have been shown effective in finding vulnerabilities in the specification or an abstract model of the system, and secure design blocks. However, they run into scalability issues when being applied to system-level verification of the RTL implementation [15]. In practice, security verification is mostly done with simulation of direct tests.

Common direct tests are written by engineers targeting properties and policies as they are specified in the design document. Such verification is largely incomplete because security verification should also include penetration tests to explore the unspecified design space. However, in-depth domain knowledge of the design and the security features is often required to develop the penetration tests that can successfully expose system vulnerability.

This work is therefore motivated by the observation that developing successful penetration tests is challenging in practice, and this challenge can be viewed in two aspects. The first aspect is the lack of domain knowledge to write a penetration test. For example, this lack of domain knowledge can be for a security expert not familiar with the design or a verification engineer not familiar with the security features. The second aspect is the lack of enough personnel with the required domain knowledge. If only very few people know how to write penetration tests, it is difficult to scale the practice to obtain a large collection of such tests.

Our approach to overcome the challenge is to construct

DAC '17, June 18-22, 2017, Austin, TX, USA
 ISBN 978-1-4503-4927-7/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3061639.3062271

a learning software to learn from the test examples written by a person. After the learning software is trained to write the particular type of tests as represented by the examples, the learning software can serve as a surrogate for the person to produce many more new tests whose characteristics are similar to the test examples. The advantage of this approach is obvious. With the machine learning, one can effectively increases the number of experts that know how to write the type of penetration tests and be able to obtain a large set of such tests in a short period of time.

The field of machine learning comprises numerous and diverse approaches. The first question is therefore to decide which approach to follow. Then, the actual implementation of a particular approach can also be application context dependent. The machine learning approach presented in this work is rooted in *grammatical inference* [5]. However, this work presents a novel implementation that combines *process discovery* [16] and constraint solving to achieve the learning.

The rest of the paper is outlined as follows. Section 2 first highlights the novelty of this work in view of other related works. Then, section 3 explains the basic concepts in grammatical inference and how it fits the learning problem considered in this work. Section 4 explains process discovery and our constrained process discovery implementation. Section 5 presents the experiment results based on a commercial SoC. Section 6 concludes the paper.

2. NOVELTY OF THE APPROACH

Applying machine learning to security is a popular research field [6] [12]. However, the focus is mostly on intrusion detection based on anomaly detection [2]. For example, one can learn a classifier to detect malicious executables [10]. Our approach is fundamentally different. Instead of detecting an attack, our goal is learning to generate attacks.

Formal methods are generally not scalable to system-level SoC security verification. For instance, [14] uses formal methods to construct an attack graph for all possible attacking scenarios. [15] describes case studies where three SoC security properties are proved using formal methods. However, these works require the user to abstract the problem or to point out critical signals or boundaries. These extra requirements limit their applicability in practice.

Graph-based test generation models are commonly available in the industry today. Commercial tools include Cadence Perspec and Breker Trek. These methods require the engineers to manually construct the scenario graph, including mapping each node in the graph to some test code, and their focus is to efficiently generate tests via graph traversal algorithms. It is costly to build graph models manually. In contrast, our approach is based on direct test examples and obtains a test generation model automatically.

3. GRAMMATICAL INFERENCE

Without loss of generality, assume each direct test is a C program. To learn from a set of C programs, we need a way to represent the programs. This representation is the basis for learning and decides the *learnability* of the learning problem we formulate. In this work, each C program is represented by a set of *primitives*. One can think of a primitive as a parametrized script that, when it is called, produces a piece of C code. These primitives serve as a TPI (Test Programming Interface) for a person to write direct tests.

With primitives defined, a direct test can be represented symbolically. For example, a test can be represented as a sequence of *steps*, e.g. [A,B,C,...]. After primitive encoding, each test then can be viewed as a "sentence" example derived from an unknown formal language [8]. In other words, primitives are *words* of the unknown language. Then, *grammatical inference* [5] can be applied to discover an automata model (such as a finite automaton) to describe this language based on a given set of examples.

The *learnability* problem in grammatical inference asks whether a model can be learned with a finite number of samples. Define in-model samples and out-model samples as the samples complying with and not complying with the model to be learned, respectively. The main result of [7] points out that if only in-model samples are available, the only learnable class is the set of finite-length languages, i.e. there is a bound on the maximum length of a sentence. If both in-model and out-model samples are available, then all classes up to the Context-Sensitive grammar in the Chomsky Hierarchy can be learned [8]. In this work, we consider the case that only in-model samples are available, i.e. all the available direct tests comply with the hidden model.

3.1 Process Discovery

Since the work in [7], the learnability of a finite automaton is among those that received the most attention [1]. More recently, *process discovery* emerged as a separate field targeting business applications. Process discovery is applied to learn a process model from an event log recording instances of business transactions [17]. Each instance is represented as a sequence of transaction steps, similar to our representation of a test as a sequence of primitive steps. In process discovery, a common representation for the process model is Petri Net [17] where the graph model allows loops and concurrency. Hence, learning such a process model is as hard as learning a finite automaton.

Process discovery and the proposed approach have fundamentally different objectives. Process discovery is for discovering business intelligence from event logs. Hence, it is important for the learning model to be interpretable. Simplicity of the model to enable visualization is a key consideration. Our goal for the model is to enable test generation. Therefore, it is not necessary for our learning to produce a model summarizing all the learned information into a single interpretable model. This difference enables us to develop a novel learning approach described in the next section.

4. LEARNING FROM TEST EXAMPLES

To illustrate the basic idea of learning in process discovery, consider the following simple example. Suppose A-G represent the primitives. Suppose we have three tests: [A,B,C,D,H], [B,C,E,F,D,H], and [A,B,C,E,G,D,H]. Fig. 1 shows a process model learned from these three tests.

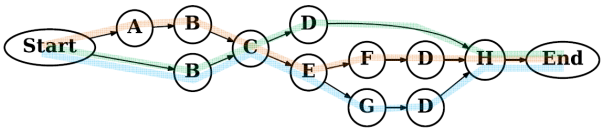

Figure 1: Process model using 1-prefix rule

This model is built based on a so-called *prefix rule* [16]. A prefix rule decides whether two steps with the same name should be represented with the same node in the process

model. Suppose one test contains a segment αX and another test contains a segment βX, where α and β each is a sequence of one or more steps and X is a step. Given a length requirement $l \geq 0$, let α_l be the last l steps in α and β_l be the last l steps in β. An *l-prefix rule* means that the two X steps would be represented by one node in the process model if $\alpha_l = \beta_l$.

Fig. 1 is based on the 1-prefix rule. For example, there is one C node in the model, representing the three C steps in the three tests. This is because every time C is involved, the step before is always B. Hence, the 1-prefix rule infers that there is only one way to use the C primitive, resulting in one C node in the model.

A process discovery algorithm essentially decides if two or more steps should be represented as a single node [17]. Observe that merged nodes can also cause new instances to be included. Including new instances not shown in the training set is called *generalization* in machine learning. In Fig. 1, three new tests are highlighted. For example, because of the merged node C, the two segments [A,B] and [E,F,D,H] can be combined to produce the new test [A,B,C,E,F,D,H].

Consider now a new test [S_1, S_2, B,C,D,H, S_3, S_4] is provided for learning. Fig. 2 shows the resulting model by adding this new test (with 1-prefix rule). S_1 to S_4 are new primitives. It is interesting to observe that the resulting model contains two new tests (as highlighted) involving the new S's primitives.

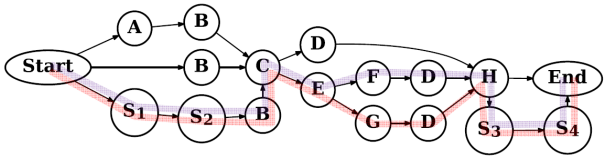

Figure 2: Process model by adding one more test

If we consider Fig. 1 as the verification knowledge learned from direct tests and the new test as a penetration test example provided by an expert, Fig. 2 illustrates how process learning can generalize from one penetration test to more penetration tests. Of course, if more penetration tests are provided, generalization can also take place among them.

Note that a process model generated based on a prefix rule can be viewed as a deterministic finite automaton (DFA). If concurrency is allowed in the tests (i.e. two segments are executed concurrently), then the model can be treated as a nondeterministic finite automaton (NFA). However, since each NFA can be converted into a DFA, for the rest of the discussion, we consider a process model as a DFA.

4.1 Constrained process discovery

Instead of learning a single process model as that in process discovery, our approach splits the learning into two parts: (1) learning an *upper-bound model*, and (2) learning a set of constraints. Fig. 3 depicts this approach.

The goal of an upper-bound model is to capture a bound on the space of all possible tests such that a desired test is ensured to be in the space. However, because it is an upper bound, the model can include many undesirable tests. A separate *constraints database* is maintained to impose constraints between and among primitives. Constraints can be learned independently of the process learning. Then, for test generation, the upper-bound model and the constraints from the database are combined for constraint solving. Each solution represents a test. In this work, we use a Boolean satisfiability (SAT) solver for constraint solving.

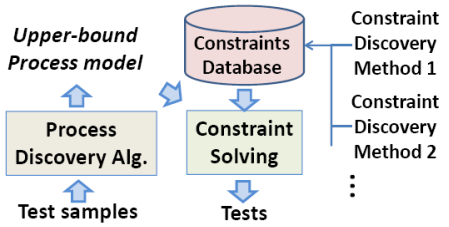

Figure 3: Overview of the proposed method

The constraints database provides a natural place to accumulate and share verification knowledge. This knowledge can be added manually, based on previous learning sessions or based on a separate constraint discovery method. It is intuitive to observe that as more constraints are included, the test space becomes smaller, enabling more focus tests to be generated. Unlike process discovery where a process model (e.g. a Petri Net model) stores all the learned information, in Fig. 3 the information is split into two parts and there is no single structure to represent all learned information.

In process discovery, one major concern is to control the *underfitting* and *overfitting* of a model [16]. In our context, underfitting means the model contains undesirable tests and overfitting means the model excludes some desirable tests. If a prefix rule is used, the key concern becomes choosing an l for the best tradeoff between underfitting and overfitting. However, because of the inflexibility of such a learning algorithm, the resulting model usually has both issues, containing undesirable tests and missing some desirable tests.

Our approach starts with an underfitting model (the upper-bound model) which is gradually refined with constraints. Suppose every constraint added to the database is valid (e.g. validated by a person before adding it to the database). Then, the approach ensures no desirable tests would be missed. As more constraints are added, the resulting model (implicitly existing) becomes closer to the desirable model. It is important to recognize that the gap between this resulting model and the desirable model is reflected in the loss of efficiency, i.e. Fig. 3 would produce undesirable tests that are not perceived as useful by the user.

4.2 The upper-bound model

Observe that for using an l-prefix rule, a larger l imposes a more stringent requirement for merging multiple steps into a single node. Hence, a larger l also leads to a less generalized model. Therefore, the upper-bound model based on a prefix rule is the 0-prefix rule model. Algorithm 1 depicts the detail of generating the upper-bound model using the 0-prefix rule.

4.3 Constraint examples

A constraint describes a dependency relationship among multiple primitives. Table 1 illustrates four types of constraints to describe a relationship, which are a subset of temporal logic where $\mathcal{B}$ denotes *before* and $\mathcal{F}$ denotes *future*.

The constraints can be used with a negation "¬" to describe a relationship. Fig. 4 shows four example constraints between two primitives. Another useful example to forbid primitive X to be used twice, i.e. preventing loop back to X, is the constraint (X $\rightarrow \neg\mathcal{B}$ X).

Constraints involving more than two primitives can be

Algorithm 1: Learning an upper-bound model

Input: a set of direct tests T
Output: a process model M
$M \leftarrow$ empty, Add states *start* and *end* to M;
foreach t *in* T **do**
  $q_0 \leftarrow start$;
  foreach q *in* t **do**
    if *state* q *not in* M **then**
      Add state q to M;
    end
    if *arc* (q_0, q) *not in* M **then**
      Add arc (q_0, q) to M;
    end
    $q_0 \leftarrow q$;
  end
  if *arc* (q_0, end) *not in* M **then**
    Add arc (q_0, end) to M;
  end
end

Table 1: Four constraints to describe a relationship

X $\rightarrow \mathcal{B}$ Y	If X is executed, Y is executed before X.
X $\rightarrow \mathcal{F}$ Y	If X is executed, Y is executed after X.
X $\rightarrow \mathcal{B}_k$ Y	If X is executed, Y is executed within k steps before X.
X $\rightarrow \mathcal{F}_k$ Y	If X is executed, Y is executed within k steps after X.

added as well, for example manually or by a constraint discovery method such as frequent episode mining [3] which might discover that a segment [A,B,C] occurs frequently. As a result, the constraint that A,B,C should be used together and in the particular sequence can become a recommended constraint for a user to include or exclude.

4.4 Test Generation - SAT Encoding

To use a SAT solver for test generation, we need the three sets of clauses: (1) those to encode the 0-prefix process model, (2) those to encode the cross-primitive constraints, and (3) those to ensure generation of new tests.

4.4.1 Encoding the process model

Inspired by [4], we use an approach that is similar to the proof of NP-completeness, but instead of encoding a Turing machine, we encode a DFA (i.e. a process model).

Let N be the number of states in the DFA and L be the maximum length of the generated tests. The proposition symbols are

- Q_t^i, for $1 \leq$ i $\leq$ N, for $1 \leq$ t $\leq$ L.

Q_t^i is True if and only if the DFA is at state q_i at step t.

The encoding of transition relation is composed of two components. First, for each state q_i, the next state of the machine can only be the states that are directly connected from q_i. Thus, transition relation at step t is encoded as $(Q_t^i \rightarrow \vee_j Q_{t+1}^j)$, where j belongs to the set of state indices of all the child states of q_i. The overall encoding for all the steps and all the states is

$$\Pi_t \Pi_i (\neg Q_t^i \vee (\vee_j Q_{t+1}^j)). \quad (1)$$

The number of clauses of this component is O(LN).

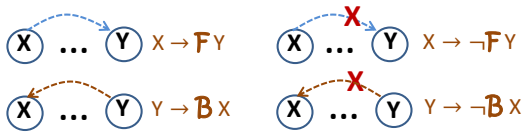

Figure 4: Example constraints between 2 primitives

Second, the machine cannot stay in more than one state at a time, i.e. *Onehot0* constraint. A naive encoding for this property is $\Pi_t \Pi_{i,j} (\neg Q_t^i \vee \neg Q_t^j)$ for all the pairs of the states and all the steps, however, the naive encoding requires $O(LN^2)$ clauses that potentially leads to a complexity problem in reality. To deal with the complexity problem, we use an approach described in [9], where the number of clauses reduces to O(N) at the expense of extra O(N) symbols. The extra proposition symbols are

- H_t^i, for $1 \leq$ i $\leq$ N, for $1 \leq$ t $\leq$ L.

H_t^i is True if and only if one of Q_t^j is True for $j \leq i$. Equivalently, H_t^i is True if and only if H_t^{i-1} is True or Q_t^i is True. Note that H_t^0 is set to False. Then the *Onehot0* property can be encoded as at most one of H_t^i and Q_t^{i+1} is True. The overall encoding is

$$\Pi_t \Pi_i (\neg H_t^i \vee H_t^{i-1} \vee Q_t^i)(H_t^i \vee \neg H_t^{i-1})(H_t^i \vee \neg Q_t^i) \quad (2)$$

and

$$\Pi_t \Pi_i (\neg H_t^i \vee \neg Q_t^{i+1}). \quad (3)$$

Overall, the number of clauses of this component is O(LN) and the number of extra symbols is O(LN).

The encoding for the start state and the end state will be discussed in Sec. 4.4.3.

4.4.2 Encoding the constraints database

There are four types of constraints in the constraints database, $\rightarrow \mathcal{B}$, $\rightarrow \mathcal{F}$, $\rightarrow \mathcal{B}_k$ and $\rightarrow \mathcal{F}_k$. The following describes their encoding separately. The similar encoding approach can be applied to encoding the constraints with negation. We omit this part due to space limitation.

To encode $q_i \rightarrow \mathcal{B} q_j$, a naive method is $\Pi_t (Q_t^i \rightarrow \vee_{s<t} Q_s^j)$. This encoding ensures that if the machine is at q_i at step t, there exist $s < t$ such that the machine is at q_j at step s. However, the naive encoding may lead to a complexity problem because the number of literals required for each constraint is $O(L^2)$. We propose another encoding method to reduce the number of literals to O(L) at the expense of extra O(L) symbols.

We introduce new proposition symbols

- B_t^j, for $1 \leq$ j $\leq$ N, for $1 \leq$ t $\leq$ L.

B_t^j is True if and only if the machine is at state q_j at some steps $<$ t. The property of the new symbols are maintained by the following relations: (1) B_1^j is False. (2) B_t^j is True if and only if B_{t-1}^j is True or Q_{t-1}^j is True. The first relation is encoded as $(\neg B_1^j)$. The second relation is encoded as

$$\Pi_t (\neg B_t^j \vee B_{t-1}^j \vee Q_{t-1}^j)(B_t^j \vee \neg B_{t-1}^j)(B_t^j \vee \neg Q_{t-1}^j). \quad (4)$$

With the help of symbols B_t^j, the encoding of $q_i \rightarrow \mathcal{B} q_j$ becomes $\Pi_t (Q_t^i \rightarrow B_t^j)$, whose final encoding is

$$\Pi_t (\neg Q_t^i \vee B_t^j). \quad (5)$$

This new encoding for $q_i \rightarrow \mathcal{B} q_j$, including the encoding of the relation of new symbols, requires O(L) clauses, which is the same as the naive encoding, but the number of literals is

reduced to O(L). Overall, let C_b be the number of constraints of this type, the number of clauses required is O(LC_b) and the number of extra symbols is O($L * min(C_b, N)$). Note that the number of extra symbols is always no larger than O(LN).

The method to encode $q_i \rightarrow \mathcal{F}q_j$ is similar to encoding $q_i \rightarrow \mathcal{B}q_j$. We introduce new proposition symbols

- F_t^j, for 1 ≤ i ≤ N, for 1 ≤ t ≤ L,

where F_t^j is True if and only if the machine is at state q_j at some steps > t. The encoding for the relation of the new symbols is

$$\Pi_t(\neg F_t^j \vee F_{t+1}^j \vee Q_{t+1}^j)(F_t^j \vee \neg F_{t+1}^j)(F_t^j \vee \neg Q_{t+1}^j), \quad (6)$$

and the encoding of $q_i \rightarrow \mathcal{F}q_j$ is

$$\Pi_t(\neg Q_t^i \vee F_t^j). \quad (7)$$

The idea for encoding $q_i \rightarrow \mathcal{B}_k q_j$ is straightforward, $\Pi_t(Q_t^i \rightarrow \vee_s Q_s^j)$ for s in {t-1, t-2, ..., t-k}. Hence, the corresponding SAT clauses are

$$\Pi_t(\neg Q_t^i \vee (\vee_s Q_s^j)). \quad (8)$$

There are O(L) clauses for each constraint. Let C_{bk} be the number of constraints of this type. Then the overall number of clauses of this type is O(LC_{bk}).

Encoding $q_i \rightarrow \mathcal{F}_k q_j$ is similar to encoding $q_i \rightarrow \mathcal{B}_k q_j$. The corresponding SAT clauses are

$$\Pi_t(\neg Q_t^i \vee (\vee_s Q_s^j)) \quad (9)$$

for s in {t+1, t+2, ..., t+k}.

4.4.3 Generating new tests

To generate a test, we set the start state to be the first state. The corresponding clause is

$$(Q_1^{start}). \quad (10)$$

To ensure the generated test reaches the end state, we add a clause

$$(B_L^{end} \vee Q_L^{end}). \quad (11)$$

Recall that B_L^{end} is True if and only if the machine is at q^{end} before step L. With this constraint, the length of the generated tests is not fixed to L but can be any length smaller than or equal to L.

To ensure the generated test are not the same with the training direct tests and the tests already generated, we add additional clauses for each test that has been seen. Let $\alpha = q_{t_1} q_{t_2} \ldots q_{t_M}$ be a test with length M. To avoid generating α, we add an clause

$$(\neg Q_1^{t_1} \vee \neg Q_2^{t_2} \vee \cdots \vee \neg Q_M^{t_M}). \quad (12)$$

Let τ be the number of tests that have been seen. The number of clauses of this type is τ.

Table 2 summarizes the number of symbols and the number of clauses of the SAT encoding.

Table 2: The number of symbols and clauses of the proposed SAT encoding.

# symbols for states	$O(LN)$
# symbols for Onehot0	$O(LN)$
# symbols for constraints	$O(L * min(C, N))$
# symbols, overall	$O(LN)$
# clauses for transitions	$O(LN)$
# clauses for Onehot0	$O(LN)$
# clauses for constraints	$O(LC)$
# clauses for new tests	$O(\tau)$
# clauses, overall	$O(L(N + C) + \tau)$

L is the maximum length. N is the number of states. C is the number of constraints. τ is the number of tests that have been seen.

5. EXPERIMENTAL RESULTS

We implemented the approach based on an in-house simulation based RTL verification environment for a commercial dual-core microcontroller SoC. There are 194 verification primitives and each of them corresponds to a block of C code. The tests in the verification environment are written in terms of these primitives, then compiled into object code and executed by the cores.

There are 22 cross-primitive constraints. These constraints were added manually when the primitives were developed. In the first experiment, for learning we took 30 direct tests used for verifying the on-chip system controller module that manages resource allocation, power modes, and security policies. The test length is between 46 and 63 primitives.

Fig. 5 shows the resulting upper-bound process model using the 0-prefix rule. There are 196 states in the graph including the start state and the end state. The model together with the database is then given to the SAT solver, zChaff[11], to generate tests. The maximum length L is set to 70, which is larger than the maximum length of the 30 direct tests. The SAT encoding involves 41090 symbols and 110659 clauses. Note that if using the naive encoding for *Onehot*0, the number of clauses would exceed 10^7. The run time of the SAT solver is negligible. It takes less than one second to generate a test.

The effectiveness is measured based on the coverage of a set of coverage points (CPs) defined by the verification engineers for the system controller. The original 30 direct tests cover 167 CPs. Then, 500 new tests were randomly generated by the proposed method. Fig. 6 shows the newly-covered CPs by those tests cumulatively as each new test is produced and simulated. Together, the 500 new tests cover additional 85 CPs.

Table 3 summarizes the finding with additional results. The benefit is shown by seeing that new tests produced from the process model can always improve the coverage.

Table 3: Additional coverage results

# manual tests	# originally covered CPs	# generated tests	# newly covered CPs
30	167	500	85
35	216	500	64
40	244	100	64

Fig. 7 then shows the result from a separate experiment. 30 different direct tests for verifying the system controller were used for learning. While the earlier results show coverage improvement, this result illustrates coverage frequency improvement (and also shows that the coverage improvement is not specific to a particular set of direct tests in use for learning). In this experiment, 100 new tests were generated. They cover 68 additional CPs. More importantly, the coverage frequency is improved across almost all CPs. This frequency improvement shows that the new tests can cover the similar functionality of the original tests, i.e. they capture the same intent of the original 30 tests.

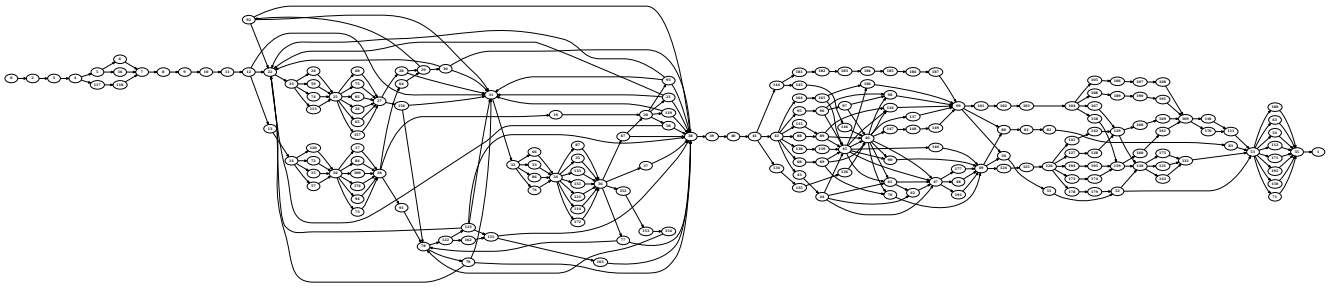

Figure 5: The upper-bound process model from the 30 direct tests

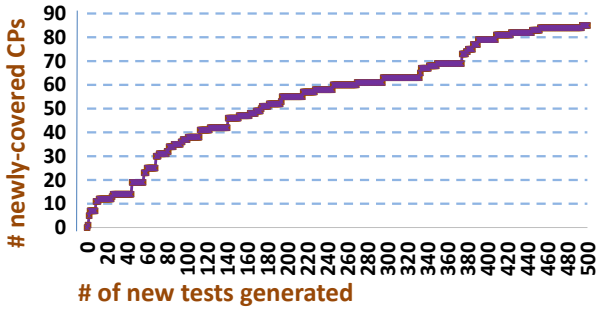

Figure 6: Coverage improvement

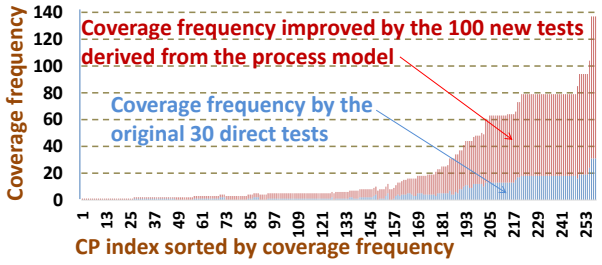

Figure 7: Coverage count improvement

6. CONCLUSION

In this work, a novel approach called constrained process discovery is proposed for learning from the direct tests developed by experts. After the learning, the software machine functions as a surrogate for the experts to generate new direct tests. The software machine comprises two components, an upper-bound model, and a constraints database. The constraints database serves as a knowledge center for accumulating verification knowledge.

We use a SAT solver for generating tests that comply with both the upper-bound model and constraints database. The SAT encoding is based on the concept of step-extension. Two techniques are used to reduce the complexity in terms of the size of SAT encoding. The number of symbols of the proposed encoding method is O(LN) and the number of clauses of the proposed encoding method is O(L(N+C)), where L is the maximum length of the generated tests, N is the number of states and C is the number of constraints.

The proposed approach is implemented in a verification environment for a commercial SoC. Experiment results show that the proposed approach not only can generate new tests to cover new functionality but also can capture the same intent of the training tests provided for learning.

Acknowledgement

This work is supported in by by Semiconductor Research Corporation, project 2012-TJ-2268 and 2016-TJ-2706.

7. REFERENCES

[1] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[2] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.

[3] P.-H. Chang and L.-C. Wang. Automatic assertion extraction via sequential data mining of simulation traces. In *2010 15th Asia and South Pacific Design Automation Conference*, pages 607–612. IEEE, 2010.

[4] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[5] C. De La Higuera. A bibliographical study of grammatical inference. *Pattern recognition*, 38(9):1332–1348, 2005.

[6] S. Dua and X. Du. *Data Mining and Machine Learning in Cybersecurity*. Auerbach Publications, Boston, MA, USA, 1st edition, 2011.

[7] E. M. Gold. Language identification in the limit. *Information and control*, 10(5):447–474, 1967.

[8] J. E. Hopcroft. *Introduction to Automata Theory, Languages and Computation: For VTU, 3/e*. Pearson Education India, 1979.

[9] W. Klieber and G. Kwon. Efficient cnf encoding for selecting 1 from n objects. In *Proc. International Workshop on Constraints in Formal Verification*, 2007.

[10] J. Z. Kolter and M. A. Maloof. *Learning to Detect Malicious Executables*, pages 47–63. Springer, 2006.

[11] Y. S. Mahajan, Z. Fu, and S. Malik. Zchaff2004: An efficient sat solver. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 360–375. Springer, 2004.

[12] M. A. Maloof. *Machine Learning and Data Mining for Computer Security: Methods and Applications (Advanced Information and Knowledge Processing)*. Springer-Verlag New York, Inc., USA, 2005.

[13] S. Ray and Y. Jin. Security policy enforcement in modern soc designs. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '15, pages 345–350, Piscataway, NJ, USA, 2015. IEEE Press.

[14] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *Security and privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 273–284. IEEE, 2002.

[15] P. Subramanyan and D. Arora. Formal verification of taint-propagation security properties in a commercial soc design. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 313. European Design and Automation Association, 2014.

[16] W. M. Van der Aalst, V. Rubin, H. Verbeek, B. F. van Dongen, E. Kindler, and C. W. Günther. Process mining: a two-step approach to balance between underfitting and overfitting. *Software & Systems Modeling*, 9(1):87–111, 2010.

[17] W. M. P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer Publishing Company, 1st edition, 2011.