# Transys: Leveraging Common Security Properties Across Hardware Designs

Rui Zhang, Cynthia Sturton
University of North Carolina at Chapel Hill
{rzhang, csturton}@cs.unc.edu

***Abstract*—This paper presents Transys, a tool for translating security critical properties written for one hardware design to analogous properties suitable for a second design. Transys works in three passes adjusting the variable names, arithmetic expressions, logical preconditions, and timing constraints of the original property to retain the intended semantics of the property while making it valid for the second design. We evaluate Transys by translating 27 assertions written in a temporal logic and 9 properties written for use with gate level information flow tracking across 38 AES designs, 3 RSA designs, and 5 RISC processor designs. Transys successfully translates 96% of the properties. Among these, the translation of 23 (64%) of the properties achieved a semantic equivalence rate of above 60%. The average translation time per property is about 70 seconds.**

## I. Introduction

The Spectre [1] and Meltdown [2] attacks, along with their variants [3], [4], have demonstrated the importance of validating the security of a processor design. To do so, one needs a comprehensive set of properties describing the security requirements of the design. Developing such a set is challenging. The high-level goals of confidentiality and integrity of a particular security domain—and availability of a machine in general—may be well understood, but mapping these goals to the cycle-by-cycle behavior of specific registers, signals, and ports in a design is difficult, and a matter of art as much as science. In practice this effort must be repeated for each new design, even for new generations of existing designs.

We present Transys, a tool that takes in a set of security critical properties developed for one hardware design and translates those properties to a form that is appropriate for a second design. The insight that led to this work is the recent research into security specification development and security validation tools, which uses properties developed for one processor design in order to evaluate the proposed methodology on a second design [5], [6], [7]. The properties must be translated manually, and this process is mentioned only in passing, but it suggests that the properties crafted for one processor design can be made suitable for a second design.

We examine the question more closely. We investigate how the translation may be done programmatically, and we build Transys to implement our approach. We go beyond processor cores and include RSA and AES implementations in our evaluation. We examine properties from the two security verification methods in use today: assertion based verification using a restricted temporal logic, and gate level information flow tracking using `set` and `assert` tags. We find that cross-design, and in the case of a processor core, cross-architecture security specification translation is feasible and practical.

The problem statement is this: given a property written for one design, produce an equivalent property suitable for the verification of a second design.

It is not always clear what "equivalent" means. For example, prior work has demonstrated that the following policy, although relatively simple, is critical to security and holds for many pipelined RISC architectures [6]:

**Policy 1.** *The zeroth general purpose register (*`GPR0`*) must always contain the value 0.*

To ensure that the above policy is upheld for a particular design $D$, a designer might craft the following property, which if proven to hold for all possible traces of execution (along with a proof that `GPR0` is initialized to 0), will enforce the desired policy.

$$P_D \doteq \texttt{wr\_enable} \rightarrow \texttt{rf\_addr} \neq 0. \quad (1)$$

Property $P_D$ states that if a write to the register file is enabled (`wr_enable`) then the register being written (`rf_addr`) is not zero—i.e., general purpose register 0 is not the target of the write.

However, the same property may not be true of a second design $D'$, even though the design enforces the same policy. Design $D'$ might require the following property:

$$P_{D'} \doteq \texttt{wr\_enable} \rightarrow \texttt{rf\_addr} \neq 0 \vee \texttt{rf\_data} = 0, \quad (2)$$

which states that writes are enabled only when `GPR0` is not the target of the write *or* when the value being written is 0. Design $D'$ does not satisfy property $P_D$ and an effort to verify the property will fail; however the underlying policy that we care about is upheld.

Given two properties written over the registers, signals, and ports of two different designs, it is not clear how to formally define equivalence between them. We therefore take an operational approach. We start with observations about how properties are likely to morph from one design to another: for example, varying pipeline stages may affect in which clock cycle a signal becomes valid; flags may be laid out differently in control registers; and additional gating signals may be used in one design, but not in another. We then define a set of steps that modify property $P_D$ in a set, limited number of ways to build a property $P_{D'}$ that is valid for design $D'$. We build a system that can reliably translate properties from one design to

another, without requiring a formal definition of the intended high-level security policies each property is in aid of.

The gist of the approach is to do the translation in three phases: the first phase substitutes the appropriate signals, ports, and register names of the second design into the property; the second phase adjusts the arithmetic expressions and timing constraints of the newly drafted property; and the third phase refines the precondition of the new property. Transys takes as input the property to be translated and the RTL implementation of both the original design and the new design. No instrumentation or manual modeling of either design is required.

Transys does not obviate the need for human involvement in security property specification. In fact, manual review of the generated properties is a required step of the Transys workflow. Transys does, however, do much of the heavy lifting for the designer, leveraging work done by others in the community tackling the security validation of similar designs, and providing an initial set of security properties. In our evaluation, we manually analyze the new properties to decide if they are semantically analogous to the original set.

We have implemented a prototype of Transys on top of Yosys[8] and it supports translating security assertions for hardware designs written in Verilog. To evaluate Transys, we collect 38 AES designs, 3 RSA designs, and 5 RISC processor designs, along with 27 temporal logic assertions and 11 information flow tracking assertions. Transys can successfully translate 96% of the properties across the evaluated hardware designs. Among these, the translation of 23 (64%) of the properties achieved a semantic equivalence rate of above 60%. The average translation time per property is about 70 seconds. The results indicate that Transys can be practically used by hardware verification teams.

## II. Security Properties

We focus on properties developed for a hardware design at the register transfer level (RTL). An RTL design defines the registers, signals, and ports in a hardware module and describes how data flows through the module in each clock cycle. Properties are written for use with a particular verification method, and each method has an associated specification language in which the properties can be expressed. We present the two main logic systems used to express security properties of hardware designs.

### A. Restricted Temporal Logic

Assertion based verification (ABV) is widely used in industry for the functional validation of hardware designs [9]. Properties expressed in a restricted temporal logic are added, in the form of assertion statements, to the RTL design and simulation-based testing or static analysis is used to find violations. Researchers have recently begun to adapt ABV for the security validation of a hardware design [10], [5], [7], [6].

The security properties that have been developed to date make use of existing industry standard libraries for expressing assertions [11] and are written in a fragment of linear temporal logic that includes the globally ($\mathbf{G}$) and next ($\mathbf{X}$) operators

$$
\begin{aligned}
LTL(\mathbf{G},\mathbf{X}) &\doteq \mathbf{G}(\phi)\\
\phi &\doteq s \rightarrow s\\
s &\doteq f \mid \mathbf{X}s\\
f &\doteq a \mid \neg f \mid f \vee f \mid f \wedge f \mid f \rightarrow f\\
a &\doteq t == t \mid t \neq t \mid true\\
t &\doteq \texttt{reg} \mid N \mid \texttt{reg} + \texttt{reg} \mid \texttt{reg} - \texttt{reg}\\
&\quad \mid \texttt{reg} << \mathrm{N} \mid \texttt{reg} >> \mathrm{N}\\
&\quad \mid \texttt{reg}[N:N]
\end{aligned}
$$

Fig. 1: The restricted temporal logic used by security properties expressed as assertions, where `reg` is a signal, register, or port in the design, and $N$ is the set of natural numbers.

$$
\begin{aligned}
property &: (set\_stmt)^* \ldots (assert\_stmt)^*\\
&\mid (set\_stmt)^* \ldots (gated\_assert\_stmt)^*\\
&\mid (set\_stmt)^* \ldots (declass\_assert\_stmt)^*\\
set\_stmt &: \text{`set'}\ \texttt{reg}\ \text{`:='}\ tag\\
assert\_stmt &: \text{`assert'}\ \texttt{reg}\ \text{`=='}\ tag\\
gated\_assert\_stmt &: \text{`assert'}\ \texttt{reg}\ \text{`=='}\ tag\ \text{`when'}\ \texttt{expr}\\
declass\_assert\_stmt &: \text{`assert'}\ \texttt{reg}\ \text{`=='}\ tag\ \text{`allow'}\ \texttt{reg}\\
tag &: \text{`high'} \mid \text{`low'}
\end{aligned}
$$

Fig. 2: The syntax used to track how information flows through a hardware design at the gate level. A property is a series of *set* statements over source variables and *assert* statements over sink variables. The *assert* statements may be made conditional using *when*. Declassification is done using *allow*.

with a syntactic restriction that conforms to the grammar shown in Figure 1. In particular, the properties are of the form $\mathbf{G}(A \rightarrow B)$, where $A$ and $B$ are boolean combinations of arithmetic expressions and may contain the $\mathbf{X}$ operator. Transys can be used to translate these properties.

### B. Information Flows

The properties expressible in the temporal logic are trace properties: individual traces of execution either satisfy or violate the given property. However, properties about how information flows through the processor are not immediately expressible as trace properties, but rather require hyperproperties [12], [13]. Whereas a trace property can be defined by a set of traces—those traces that satisfy the property, a hyperproperty is defined by a set of sets of traces—those systems that satisfy the property. Properties about confidentiality, such as asserting an absence of side channels, or about integrity, such as asserting which security domains can influence the control flow of a protected domain are examples of hyperproperties.

These properties can be handled at the language level, using typed hardware description languages [14], [15], [16]. An alternative approach is gate level information flow tracking in which shadow state added to the hardware design tracks how data flows. Standard trace properties expressed over the shadow state can then evaluate how information is allowed to flow through the original design. This approach has the advantage that existing designs, written in current industry standard hardware description languages, can be validated. The approach has been studied in the literature in a series of papers [17], [18], [19].

| Type | Description |
|---|---|
| Memory Access | *P01*: Memory value in equals register value out |
| | *P02*: Register value in equals memory value out |
| | *P03*: Memory address equals effective address |
| | *P04*: Calculation of memory address or memory data is correct |
| Exception Related | *P05*: Execution privilege matches page privilege |
| | *P06*: Updates to exception registers make sense |
| | *P07*: Privilege escalates correctly |
| | *P08*: Privilege deescalates correctly |
| | *P09*: Exception return updates state correctly |
| | *P10*: Interrupt implies handled |
| | *P11*: Enter supervisor mode is on reset or exception |
| | *P12*: Exception handling implies exception mechanism activated |
| | *P13*: Exception handler accessed only during exception, in supvr mode, or on reset |
| Control Flow | *P14*: Jumps update the PC correctly |
| | *P15*: Jumps update the LR correctly |
| | *P16*: Continuous Control Flow |
| | *P17*: Flags that influence control flow should be set correctly |
| | *P18*: Link address is not modified during function call execution |
| Update Registers | *P19*: SPR equals GPR in register move instructions |
| | *P20*: SR is not written to a GPR in user mode |
| | *P21*: SPR modified only in supervisor mode |
| Correct Results | *P22*: Destination matches the target |
| | *P23*: Reg change implies that it is the instruction target |
| Instruction Executed | *P24*: Instruction is in a valid format |
| | *P25*: Instructions unchanged in pipeline |
| | *P26*: Unspecified custom instructions are not allowed |

Table I: Security properties of OR1200 processor mined from the specification.

Gate level information flow tracking requires tagging source variables with the appropriate level (e.g., "high" or "low") of information, asserting the correct level is maintained for sink variables, and deciding when to conditionally disable the assert or under what circumstances to allow declassification. Transys can be used to translate these properties as well and we describe their syntax in Figure 2.

### C. Hardware Security Properties

We present the security properties for three classes of designs: RISC processor cores, AES implementations, and RSA implementations. Table I shows the security properties of the OR1200 processor. These security properties are collected from the literature [10], [5], [6] and can be categorized as follows: control flow related properties, exception related properties, memory access related properties, properties to ensure execution of the correct and specified instructions, and properties about correctly updating results.

Tables II and III show the security properties of the AES designs and RSA designs, respectively. These we developed manually by studying the respective specifications.

Table IV shows information flow properties for AES and RSA implementations. These properties are collected from work on gate level information flow tracking [20] and were, to the best of our knowledge, developed manually.

We used only a subset of the AES properties during the development of Transys. The rest of the properties we reserved for use in the evaluation of Transys.

| Module | Description |
|---|---|
| Key Expansion | *P27*: The round constant for each round of the key expansion should be correct. |
| | *P28*: Round keys should be derived from the cipher key correctly. |
| Substitution Box | *P29*: The S-box should avoid any fixed points and any opposite fixed points. |
| Add Round Key | *P30*: The subkey is added by combining each byte of the state with the corresponding byte of the subkey using bitwise XOR. |
| Shift Rows | *P31*: The ShiftRows step operates on the rows of the state; it cyclically shifts the bytes in each row by a certain offset. |

Table II: Security critical properties of AES cryptographic hardware mined from the specification.

| Module | Description |
|---|---|
| RSA Top | *P32*: The output cipher should be different from the input key. |

Table III: Security critical properties of RSA cryptographic hardware mined from the specification.

| Type | Description |
|---|---|
| Confidentiality | *P33*: The key or intermediate results should not directly flow to a point observable by an attacker. |
| Integrity | *P34*: The key should never be altered. |
| Isolation | *P35*: The intermediate encryption results are allowed to flow to output when the core is working in debug mode, but are prohibited under normal operation. |
| | *P36*: The key is safe to flow to the ciphertext while it should not flow to another location. |
| Timing Channel | *P37*: The secret key should not flow to the ciphertext ready signal otherwise there would be a timing side channel. |

Table IV: Information flow security properties of cryptographic hardware.

## III. Problem Statement

Given an RTL design $D_1$, a property $P_{D_1}$ that is written in a formal logic stated over the registers, signals, and ports of design $D_1$, and a second design $D_2$, how can we produce a second property $P_{D_2}$ that

1) is a valid property for the specification of design $D_2$, and
2) captures the same security policy as property $P_{D_1}$.

## IV. Threat Model

Transys is a tool to ease the development of security critical properties, and in doing so promote and encourage the security validation of hardware designs and expand the set of security critical properties validated.

The end goal is to strengthen the security of our hardware designs by eliminating bugs in the implementation or flaws in the design that are exploitable in software, post deployment, by the attacker. The attacker has knowledge of or can learn the details of the hardware design and is capable of finding and designing exploits for any bugs or flaws in the design.

Security validation is not addressing the threat of malicious trojans that get added during fabrication, nor does it prevent attacks post-deployment that involve tampering with or modifying the hardware.

Once the set of properties have been developed for a design they can be used to detect subsequent malicious modifications to the design. If the modification violates one of the security

| No. | Original | New Format | Simplified |
|---|---|---|---|
| 1 | $A \to B$ | $A \wedge C \to B$ | $(A \wedge C) \to B$ |
| 2 | | $A \vee C \to B$ | $(A \to B) \wedge (C \to B)$ |
| 3 | | $A \to B \wedge D$ | $(A \to B) \wedge (A \to D)$ |
| 4 | | $A \to B \vee D$ | $(A \wedge \neg D) \to B$ |

Table V: Possible formats of translated assertions in the new design. The simplifications are standard propositional rewrite rules.

properties, the violation can be found during verification. (The method of verification matters here—model checking, execution monitors in use post-deployment, and symbolic execution can provide guarantees about coverage, whereas simulation based testing does not.) We caution, however, that Transys uses the code of the second design to build the translated property; a well crafted trojan already extant in the code can affect the final property. Manual review of the set of properties created is a required step of the Transys workflow.

## V. Design

Transys takes as input two hardware designs and a set of security-critical properties for the first design, and outputs a set of translated properties for the second design. For each property $P$ of the first design, the goal is to produce a new property $P'$ that is written over the registers, signals, and ports of the second design and that preserves the semantics of $P$ for the second design. To achieve this goal, Transys must solve four challenges:

1) The registers, signals, and ports in the original property may not have counterparts in the second design; if they do, the counterparts will likely not have the same name.
2) The arithmetic expressions in the original property may not be appropriate for the second design.
3) The conditions required to enforce a given policy might differ between designs. For example, in the property described in the introduction, $P_D$ has the form $A \to B$, but $P_{D'}$ requires the form $A \to B \ \vee \ C$ to capture the same policy.
4) Policies often have to be stated across multiple clock cycles. For example, a `wr_enable` signal set in one clock cycle may be seen by the register file in the following clock cycle. Timing details depend on the specifics of an implementation and can vary across designs. The translated property will need to take that into account.

### A. Overview

Transys works in three passes to address the four challenges above: variable mapping pass, structural transformation pass, and constraint refinement pass. We start with an overview of the three passes and then describe each one in detail in the following sections. Figure 3 shows the workflow of Transys.

**Variable Mapping Pass.** To begin, Transys maps the registers, signals, and ports named in the properties of the first design to the registers, signals, and ports (hereafter, *variables*) of the second design (Section V-B).

We first find the matching code windows of the two designs to narrow the scope of variables to map. We then extract statistical, semantic, and structural features of each variable,

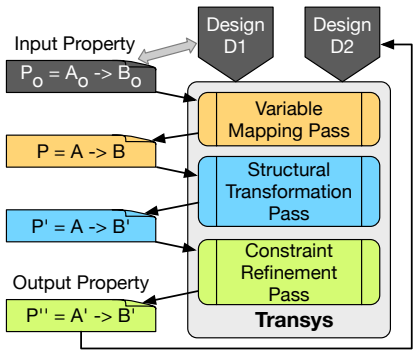


Fig. 3: The workflow of Transys.

| Type | Feature |
|---|---|
| Statistical | Variable Type (Input, Output, Wire, Reg) |
| | No. of Blocking Assignments |
| | No. of NonBlocking Assignments |
| | No. of Assignments |
| | No. of Branch Conditions |
| | No. of Always Block Conditions |
| Semantic | Variable Names |
| Structural | Dependence Graph Depth |
| | No. of Operators |
| | Centroid |

Table VI: Features from AST and PDG for variable mapping.

and calculate the distances between each pair of variables from the two designs. The variable pairs with shortest distance are used as mapped variables.

**Structural Transformation Pass.** In the next pass, Transys uses the Program Dependence Graphs (PDGs) [21] of the two designs to adjust the arithmetic expressions in the translated property. We use the PDG of the first design to learn the relationship between multiple variables in the property, and we traverse the PDG of the second design to build the arithmetic expressions of, and capture the analogous relationship between, the variables in the translated property. In practice we apply this step to only the consequent part of the property; we found the structural transformation was not needed for the antecedent. However, there is no limitation that would prevent applying this pass to the antecedent as well, should future properties require it.

**Constraint Refinement Pass.** In the third pass Transys refines the constraints of the property by adding terms to the boolean formula. Starting with the form $A \to B$, there are four possible modifications Transys might make. These, along with their simplified forms, are laid out in Table V. The first and fourth formats represent a refinement of the original property—an added constraint under which the property holds—and Transys will produce properties that require this refinement. The second and third formats are not refinements of the original property, but rather introduce new properties of the second design. This can be seen in the "Simplified" column of Table V. Transys does not produce these new properties.

### B. Variable Mapping Pass

In this pass we are concerned only with mapping variables named in one design to their appropriate counterpart in the second design.

*1) Matching Windows:* Similar to feature-based image alignment approaches [22], we search for matching variables within a reasonable range instead of within the entire code base. Modules in the Hardware Description Language by nature are good windows for matching: it keeps the semantic meaning of some functionalities and the size of each module is often reasonable to search. As the two hardware designs for assertion translation often share the same specification, we simply match modules with their names using Equation 4. We thus narrow down the scope of variables to map and search the mapped variables within corresponding modules.

*2) Extracting Features:* For each variable from the two designs within the corresponding matching windows, we extract three types of features from the Abstract Syntax Tree (AST) and the Program Dependence Graph (PDG): statistical features, semantic features, and structural features (see Table VI).

The statistical features include: the variable type; the number of times this variable appears in the left-hand-side of blocking assignments, nonblocking assignments, and assignment statements; and the number of times it appears in the branch conditions and always block conditions. The statistical features describe local statistics of a variable within a module. These features are extracted from the AST of the design.

The semantic features point to the semantic meaning of a variable. We use the variable name as a feature because it usually explains what this variable is about. For example, the variable `ex_insn` in the OR1200 processor holds the instruction in the EX pipeline stage. Different design implementations often share similar variable names for the same variable.

The structural features capture the position of a variable in a PDG. We choose three features: dependence graph depth, numbers of operators, and centroid. The dependence graph depth is the maximum length of paths of the PDG from any statement that contains the variable to the input ports of the module. The numbers of operators calculates the number of times each operation (e.g. &&, ||, $\gg$, ==, >, etc.) appears in the paths from the statements to the input ports in the PDG. The centroid measures the centrality of the dependence graph [23]. We assign each operator a weight (we use the same weight for every operator) and calculate the centrality of all the paths from the variable to the input ports of the PDG.

*3) Matching Variables:* To match variables of two hardware designs, we calculate distances between the features of pairs of variables, one from each design. The variable pairs with shortest distance are used for drafting the assertions.

For statistical features, we use the Euclidean distance for distance calculation:

$$d_{stat}(p,q) = \sqrt{(q_1 - p_1)^2 + \cdots + (q_n - p_n)^2} \tag{3}$$

For semantic features, we use the Sørensen-Dice index [24] for distance between two strings calculation:

$$d_{seman}(s_1, s_2) = 1 - \frac{2 \times |pairs(s_1) \cap pairs(s_2)|}{|pairs(s_1)| + |pairs(s_2)|} \tag{4}$$

where $pairs(s)$ is a set of character pairs in string $s$. The Sørensen-Dice index satisfies two requirements: (1) a significant substring overlap should point to a high level of similarity between the strings; (2) two strings which contain the same words, but in a different order, should be recognized as being similar. The factor 2 ensures that when the two strings are exactly the same, the distance is 0.

**Design 1**

```
always @(round_i)
begin
  case (round_i)
    1: rcon_o = 1;
    2: rcon_o = 2;
    3: rcon_o = 4;
    ......
end
```

**Design 2**

```
initial
begin
  rcon[0] = 8'h01;
  rcon[1] = 8'h02;
  rcon[2] = 8'h04;
  rcon[3] = 8'h08;
  ......
end
```

Fig. 4: Code snippets from AES designs.

**Design 1**

```
assign w0 = key[127:96];
assign keyout[127:96] =
    w0^tem^rcon(rc);
```

**Design 2**

```
always @*
begin
  w0 = key[127:096];
  w4 = w0^subword1^{rcon1,24'b0};
  w8 = w4^subword2^{rcon2,24'b0};
  w12 = w8^subword3^{rcon3,24'b0};
  .....
end
```

Fig. 5: Code snippets from AES designs.

For structural features, we use Euclidean distance (Equation 3). Each feature—depth, number of operators, and centroid—appears as a term in the calculation.

We combine the three distances by assigning each of them a weight, and thus the distance between two variables is:

$$d(v_1, v_2) = \alpha d_{seman} + \beta d_{stat} + \gamma d_{struct} \tag{5}$$

where $v_1$ and $v_2$ are variables from the first and second designs, respectively. When assigning values to parameters $\alpha$, $\beta$, and $\gamma$, we empirically choose $\alpha$ to be the largest as the semantic meanings of variable names are usually similar between designs. We choose $\beta$ to be the smallest as the detailed implementation are often different between designs, thus the structural information will be less similar.

### C. Structural Transformation Pass

In the structural transformation pass, we amend the arithmetic expressions that make up each of the terms in the property. We start by describing the challenges we met in translating the properties after the variable mapping pass. We then discuss our observations and solutions to the challenges.

*1) Challenges:* We identify three types of structural dissimilarities between designs, which Transys must handle: mapping state to array, mapping one to many, and mapping constants.

Mapping state to array refers to the case where a variable is updated according to a state machine in one design, but in another design, the variable is an array that stores all the possible values at different states of the state machine. Figure 4 shows code snippets of two AES implementations of the key expansion. In Design 1, the round constant `rcon_o` changes every time the state machine changes to the next state. In Design 2, all possible values of `rcon` are stored in an array.

Mapping one to many refers to the case where a variable from one design can be mapped to several variables in another design. For example, one design might use temporary variables to store the intermediate results of long calculations or avoid large arrays, and a second design might not. Figure 5 shows code snippets from two AES cores. The variable `keyout` in Design 1 maps to the concatenation of variables `w0`, `w4`, `w8`, and `w12` in Design 2. Mapping many to one is the dual case and also requires structural transformation.

The last type is mapping the constant values used in one design to the analogous constant values of a second design. For example, the syscall instruction is encoded differently in OpenRISC cores versus RISC-V cores. In some cases it is possible to find a linear transformation from the constant of one design to its semantic equivalent in the second design, but in other cases, such as with the syscall encoding, it is not.

*2) Transformation Algorithm:* We observe that if in the first design, the variables in the property are related to each other, the correlation among the variables in design two are often explicitly stated in the code. Thus, we leverage the PDG to build the arithmetic expressions of, and capture the analogous relationship between, the variables in the translated property.

As shown in Algorithm 1, we first check whether in the first design, the variables in the property are in the same PDG. If not, we assume that in the second design, the variables in the translated property are also not in the same PDG. In this case, we use the translation result of the Variable Mapping Pass as the result for this pass.

Otherwise, we leverage the PDG to build the property. We take the mapped variable with the highest score (`max_var`) and check whether the other mapped variables are in the same PDG as the `max_var`. If not, we move to the next variable in the vector of mapped variables and check again. We iterate until all variables in the translated property are in the same PDG as `max_var`. Then we find the variable with the second highest score (line 10).

Finally, we use a propagation algorithm in the PDG to build the new property. The propagation algorithm takes in two variables: a starting point variable, and an ending point variable (`max_var` is usually taken as the starting point). The ending point variable can be either an ancestor or a descendant of the starting point in the dependency graph. We explore both the ancestors and descendants of the starting point variable in the PDG until we hit the ending point variable. During the exploration of each node in the PDG, we replace the intermediate variables until the ending point variable is shown in the property. We stop at the ending point variable so that the property can cover the logic involving the mapped variables but does not include too long of a calculation.

There is a timing issue during the propagation. Every time we encounter a nonblocking assignment, we add a `Next` (X) to the property (or equivalently, a `prev`), indicating that there will be a delay of one clock-cycle for this assignment. Section VI shows an example of how we handle the nonblocking assignment timing.

**ALGORITHM 1:** Transformation Pass

**Input :** The property generated from the VM Pass $P$
**Input :** A set of PDGs of the Design 1 *pdgSet1*
**Input :** A set of PDGs of the Design 2 *pdgSet2*
**Input :** A map of variable mapping scores *vScoreMap*
**Output:** A new property $P'$

```
1  newAssertSet ← ∅;
2  if in_same_pdg(P, pdgSet1) then
3      max_var ← max_score(P, vScoreMap);
4      for var in P do
5          for v in vScoreMap[var] do
6              if in_same_pdg(max_var, v, pdgSet1) then
                   break ;
7          end
8          substitute(P, var, v);
9      end
10     var ← max_score(P-{max_var}, vScoreMap);
11     P' ← propagate(max_var, var);
12 else
13     P' ← P;
14 end
15 return P';
```

### D. Constraint Refinement Pass

At this point, we have a draft property of Design 2 in the form $P' \doteq A \rightarrow B$. We first check whether $P'$ is a valid property of Design 2. If it is, we are done. If it is not, then we continue with the constraint refinement pass. The goal of this step is to refine $A$ to $A'$, such that $P'' \doteq A' \rightarrow B$ is a valid property of Design 2.

We first introduce notation and define the problem; we then describe the algorithm.

*1) Notation and Problem Statement:* A hardware design unrolled for multiple clock cycles can be represented as a boolean formula $\phi$ in conjunctive normal form (CNF): $\phi \doteq (l_p \vee l_q) \wedge (l_r \vee l_s \vee l_t) \wedge \ldots$, which is written as a conjunction of clauses $\omega$, where each clause is a disjunction of literals $l$ (e.g., $\omega \doteq (l_p \vee l_q)$). A literal is either a variable $x_i$ or its negation $\neg x_i$.

Let $\phi_{D_2}$ be the CNF formula representing Design 2 unrolled for some finite but unbounded number of clock cycles. $P$ is a valid property of Design 2 if and only if the boolean formula $\phi_{D_2} \wedge \neg P$ is unsatisfiable:

$$\phi_{D_2} \models P \Leftrightarrow (\phi_{D_2} \wedge \neg P) \text{ UNSAT} \tag{6}$$

If $\phi_{D_2} \wedge \neg P$ is satisfiable, in other words, if $P$ is not a valid property of Design 2, then we look for a sequence of conjuncts $A_1 \wedge A_2 \wedge \ldots \wedge A_n$ such that the formula $F \doteq \phi_{D_2} \wedge \neg P \wedge A_1 \wedge A_2 \wedge \ldots \wedge A_n$ is unsatisfiable. Using the new conjuncts, we define $P'$ as follows:

$$P' \doteq (A_1 \wedge A_2 \wedge \ldots \wedge A_n \wedge A) \rightarrow B \tag{7}$$

Then $\phi_{D_2} \wedge \neg P'$ is equivalent to $F$: $F \Leftrightarrow \phi_{D_2} \wedge \neg P'$, and therefore equisatisfiable with $F$. If we are successful in findng $A_1 \wedge A_2 \wedge \ldots \wedge A_n$ that make $F$ unsatisfiable, then $\phi_{D_2} \wedge \neg P'$ will also be unsatisfiable, and $P'$ will be a valid property of the design: $\phi_{D_2} \models P'$.

There are two possible cases when $F$ is unsatisfiable. The first case is that the subformula $\phi_{D_2} \wedge A_1 \wedge A_2 \wedge ... \wedge A_n$ is

**ALGORITHM 2:** Refinement Pass

**Input :** A CNF formula $\phi$
**Input :** The property generated from the T Pass $P'$
**Output:** A new property with refined antecedent $P''$

1 **if** $\phi \wedge \neg P'$ *is UNSAT* **then** return $P'$;
2 **for** $t$ *in range(1,MAX_SEQ)* **do**
3 $\quad \Omega_t \leftarrow \{\omega_i | (\omega_i \text{ in } \phi) \wedge (P'_t \text{ in } \omega_i)\}$;
4 $\quad$ **for** $\omega_i$ *in* $\Omega_t$ **do**
5 $\quad\quad \Omega'_t \leftarrow \{\omega_j | (\omega_j \text{ in } \phi) \wedge (\neg l \text{ in } \omega_j) \wedge (l \text{ in } \omega_i)\}$;
6 $\quad\quad$ **for** $\omega_j$ *in* $\Omega'_t$ **do**
7 $\quad\quad\quad S \leftarrow \emptyset$; step $\leftarrow 0$;
8 $\quad\quad\quad \omega_l \leftarrow \omega_i \odot \omega_j$;
9 $\quad\quad\quad S \leftarrow S \cup \{l | l \text{ in } \omega_l\}$;
10 $\quad\quad\quad$ **while** *step < MAX_STEP* ***or*** *False not in* $\omega_l$ ***or*** $\omega_l$ *changes* **do**
11 $\quad\quad\quad\quad \omega_{ante} \leftarrow$ find_ante$(\omega_l, S)$;
12 $\quad\quad\quad\quad S \leftarrow S \cup \{l | l \text{ in } \omega_{ante}\}$;
13 $\quad\quad\quad\quad \omega_l \leftarrow \omega_{ante} \odot \omega_l$;
14 $\quad\quad\quad\quad$ step $\leftarrow$ step $+1$;
15 $\quad\quad\quad$ **end**
16 $\quad\quad\quad$ Ante $\leftarrow \bigwedge_{l \text{ in } \omega_l, l \neq I'_t} \lambda(l, 0)$;
17 $\quad\quad\quad$ **if** $\phi \wedge$ *Ante is SAT* **then**
18 $\quad\quad\quad\quad$ return $P' \wedge \neg$Ante;
19 $\quad\quad\quad$ **else**
20 $\quad\quad\quad$ **end**
21 $\quad\quad$ **end**
22 $\quad$ **end**
23 **end**
24 return Not Found;

unsatisfiable. In this case, the negation of the new conjuncts $\neg(A_1 \wedge A_2 \wedge ... \wedge A_n)$ is itself a valid property of $\phi_{D_2}$. We are not interested in this case as it does not relate to the original property we are translating. The second case is that $\phi_{D_2} \wedge A_1 \wedge A_2 \wedge ... \wedge A_n$ is satisfiable, and $F = \phi_{D_2} \wedge \neg P \wedge A_1 \wedge A_2 \wedge ... \wedge A_n$ is unsatisfiable. In this case, $A_1 \wedge A_2 \wedge ... \wedge A_n$ are the preconditions of the property $P$. This is the refinement of the constraints of the translated property.

**Constraint Refinement Problem.** Given $\phi_D$, the CNF representation of a hardware design unrolled a finite but unbounded number of clock cycles, and a draft property $P$ such that $\phi_D \wedge \neg P$ is satisfiable, find a sequence of $n$ conjuncts $A_1 \wedge A_2 \wedge ... \wedge A_n$ such that:

- $\phi_D \wedge A_1 \wedge A_2 \wedge ... \wedge A_n$ is satisfiable, and
- $\phi_D \wedge \neg P \wedge A_1 \wedge A_2 \wedge ... \wedge A_n$ is unsatisfiable.

*2) Constraint Refinement Algorithm:* The constraint refinement algorithm works by finding *conflict clauses* in the CNF representation of the design. For each literal $l$ appearing in the clause $\omega$ that contains $B$ (the consequent of the property), the algorithm searches for a clause $\omega'$ in $\phi_D$ such that $\neg l$ appears in the clause. These two clauses are conflict clauses. If we force all other literals appearing in $\omega$ and $\omega'$ to evaluate to false, then $\phi_D$ will be unsatisfiable.

Let $\lambda(l, v)$ be a function that takes in a literal $l \in \{x, \neg x\}$ and a truth value $v \in \{\texttt{true}, \texttt{false}\}$ and returns a new literal $l' \in \{x, \neg x\}$ such that $l'$ evaluates to `true` when $l$ evaluates to $v$.

$$\lambda(l, v) = \begin{cases} x & \text{if } l = x,\ v = \texttt{true} \\ x & \text{if } l = \neg x,\ v = \texttt{false} \\ \neg x & \text{otherwise} \end{cases}$$

Given a CNF formula $\phi$, if there exist conflict clauses $\omega_i$ and $\omega_j$ in $\phi$, where $\omega_i = l_{i1} \vee ... \vee l_{is} \vee x_c$, and $\omega_j = l_{j1} \vee ... \vee l_{jt} \vee \neg x_c$, then $\phi \wedge \lambda(l_{i1}, 0) \wedge ... \wedge \lambda(l_{is}, 0) \wedge \lambda(l_{j1}, 0) \wedge ... \wedge \lambda(l_{jt}, 0)$ is unsatisfiable. This is because $x_c \wedge \neg x_c$ is unsatisfiable. By assigning all other literals in the two clauses $\omega_i$ and $\omega_j$ to 0, subformula $\omega_i \wedge \omega_j$ can be simplified to $x_c \wedge \neg x_c$, which is unsatisfiable. Thus, $P = \neg(\lambda(l_{i1}, 0) \wedge ... \wedge \lambda(l_{is}, 0) \wedge \lambda(l_{j1}, 0) \wedge ... \wedge \lambda(l_{jt}, 0))$ is a property of $\phi$.

Algorithm 2 takes a CNF formula $\phi_D$ and the property to be refined $P'$ as inputs. It first checks whether $P'$ is a valid property of $\phi_D$, if it is, the algorithm just returns $P'$. Otherwise, it searches for clauses that contain the property $P'$ (line 3), and for each clause that contains $P'$, it searches for its conflict clauses (line 5). By combining the results of these two sets of clauses, the algorithm produces the new property for $\phi_D$.

*3) Greedy Search:* The results we obtained from combining $\omega_i$ and $\omega_j$ often do not include any interesting preconditions, but just a restatement of the property $P'$. This is because when unrolling the design together with the invariant, some clauses to connect the invariant with the design need to be added to $\phi_D$. To get the preconditions, we have to search further.

We first define the resolve operator $\odot$: given two clauses $\omega_i$ and $\omega_j$, for which there is a unique variable $x$ such that one clause has a literal $x$ and the other has $\neg x$, $\omega_i \odot \omega_j$ contains all the literals of $\omega_i$ and $\omega_j$ with the exception of $x$ and $\neg x$.

Starting from the conflict clauses (line 8), we search for more clauses that can introduce potential precondition variables (line 11). $\omega_l$ keeps track of the current resolved clause. Every time we find a new conflict clause, we resolve $\omega_l$ with the new clause (line 13). The new $\omega_l$ clause can still make $\phi$ unsatisfiable. We keep expanding the resolved clause, until we reach the maximum step, or `False` shows in $\omega_l$, or $\omega_l$ does not change any more (line 10). Then we generate the antecedent from $\omega_l$ and check whether it satisfies the requirements (line 16). If yes, we output the new invariant; otherwise, we keep on searching (line 17-19).

During the search in `find_ante`, we search for clauses greedily. The goal is to keep the antecedent short to be readable and managable. Thus, every time we find a conflict clause, we only find the one that introduces one new variable to $\omega_l$ (we use a set $S$ to keep track of the found variables).

*4) Timing in the Assertions:* A property $P'$ is asserted at each clock cycle: $\phi \wedge \neg P' \doteq \phi \wedge \neg P'_{t=1} \wedge \neg P'_{t=2} \wedge ... \wedge \neg P'_{t=MAX\_SEQ}$. To determine the timing constraints in the assertion, the search for conflict clauses takes place only within a specific clock cycle ($\phi \wedge \neg P'_{t=t_i}$, line 2 in Algorithm 2), instead of all clock cycles together ($\phi \wedge \neg P'$).

The generated property $P''$ from the refinement pass can contain literals in different time steps. We rank them according to the timing information, and add the delays between them.

### E. Property Does not Exist

A property of one design may not be true of a second design. This can happen when the two designs implement different specifications or when one of the designs implements only part of the specification. For example, some of the AES designs we collected implemented only encryption and did not implement decryption. Thus, the properties related to decryption cannot be translated to these designs. Another example is that for RISC-V processors, there are three privilege levels, but for OpenRISC processors, there are only two privilege levels. Thus, properties related to the middle privilege level of the RISC-V processor do not have corresponding properties in the OpenRISC processors. In these cases Transys will typically fail to produce a translation, which is a reasonable outcome.

### F. Bugs in the Code

The structural transformation and constraint refinement passes leverage the second design itself to translate the property. This raises a concern: If there is a bug in the design, it will be captured in the translated property. This is true. Transys is meant to be used as an aide to the verification team tasked with writing security critical properties of a design. Transys does the heavy lifting of producing a candidate translation, but it does not obviate the need for human involvement in property design. A manual review of the translated properties is a required part of the workflow.

## VI. Implementation

We implement Transys based on the Yosys Open Synthesis Suite [8], a framework for Verilog synthesis. Transys is implemented in C++ with approximately 4,500 lines of code. The assertions are implemented in SystemVerilog. Each Pass is implemented as a command in Yosys: the Variable Mapping Pass and the Transformation Pass are implemented as new commands (`match_variables` and `transform`), and the Refinement Pass is implemented by modifying the `sat` command. We also implement three assisting commands for building the program dependence graphs (`build_pdg`), parsing security assertions to a standard format (`read_assertlist`) and adding assertions to the designs for refinement and validation (`append_assertlist`).

We build the PDGs on the Register Transfer Level Intermediate Language (RTLIL) representation in Yosys. Each node in the PDG is a `Cell` or a `Wire` object, which represents the netlist data; or a `Switch`, a `Case`, or a `Sync` object, which represents the decision trees and synchronization declarations; or an `assignment` block, which we build to represent the assign statements. Each edge represents either the control or data dependence. To build the PDG, we first convert the objects into nodes. An edge from node A to node B is added if the inputs to B depend on the outputs of A.

For the timing delays caused by non-blocking assignments from the Transformation Pass, we add a state machine to keep track of the signal values in different clock cycles. For example, if we have an assertion `(a == prev(b))`, the implementation of this assertion is:

```
always @(posedge clk)
begin
  prev_b <= b;
end
assert property (a == prev_b);
```

## VII. Evaluation

Our evaluation aims to answer the following questions: (1) whether Transys can successfully translate security-critical assertions from one design to another; (2) whether the translated assertions are valid and capture the meaning of the original assertions; (3) whether Transys is practical in terms of runtime; (4) how the translation results are affected by bugs in the second design.

### A. Experiment Setup and Dataset

The experiments are performed on a machine with the Intel Xeon E5-2620 V3 12-core CPU (2.40GHz, dual-socket) and 62GB RAM. We evaluate Transys on 38 AES designs, 3 RSA designs, and 5 RISC processor designs in total.

Specifically, we collect 36 open-source AES cores from GitHub and OpenCores. Of these, 18 are implemented in Verilog and are evaluated. The remaining 20 are written in SystemVerilog, which Transys currently does not support. In addition, we collect 20 AES cores with injected trojans from TrustHub [25], [26]. We also collect 11 open-source RSA cores from GitHub, OpenCores, and TrustHub, and 3 of the them are implemented in Verilog. For CPU designs, we collect 5 open-source RISC processor, 3 of them are implementations of the OpenRISC architecture (OR1200, Espresso, Cappuccino) and 2 of them are implementations of the RISC-V architecture (OpenV, Picorv32).

To evaluate Transys on the AES and RSA designs, we draft 17 assertions for 3 designs to feed as input to Transys (see Table VII). We also collect 14 information-flow security assertions for AES and RSA cores from the IFT Model project [20] (see Table IX). These assertions are drafted for 3 AES and 3 RSA implementations, and cover properties about confidentiality, integrity, isolation and timing channels. The first 9 assertions in Table IX are drafted for general AES and RSA designs, and the last 5 assertions are drafted for specific malicious designs. Thus, we use the first 9 assertions for our translation evaluation. We use the last 5 for evaluating the security impact of translated assertions (see Section VII-G). To evaluate Transys on the processor designs, we collect 10 security assertions for OR1200 processors from the SPECS [5], Security Checkers [10], and SCIFinder [7] projects (see Table VIII). These assertions represent the 6 types of security properties in Table I.

### B. Translation Results

To evaluate whether Transys can successfully translate security-critical assertions from one design to another, we test whether it can successfully generate valid assertions for the new designs. Table X shows the main translation results. Figures 6, 7, 8, and 9 show the detailed results of the translation rate for each assertion.

| A No. | Assertions |
|---|---|
| A27-01 | (keysched.round_i == 1) → (keysched.rcon_o == 'h1) |
| A27-02 | (keysched.round_i == 2) → (keysched.rcon_o == 'h2) |
| A27-03 | (keysched.round_i == 3) → (keysched.rcon_o == 'h4) |
| A27-04 | (keysched.round_i == 4) → (keysched.rcon_o == 'h8) |
| A27-05 | (keysched.round_i == 5) → (keysched.rcon_o == 'h10) |
| A27-06 | (keysched.round_i == 6) → (keysched.rcon_o == 'h20) |
| A27-07 | (keysched.round_i == 7) → (keysched.rcon_o == 'h40) |
| A27-08 | (keysched.round_i == 8) → (keysched.rcon_o == 'h80) |
| A27-09 | (keysched.round_i == 9) → (keysched.rcon_o == 'h1b) |
| A27-10 | (keysched.round_i == 10) → (keysched.rcon_o == 'h36) |
| A28-01 | (keysched.state == 4) → (keysched.next_key_reg[31:0] == keysched.next_key_reg[63:32] ⊕ keysched.last_key_i[31:0]) |
| A28-02 | (keysched.state == 4) → (keysched.next_key_reg[63:32] == keysched.next_key_reg[95:64] ⊕ keysched.last_key_i[63:32]) |
| A28-03 | (keysched.state == 4) → (keysched.next_key_reg[95:64] == keysched.next_key_reg[127:96] ⊕ keysched.last_key_i[95:64]) |
| A28-04 | (keysched.state == 4) → (keysched.next_key_reg[127:96] == keysched.col_t ⊕ keysched.last_key_i[127:96] ⊕ {keysched.rcon_o, 32'h0}) |
| A29-01 | (aes_sbox.d ⊕ aes_sbox.a != 8'hff) |
| A29-02 | (aes_sbox.d != aes_sbox.a) |
| A32-01 | (rsa.msg_in != rsa.msg_out) |

Table VII: Security critical assertions of cryptographic hardware. Assertion A27-01—10 and A28-01—04 are drafted for the AES09 design; Assertion A29-01—02 are for AES11; Assertion A32-01 is for RSA03. The first number in A No. refers to the property number in Table II.

| ANo. | Example Assertions |
|---|---|
| A01 | ((or1200_ctrl.ex_insn&'hFC000000)≫26=='h21)→ (or1200_rf.rf_dataw==dcpu_dat_o) |
| A03 | ((or1200_ctrl.ex_insn&'hFC000000)≫26=='h21)→ (dcpu_adr_o==operand_a+ex_simm) |
| A04 | (or1200_rf.rf_we==1)→(or1200_rf.rf_addrw!=0)\| (or1200_rf.rf_dataw==0) |
| A08 | ((or1200_ctrl.ex_insn&'hFC000000)≫26==9)→ (or1200_sprs.to_sr==or1200_except.esr) |
| A09 | ((or1200_ctrl.ex_insn&'hFC000000)≫26==9)→ (or1200_genpc.pc==or1200_except.epcr) |
| A15 | ((or1200_ctrl.ex_insn&'hFC000000)≫26==1)→ (or1200_rf.rf_addrw==9) |
| A17 | ((or1200_ctrl.ex_insn&'hFFE00000)≫21==1826)& (operand_a>operand_b)→(or1200_sprs.to_sr[9]==1) |
| A19 | ((or1200_ctrl.ex_insn&'hFC000000)≫26==48)→ (or1200_sprs.spr_dat_o==operand_b) |
| A23 | ((or1200_ctrl.ex_insn&'hFC000000)≫26=='h38)→ ((or1200_ctrl.ex_insn&'h03e00000)≫21==or1200_rf.addrw) |
| A26 | ((or1200_ctrl.ex_insn&'hFC000000)≫26!='h1c) |

Table VIII: Security critical assertions of the OR1200 design. The first number in A No. refers to the property number in Table I.

(1) For AES designs, the overall translation rate is 93%. The 8 failures in the Transformation Pass occur in translating A28 to the AES08 design, and A29 to the AES06 and AES12 designs. The reason that the Transformation Pass fails is that the highest-score variable found in the first pass is incorrect, making it impossible to find a subgraph in the PDG that includes at least two variables in the assertions.

For the AES05 design, the implementation of one module is missing in the code we collected, which caused 16 failures in the Refinement Pass. Transys can translate the assertions in the first two passes, but fails in the third pass as the code is incomplete. This shows that our first two passes do not rely on the completeness of the code base, but the third pass requires that the code should be complete. If we comment out the part of the code that instantiates the missing module

| A No. | Assertion | Core |
|---|---|---|
| A36-01 | set key[0] := high; assert cipher[0] == high | AES-04 |
| A36-02 | set key[1] := high; assert cipher[7:0] == high | AES-04 |
| A36-03 | set key[1] := high; assert cipher[31:0] == high | AES-04 |
| A36-04 | set key[1] := high; assert cipher[63:0] == high | AES-04 |
| A33-06 | set indata[1] := high; assert count[1] == low | RSA-03 |
| A36-05 | set inExp[1] := high; assert cipher[1] == high when ready == 1 | RSA-03 |
| A36-06 | set inExp[0] := high; assert cipher[0] == low | RSA-03 |
| A37-01 | set inExp[0] := high; assert ready == low | RSA-03 |
| A37-02 | set inExp[1] := high; assert ready == low | RSA-03 |
| A33-01 | set key[0] := high; assert Antena == low | AES-T400 |
| A33-02 | set key[0] := high; assert TSC_SHIFTReg[0] == low | AES-T400 |
| A33-03 | set key[0] := high; assert Capacitance[0] == low | AES-T1100 |
| A33-04 | set key[1] := high; assert Capacitance[1] == low | AES-T1100 |
| A33-05 | set key[1]:=high; assert Capacitance[0] == high | AES-T1100 |

Table IX: Information flow assertions of cryptographic hardware. The first num in A No. refers to the property num in Table IV, III.

in the original design, Transys can successfully translate the assertions to AES05.

(2) For AES designs with trojans, Transys successfully translates all assertions to the 20 trojan-injected AES designs. For example, as shown in Figure 7, Transys translates 4 AES Information Flow Tracking assertions written in the AES-04 design (a trojan-free design) to the 20 AES designs with different trojans injected. The trojans include leaking the secret key through AM radio, leakage current, spread spectrum communications, and draining the battery to cause denial-of-service [25], [26]. In this case, the translated assertions can potentially be used to detect the injected trojans.

(3) For processor designs, we translate assertions from the OR1200 to 5 processor designs in two different architectures. We found that the assertions A19 and A26 do not exist in the two RISC-V cores: A19 and A26 are about the `l.mtspr` instruction and custom instructions, which are not implemented in the two RISC-V cores.

We first evaluate the remaining 46 of the 50 total translations, and among those the translation rate is 85%. Among the 7 failed cases, 3 of them fail in the Transformation Pass and 4 of them fail in the Refinement Pass—Transys cannot find valid preconditions to make the consequent true. All the failed cases happen when we try to translate the assertions from OR1K designs to RISC-V designs: 2 of them are to the OpenV core, and 5 of them are to the Picorv32 core.

We separately evaluate the 4 translations for which the assertion does not exist in the target design. Transys successfully translates 3 of them. These 3 new assertions are valid but the policies they capture are different than the original assertions' policies. The false positive rate here is 75%.

(4) For RSA designs, we translate 1 assertion mined from the specification, and 5 Information Flow Tracking assertions. All of them are successfully translated to the new designs.

(5) We also test Transys by translating the assertions back to the original designs. Transys successfully translates all assertions back to the original designs. This implies the variable mapping pass can map the variables to themselves, and the second and third pass preserve the structure of the assertions.

| Designs | Total Translations | Total Succ | Fail in VM Pass | Fail in T Pass | Fail in R Pass | Total Transl. Rate |
|---|---|---|---|---|---|---|
| **AES** | 360 | 336 | 0 | 8 | 16 | 93% |
| **AES w/ Trojan** | 400 | 400 | 0 | 0 | 0 | 100% |
| **CPU** | 46 | 39 | 0 | 3 | 4 | 85% |
| **RSA** | 18 | 18 | 0 | 0 | 0 | 100% |
| **Total** | 824 | 793 | 0 | 11 | 20 | 96% |

Table X: Main results of assertion translation for 18 AES designs, 20 AES designs with trojans, 5 processor designs, and 3 RSA designs.

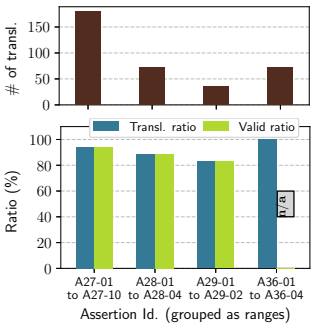


Fig. 6: AES01—AES18 translation results: total transl. number and success transl. rate.

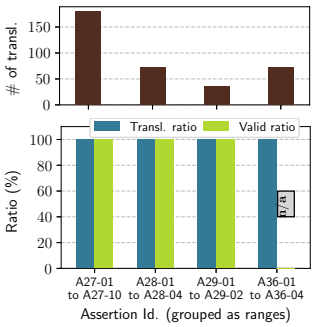


Fig. 7: AES-T100—AES-T2100 transl. results: total transl. number and success transl. rate.

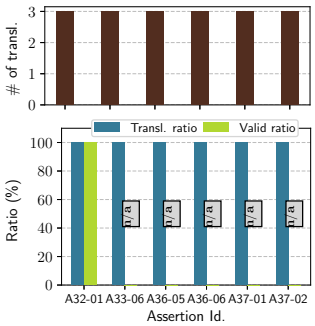


Fig. 8: RSA01—RSA03 translation results: total transl. number and success transl. rate.

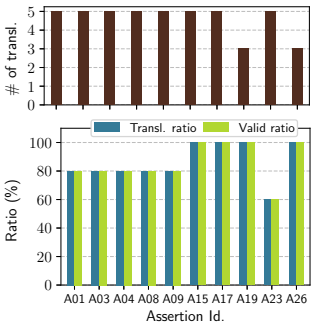


Fig. 9: CPU translation results: total transl. number and success transl. rate.

### C. Quality

To evaluate the quality of the translated assertions, we first check whether the translated assertions are valid for the target design using the model checking tool Cadence IFV. We then manually review the assertions alongside the design specifications to determine whether the translated assertions are semantically equivalent to the original assertions.

*1) Validity:* We check whether the translated assertions are valid by adding them to the target designs and running Cadence IFV. Figures 6, 7, 8, and 9 shows the results. For the nine Information Flow Tracking assertions, we do not have the tool to check the validity of the translated assertions (167 in total) and thus their validity result is not available. All the other 626 translated assertions can pass verification by Cadence IFV, indicating that the assertions Transys generates are valid.

*2) Equivalence:* Figures 10, 11, 12, and 13 show the results of the equivalence checking. Type equivalence refers to the case that the translated assertion and the original assertion belong to the same type or module of security properties, as given in column 1 of Tables I, II, III, and IV. Semantic equivalence refers to the case that the translated assertion and the original assertion are semantically the same.

The translation of assertions to trojan-injected AES designs achieves 100% semantic equivalence rate. For other designs, the translation of 23 (64%) assertions has type and semantic equivalence rate above 60% (between 60% and 100%). The translations of the remaining 13 (36%) assertions have type and semantic equivalence rate between 20% to 50%. The low rates mainly happen in two cases: the translation of Information Flow Tracking assertions and the translation from OpenRISC cores to RISC-V cores.

The main reason for the translated assertions to fail to capture the meaning of the original assertion is because the variable mapping pass fails to map to an accurate variable or even fails to map to the correct module in the target design. In all our experiments, we choose the parameters in the Variable Mapping Phase empirically to be $\alpha : \beta : \gamma = 3 : 2 : 1$. This combination works well in most cases, but not all of them.

| No. | Translation Results |
|---|---|
| Original | (keysched.state == 4) → (keysched.next_key_reg[31:0] == keysched.next_key_reg[63:32] ⊕ keysched.last_key_i[31:0]) |
| AES01 | (round_ctr_reg[0]) & (key_mem_we) & (!round_ctr_inc) → (key_mem_new == key[255:128]) |
| AES02 | u1.r1.t0.w0 == u1.r1.t0.key[127:96] |
| AES03 | (key_exp.key_start==1)&(key_exp.round[1:0]==2'b01)→#1 (key_exp.wr_data==prev(key_exp.key_in[255:192]))\| (key_exp.wr3==0) |
| AES04 | a1.k0b == a1.k0a ⊕ a1.k4a |
| AES05 | n.a. |
| AES06 | (!u0.kld)→#1(u0.w[0]==prev(u0.w[0]⊕u0.subword⊕u0.rcon)) |
| AES07 | a1.k0a == prev({a1.k0[31:24] ⊕ rcon, a1.k0[23:0]}) |
| AES08 | n.a. |
| AES09 | (keysched.state == 4) → (keysched.next_key_reg[31:0] == keysched.next_key_reg[63:32] ⊕ keysched.last_key_i[31:0]) |
| AES10 | AES_CORE_DATAPATH.KEY_EXPANDER.key[3] == AES_CORE_DATAPATH.KEY_EXPANDER.key_in[31:0] |
| AES11 | (!u0.kld) → #1 (u0.w[1] == prev(u0.w[0] ⊕ u0.w[1] ⊕ u0.subword ⊕ u0.rcon)) |
| AES12 | w0_next == sbox_out ⊕ rcon ⊕ w0 |
| AES13 | w4 == key[127:96] ⊕ subword ⊕ 16777216 |
| AES14 | w4 == w0 ⊕ subword ⊕ {rcon2[31:24],24'b0} |
| AES15 | wNext[1] == w[1] ⊕ wNext[0] |
| AES16 | roundkey_text == mixcolumns_text ⊕ okey |
| AES17 | roundkey_text == mixcolumns_text ⊕ okey |
| AES18 | w7 == key[127:96] ⊕ key[95:64] ⊕ key[63:32] ⊕ key[31:0] ⊕ subword ⊕ 16777216 |

Table XI: The results of translating A28-01 to 18 AES designs.

### D. Case Studies

In this section, we show 3 examples: (1) translation from one AES design to another AES design; (2) translation from one processor design to two different processor designs from two architectures (OR1K architecture and RISC-V architecture); (3) translating an Information Flow Tracking assertion from one trojan-free AES design to a trojan-injected design.

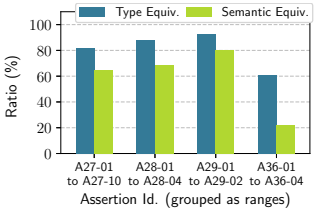


Fig. 10: Type and semantic equiv. for AES01—AES18 designs.

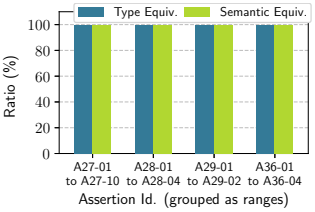


Fig. 11: Type and semantic equiv. for AES-T100—AES-T2100 designs.

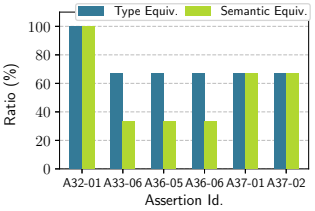


Fig. 12: Type and semantic equiv. for RSA01—RSA03 designs.

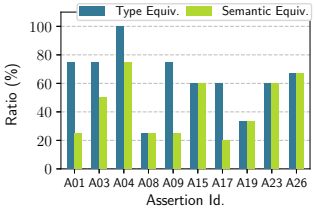


Fig. 13: Type and semantic equiv. for CPU designs.

| Pass | Translation Results |
|---|---|
| VM Pass | (key_exp.pstate==4)→(key_exp.key_in[31:0]== key_exp.key_in[63:32]⊕key_exp.key_in[31:0]) |
| ST Pass | (key_exp.pstate==4) → (key_exp.wr_data==key_exp.key_in[255:192]) |
| | (key_exp.pstate==4) → (key_exp.wr_data==key_exp.key_in[191:128]) |
| | (key_exp.pstate==4) → (key_exp.wr_data==key_exp.key_in[127:64]) |
| | (key_exp.pstate==4) → (key_exp.wr_data==key_exp.key_in[63:0]) |
| | i_key == key_exp.key_in |
| CR Pass | (key_exp.key_start==1)&(key_exp.round[1:0]==2'b01)→#1 (key_exp.wr_data==prev(key_exp.key_in[255:192]))\|(key_exp.wr3==0) |
| | (key_exp.key_start==0)&(key_exp.key_start_L==1) &(key_exp.round[1:0]==2'b01)→#1 (key_exp.wr_data==prev(key_exp.key_in[191:128]))\|(key_exp.wr3==0) |
| | (key_exp.key_start==0)&(key_exp.wr3==1)&(key_exp.init_wr3==1) &(key_exp.round[1:0]==2'b01)→#1 (key_exp.wr_data==prev(key_exp.key_in[127:64]))\| (key_exp.wr3==0) |
| | (key_exp.key_start==0)&(key_exp.wr3==1)&(key_exp.init_wr4==1) &(key_exp.round[1:0]==2'b01)→#1 (key_exp.wr_data==prev(key_exp.key_in[63:0]))\| (key_exp.wr3==0) |
| | i_key == key_exp.key_in |

Table XII: Detailed results of translating A28-01 to the AES03 design. VM: Variable Mapping, ST: Structural Transformation, CR: Constraint Refinement.

| No. | Translation Results |
|---|---|
| Original | (or1200_rf.rf_we==1)→(or1200_rf.rf_addrw!=0)\| (or1200_rf.rf_dataw==0) |
| OR1200 | (or1200_rf.rf_we==1)→(or1200_rf.rf_addrw!=0)\| (or1200_rf.rf_dataw==0) |
| Espresso | (mor1kx_rf_espresso.rfa_o_use_last)& (mor1kx_rf_espresso.result_last[0]==0)& (mor1kx_rf_espresso.rfd_last==mor1kx_rf_espresso.rfa_r)& (mor1kx_rf_espresso.rfa_adr_i[0])& (mor1kx_rf_espresso.rfa_o[0]==0)→ (mor1kx_rf_espresso.rfa_adr_i≠0)\| (mor1kx_rf_espresso.rfa_o==0) |
| Cappuccino | mor1kx_rf_cappuccino.rf_wradr== mor1kx_rf_cappuccino.wb_rfd_adr_i |
| | (mor1kx_rf_cappuccino.rf_wradr)& (mor1kx_rf_cappuccino.rf_wrdat)→ (mor1kx_rf_cappuccino.rf_wrdat==0)\| (mor1kx_rf_cappuccino.rf_wraddr!=0) |
| OpenV | n.a. |
| Picorv32 | picorv32.dbg_mem_rdata == picorv32.mem_rdata |

Table XIII: The results of translating A04 to 5 CPU designs.

*1) Example 1:* We show the details of translating the assertion A28-01 from AES09 to all AES designs. Table XI shows the resulting assertions. For the assertions in AES02, AES03, AES12, we classify them as in the same type as the original assertion, but not as having equivalent semantics. For the assertions in AES16 and AES17, they belong to the calculation of round keys, and thus are neither type equivalent nor semantically equivalent to the original assertion.

Table XII shows the detailed results of translating assertion A28-01 from AES09 to AES03. After the Variable Mapping Pass, `keysched.next_key_reg` and `keysched.last_key_i` are both mapped to `key_exp.key_in`. The assertion generated is not valid yet. After the Transformation Pass, Transys outputs 5 assertions. These assertions are generated from the part of the PDG that contains the variable `key_exp.key_in`. Only the 5th assertion is valid. Finally, from the Refinement Pass, all the 4 assertions are refined and are valid. It is worth noting that the antecedents generated from the Refinement Pass are neither close to the part of the code of the consequent nor similar to the original code, and thus it would be difficult for a human to figure them out manually.

*2) Example 2:* Table XIII shows the translation results for translating assertion A04 to five processor designs. The translation fails in the Refinement Pass when translating the assertion to the OpenV design. For the other designs, Transys can successfully generate valid assertions. The translated assertions for the OR1200, Espresso, and Cappuccino processors are semantically equivalent. These three designs are all implementations of the OR1K architecture and it is easier to translate assertions among them. The assertion for the Picorv32 does not capture the same semantic meaning, but it also belongs to the type of security properties that are relavent to the memory.

*3) Example 3:* In this example, the Information Flow Tracking assertion A36-01 for the AES04 design is translated to the AES-T400 design. In the AES-T400 design, the injected trojan utilizes an unused pin to generate an RF signal that can be used to transmit the key bits. The leaked data can be received by an AM radio, and can be interpreted with a specific beep scheme. The trojan is implemented in two additional modules: `AM_Transmission` and `Trojan_Trigger`. When a predefined plaintext is observed, the trojan will be triggered and the `AM_Transmission` module will output the key to the *Antena* signal following the beep scheme to leak data.

Ideally, the key will flow only to the output ciphertext (A36-01). The result of our translation for A36-01 to the AES-T400 design is: `set key[0] := high; assert cipher[0] == high`. This indicates that Transys can successfully translate the assertion to a new design and is not influenced by the two additional modules of the trojan.

## E. Performance

We evaluate the total time it takes for Transys to translate each assertion from a source design to a target design. Fig-

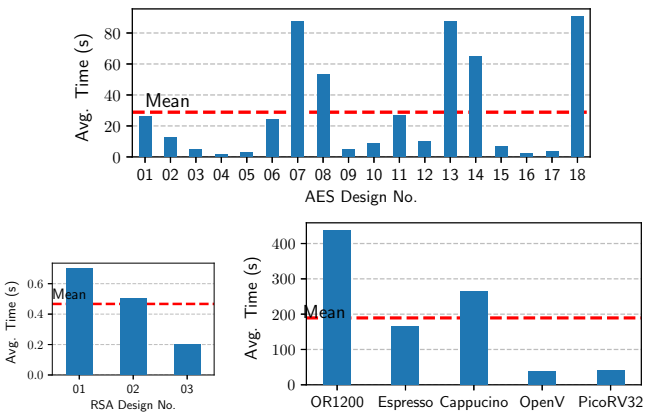


Fig. 14: Translation time for the AES, RSA and CPU designs.

| Assertion | VM Pass | ST Pass | CR Pass |
|---|---|---|---|
| Total Transl. | 360 | 352 | 336 |
| Valid Ratio | 14% | 52% | 93% |

Table XIV: Accumulative valid ratio of each pass for AES designs.

| Assertion | VM Pass | ST Pass | CR Pass |
|---|---|---|---|
| Total Transl. | 46 | 43 | 39 |
| Valid Ratio | 39% | 59% | 85% |

Table XV: Accumulative valid ratio of each pass for CPU designs.

| Orig Assert No. | Trans. assert can detect trojans in |
|---|---|
| A33-01, A33-02 | AES-T1600, AES-T1700, AES-T400 |
| A33-03, A33-04 A33-05 | AES-T100, AES-T1000, AES-T1100, AES-T1200 AES-T200, AES-T700, AES-T800, AES-T900 |

Table XVI: Results of security impact of translated assertions to detect trojans in AES cores.

ure 14 shows the results. The translation times for the trojan-injected AES designs are similar to the time for the trojan-free AES design, and are not shown due to space constraints.

We observe that the translation time varies across different designs, depending on their complexity. The average times for translating one assertion for AES designs, RSA designs and CPU designs are 28.8 seconds, 0.46 seconds, and 189 seconds, respectively. For AES and RSA designs, most of the translation time is spent on the Refinement Pass. For processor designs, most of the translation time is spent on the Variable Mapping Pass. The maximum average property-translation time is 436.8 seconds for the OR1200 design. The results suggest that Transys is practical enough to be used by hardware designers on a daily basis to quickly generate security assertions through translating existing ones.

### F. Effectiveness of Each Pass

We evaluate the effectiveness of each pass on translating assertions across the AES designs and the processor designs. Tables XIV and XV show the ratio of valid results at the end of each pass. We observe that each pass increases the valid-to-invalid ratio substantially, indicating that each pass is effective.

### G. Security Impact

In this section, we discuss the security impact of the translated information flow tracking assertions when there is a vulnerability in the code. Assertions A33-01—A33-05 in Table IX can detect trojans in AES-T400 and AES-T1100 [20]. We translate these five assertions to the AES cores with trojans.

We do not have access to the information flow tracking tool [20] needed to add the tracking logic necessary to verify whether the translated assertions can detect trojans. Therefore, we instead compare the translated assertions with the original assertions, and compare the trojans between designs. If the assertions are logically equivalent, and the information leakage circuits are the same other than the triggering mechanism, then we infer that the translated assertions would detect the injected trojans as well.

Table XVI shows the results. The translated assertions of A33-01 and A33-02 would detect trojans in three AES designs, and the translated assertions of A33-03—A33-05 would detect trojans in eight AES designs. For the remaining nine trojan-injected designs, we do not have assertions that can detect the trojans and therefore we cannot determine whether translated assertions would detect them.

### H. Bugs in the Code

We discuss three examples to show the translation results of Transys when there is a bug in the design. For different types of bugs, the translation results of Transys can be: failing to translate, outputting trivially true assertions, or propagating the bug to the resulting assertions.

*1) Translation Failed:* The first example shows the case of translation failure. In the AES05 design we mentioned in Section VII-B, part of the code base is missing. When we use Transys to translate the assertions to the AES05 design, we get the error message in the Refinement Pass showing that some modules or cells are not part of the design. Thus, one possible reason for translation failure is missing parts of the code. This corresponds to the case of no refinement output at all.

*2) Trivial Assertions:* The second example shows the case that a certain constraint should be explicitly stated in the design, but it is not. We show the `GPR0` bug in the OpenRISC cores. In the OR1K specification, the general purpose register `R0` should always be set to zero [27]. A violation of this property can lead to malicious modification of the memory data or memory address in calculation. This bug exists in both the Espresso and the Cappuccino designs [6].

We translate the assertion that enforces `R0` to always be 0 (A04 in Table VIII) from the OR1200 to both the Espresso and the Cappuccino designs. The results are shown in Table XIII. The result assertion for the Espresso design can be simplified to `(mor1kx_rf_espresso.rfa_adr_i≠0)`→ `(mor1kx_rf_espresso.rfa_adr_i≠0)`. The result assertion for the Cappuccino design can be simplified to `(mor1kx_rf_cappuccino.rf_wraddr≠0)`→ `(mor1kx_rf_cappuccino.rf_wraddr≠0)`. In both cases, the assertions are trivially true ($A \rightarrow A$) and there are no other valid and meaningful assertions. Thus, a bug in the design due to missing constraints is reflected in translation results that only have trivially true assertions.

*3) Overly Restrictive Assertions:* The third example shows the case that some malicious or buggy code are explicitly added in the design. For the AES assertion A29-02 from the AES11 design, Transys successfully translate it to the AES18 design: `aes_sbox.a != aes_sbox.d`. This assertion states the security property that the S-box should avoid any fixed points. We then maliciously modify the S-box design in AES18 such that when the input to the S-box is `8'hff`, it should output `8'h16` but instead outputs `8'hff`. We then run Transys to translate this assertion again and we get the new assertion: `(aes_sbox.a[7]` $\neq$ `aes_sbox.d[7])` $\rightarrow$ `(aes_sbox.a` $\neq$ `aes_sbox.d)`. This new assertion is valid for the buggy design. With the additional antecedent, hardware experts can easily identify the bug and the condition to trigger it.Thus, a malicious bug in the design can manifest itself in the translated assertions (typically as additional antecedents).

## VIII. Related Work

**Property driven hardware security.** There has lately been a call for "property driven hardware security" [28], [29], [30] that advocates building security specifications into the hardware design workflow, automating the process of doing so, and developing quantifiable measures of security. We see Transys as a contribution in response to this call.

**Developing security specifications.** A body of work on the use of execution monitors in processor designs has produced a set of security properties for various open source designs [31], [10], [32], [5]. These properties were developed manually. Subsequent work showed how to partially automate the process [7], and tackled temporal properties [33], but still required an initial set of manually written properties for each design under consideration. With Transys, the work done to specify properties for one design can be leveraged to bootstrap property generation for a second design.

**Extracting assertions from hardware designs.** Considering properties beyond those critical to security, there is a body of work on specification mining from hardware designs. The Iodine tool looks for possible instances of known design patterns, such as one-hot encoding or mutual exclusion between signals, and creates assertions that encode the found patterns [34]. More recent papers use data mining of simulation traces to extract more detailed assertions [35], [36] or temporal properties [37]. While these techniques are not concerned with finding security properties, they provide lessons on how to scale assertion extraction effectively.

**Assertion based verification of hardware designs.** The properties developed by Transys can be encoded as assertions and added to the design under review, at which point standard assertion based verification (ABV) techniques can be used to find property violations [38]. These techniques include simulation-based testing [39] and formal static analysis [40], [41], and are implemented in both commercial [42] and open source tools [43]. Software-style symbolic execution has also proven to be effective at finding property violations in hardware designs [44], [6].

**Language based verification.** A body of work has emerged on developing new or extending current hardware description languages for hardware verification. One language based approach uses typed hardware description languages, which can enforce security policies by construction [16], [45], [14], [15]. A second language based approach uses a formally defined language to first specify a policy and then refine the specification to a provably correct design [46], [47], [48].

**Tracking information flow in hardware.** Information Flow Tracking logic can be added at the gate level [17] or register transfer level [49] of a hardware design, and can capture timing flows [18], [19] or data flows [50]. While there is a trade-off to be made between precision and performance [51], [52], these techniques can demonstrate whether sensitive inputs to a design, e.g., the key material input to a cryptographic core, is directly or indirectly visible in the output signals. As with language based verification, this approach can provide strong guarantees, but also requires modifying the original design, either by adding tracking logic or, as in the case of CPUs, redesigning from the ground up to provide provable isolation between software contexts [53], [54].

**Software code clone detection.** Our Variable Mapping Pass is inspired by research in software code clone detection. The techniques used are token-based [55], [56], [57], semantic-based [58], [59], [60], [61], graph-based [62], [63], [64], and tracelet-based [65] approaches. Genius [63] uses features extracted from control flow graphs and converted to high-level numeric feature vectors to conduct searches. The approach is scalable and robust to code variation. Gemini [64] uses a graph-based deep learning approach and achieves high accuracy and high speed. Our approach combines graph and semantic-based features and adapts them to RTL code.

## IX. Conclusion

In this work, we advocate building security properties for new designs by leveraging existing properties. We present Transys, an automated tool that translates given security assertions from one hardware design to another in three passes—variable mapping, structural transformation and constraint refinement. Transys is able to translate 27 temporal logic assertions and 11 information flow tracking assertions across 38 AES designs, 3 RSA designs, and 5 RISC processor designs. The overall translation success rate is 96%. Among them, the translations of 23 (64%) assertions achieve semantic equivalence rates of above 60%. The average translation time per assertion is about 70 seconds.


## Acknowledgments

We would like to thank our shepherd, Dr. Yan Shoshitaishvili, and the anonymous reviewers for their helpful and insightful feedback. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1816637. Any opinions, findings, conclusions, and recommendations expressed in this paper are solely those of the authors.

## REFERENCES


[1] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[3] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018.

[4] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre Returns! Speculation Attacks using the Return Stack Buffer," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. Baltimore, MD: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/koruyeh

[5] M. Hicks, C. Sturton, S. T. King, and J. M. Smith, "SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 517–529. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694366

[6] R. Zhang, C. Deutschbein, P. Huang, and C. Sturton, "End-to-end automated exploit generation for validating the security of processor designs," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2018.

[7] R. Zhang, N. Stanley, C. Griggs, A. Chi, and C. Sturton, "Identifying security critical properties for the dynamic verification of a processor," in *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[8] C. Wolf, "Yosys open synthesis suite," http://www.clifford.at/yosys/.

[9] C. N. Coelho and H. D. Foster, *Assertion-Based Verification*. Boston, MA: Springer US, 2004, pp. 167–204. [Online]. Available: https://doi.org/10.1007/1-4020-2530-0_5

[10] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin, "Security Checkers: Detecting processor malicious inclusions at runtime," in *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*, June 2011, pp. 34–39.

[11] M. T. Harry Foster, Kenneth Larsen, "Introduction to the new accellera open verification library," 2006.

[12] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *J. Comput. Secur.*, vol. 18, no. 6, pp. 1157–1210, Sep. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1891823.1891830

[13] L. V. Nguyen, J. Kapinski, X. Jin, J. V. Deshmukh, and T. T. Johnson, "Hyperproperties of real-valued signals," in *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*. ACM, 2017, pp. 104–113.

[14] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A Hardware Design Language for Timing-Sensitive Information-Flow Security," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 503–516. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694372

[15] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh, "Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 555–568. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037739

[16] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: A Hardware Description Language for Secure Information Flow," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 109–120. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993512

[17] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA: ACM, 2009, pp. 109–120. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508258

[18] A. Ardeshiricham, W. Hu, and R. Kastner, "Clepsydra: Modeling timing flows in hardware designs," in *International Conference on Computer-Aided Design (ICCAD)*. IEEE/ACM, Nov 2017, pp. 147–154.

[19] B. Mao, W. Hu, A. Althoff, J. Matai, J. Oberg, D. Mu, T. Sherwood, and R. Kastner, "Quantifying timing-based information flow in cryptographic hardware," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 2015, pp. 552–559.

[20] W. Hu, A. Ardeshiricham, M. S. Gobulukoglu, X. Wang, and R. Kastner, "Property specific information flow analysis for hardware security verification," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. ACM, 2018, pp. 89:1–89:8.

[21] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[22] R. Szeliski, *Image Alignment and Stitching*. Boston, MA: Springer US, 2006, pp. 273–292. [Online]. Available: https://doi.org/10.1007/0-387-28831-7_17

[23] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 175–186. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568286

[24] L. R. Dice, "Measures of the amount of ecologic association between species," *Ecology*, vol. 26, no. 3, pp. 297–302, 1945. [Online]. Available: http://www.jstor.org/stable/1932409

[25] H. Salmani, M. Tehranipoor, and R. Karri, "On design vulnerability analysis and trust benchmarks development," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, 2013, pp. 471–474.

[26] B. Shakya, T. He, H. Salmani, D. Forte, S. Bhunia, and M. Tehranipoor, "Benchmarking of hardware trojans and maliciously affected circuits," *Journal of Hardware and Systems Security*, pp. 85–102, 2017. [Online]. Available: https://doi.org/10.1007/s41635-017-0001-6

[27] D. Lampret, "OpenRISC 1200 IP core specification," 2001. [Online]. Available: http://www.isy.liu.se/en/edu/kurs/TSEA44/OpenRISC/or1200_spec.pdf

[28] W. Hu, A. Althoff, A. Ardeshiricham, and R. Kastner, "Towards property driven hardware security," in *Microprocessor and SOC Test and Verification (MTV), 2016 17th International Workshop on*. IEEE, 2016, pp. 51–56.

[29] W. Hu, A. Ardeshiricham, and R. Kastner, "Identifying and measuring security critical path for uncovering circuit vulnerabilities," in *International Workshop on Microprocessor and SOC Test and Verification (MTV)*, Dec 2017, pp. 62–67.

[30] R. Kastner, W. Hu, and A. Althoff, "Quantifying hardware security using joint information flow analysis," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*. IEEE, 2016, pp. 1523–1528.

[31] M. Abramovici and P. Bradley, "Integrated Circuit Security: New Threats and Solutions," in *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*, ser. CSIIRW '09. New York, NY, USA: ACM, 2009, pp. 55:1–55:3. [Online]. Available: http://doi.acm.org/10.1145/1558607.1558671

[32] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin, "Evaluating security requirements in a general-purpose processor by combining assertion checkers with code coverage," in *Hardware-Oriented Security and Trust (HOST), 2012 IEEE International Symposium on*. IEEE, 2012, pp. 49–54.

[33] C. Deutschbein and C. Sturton, "Mining security critical linear temporal logic specifications for processors," in *Proceedings of the International Workshop on Microprocessor and SoC Test, Security, and Verification (MTV)*. IEEE, 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8746060

[34] S. Hangal, S. Narayanan, N. Chandra, and S. Chakravorty, "IODINE: A tool to automatically infer dynamic invariants for hardware designs," in *Proceedings of 42nd Design Automation Conference*. IEEE, 2005.

[35] P.-H. Chang and L. C. Wang, "Automatic assertion extraction via sequential data mining of simulation traces," in *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*. IEEE, 2010, pp. 607–612.

[36] S. Hertz, D. Sheridan, and S. Vasudevan, "Mining hardware assertions with guidance from static analysis," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 32, no. 6, pp. 952–965, 2013.

[37] W. Li, A. Forin, and S. A. Seshia, "Scalable specification mining for verification and diagnosis," in *Proceedings of the 47th Design Automation Conference*, ser. DAC. ACM, 2010, pp. 755–760. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837466

[38] H. Foster, *Applied Assertion-Based Verification: An Industry Perspective*, ser. Foundations and Trends(r) in Electronic Design Automation. Now Publishers, 2009. [Online]. Available: https://books.google.com/books?id=hL6d2t6Oh4EC

[39] L.-T. Wang, Y.-W. Chang, and K.-T. Cheng, *Electronic Design Automation: Synthesis, Verification, and Test*. Morgan Kaufmann, 2009.

[40] D. Brand, "Verification of Large Synthesized Designs," in *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD-93)*. IEEE, 1993.

[41] D.Lin, E.Singh, C.Barrett, and S.Mitra, "A structured approach to post-silicon validation and debug using symbolic dquick error detection," in *Proceedings of the IEEE International Test Conference*, 2015.

[42] "Cadence Verification Suite." [Online]. Available: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification.html

[43] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verfication Tool," in *Comuter Aided Verification (CAV)*. Lecture Notes in Computer Science, 2010.

[44] R. Zhang and C. Sturton, "A recursive strategy for symbolic execution to find exploits in hardware designs," in *Proceedings of the International Workshop on Formal Methods and Security (FMS)*. ACM, 2018.

[45] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, "Sapper: A Language for Hardware-level Security Policy Enforcement," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 97–112. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541947

[46] M. Vijayaraghavan, A. Chlipala, Arvind, and N. Dave, "Modular deductive verification of multiprocessor hardware designs," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, D. Kroening and C. S. Păsăreanu, Eds. Springer International Publishing, 2015, vol. 9207, pp. 109–127. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-21668-3_7

[47] T. Braibant and A. Chlipala, "Formal verification of hardware synthesis," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, N. Sharygina and H. Veith, Eds. Springer Berlin Heidelberg, 2013, vol. 8044, pp. 213–228. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39799-8_14

[48] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, "Kami: A platform for high-level parametric hardware specification and its modular verification," in *Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming (ICFP'17)*, 2017.

[49] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register transfer level information flow tracking for provably secure hardware design," in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*. European Design and Automation Association, 2017, pp. 1695–1700.

[50] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner, "Information flow isolation in i2c and usb," in *Proceedings of the 48th Design Automation Conference*. ACM, 2011, pp. 254–259.

[51] A. Becker, W. Hu, Y. Tai, P. Brisk, R. Kastner, and P. Ienne, "Arbitrary precision and complexity tradeoffs for gate-level information flow tracking," in *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*. IEEE, 2017, pp. 1–6.

[52] W. Hu, A. Becker, A. Ardeshiricham, Y. Tai, P. Ienne, D. Mu, and R. Kastner, "Imprecise security: quality and complexity tradeoffs for hardware information flow tracking," in *Computer-Aided Design (ICCAD), 2016 IEEE/ACM International Conference on*. IEEE, 2016, pp. 1–8.

[53] M. Tiwari, X. Li, H. M. Wassel, F. T. Chong, and T. Sherwood, "Execution leases: A hardware-supported mechanism for enforcing strong non-interference," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2009, pp. 493–504.

[54] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood, "Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3. ACM, 2011, pp. 189–200.

[55] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, July 2002.

[56] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176–192, March 2006.

[57] W. M. Khoo, A. Mycroft, and R. Anderson, "Rendezvous: A Search Engine for Binary Code," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 329–338. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487147

[58] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue, "MUDABlue: an automatic categorization system for open source repositories," in *11th Asia-Pacific Software Engineering Conference*, Nov 2004, pp. 184–193.

[59] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14. New York, NY, USA: ACM, 2014, pp. 406–415. [Online]. Available: http://doi.acm.org/10.1145/2664243.2664269

[60] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 709–724.

[61] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Detecting similar software applications," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 364–374. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337267

[62] H. Flake, "Structural comparison of executable objects," in *In Proceedings of the IEEE Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2004, pp. 161–173.

[63] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 480–491. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978370

[64] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 363–376. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134018

[65] Y. David and E. Yahav, "Tracelet-based code search in executables," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 349–360. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594343