Mining Security Critical Linear Temporal Logic Specifications for Processors

Calvin Deutschbein, Cynthia Sturton
The University of North Carolina at Chapel Hill
{cd, csturton}@cs.unc.edu

***Abstract*—This paper presents UNDINE, a tool to automatically generate security critical Linear Temporal Logic (LTL) properties of processor architectures. UNDINE handles complex templates, such as those involving four or more variables, register equality to a constant, and terms written over register slices. We introduce the notion of event types, which allows us to reduce the complexity of the search for a given template. We build a library of nine typed property templates that capture the patterns that are common to security critical properties for RISC processors. We evaluate the performance and efficacy of UNDINE and our library of typed templates on the OR1200, Mor1kx, and RISC-V processors.**

I. Introduction

A recent analysis of seven years of published AMD errata found that a significant fraction (9%) of the errata posed security vulnerabilities; moreover, each of the 60 processors studied was affected by at least one security vulnerability [1]. Eliminating these bugs is a challenge. Hardware companies invest heavily in testing and verifying their designs, but these techniques work by finding violations of specified properties. If secure behavior has not been specified, insecure behavior will not be noticed.

Specification mining automates the process of developing a set of properties from a given hardware design. The current state of the art focuses on finding properties that fit known patterns common across hardware designs, such as one-hot encoding or alternating bits [2], [3], [4]. While existing tools can produce tens of thousands of properties for a single design, they do not produce a security specification.

Prior work in generating security specifications for processors has relied on human expertise to manually develop a set of security properties [5], [1], [6]. More recently, SCIFinder used statistical learning to label a mined property as security critical or not [7]; however, the technique produced only non-temporal properties and relied on human expertise to produce the initial training set of properties. What is missing is a library of the patterns security critical properties exhibit, analogous to the one-hot encoding or alternating bit patterns used in functional specification mining. Given such a library, a new design can be mined to generate a set of security properties with little human intervention.

We present UNDINE, a tool for mining security specifications of processor designs. The specifications take the form of linear temporal logic formulas and capture properties that are critical to the security of the processor.

We introduce the notion of event types and we use these to find the patterns that are common to the manually and semi-automatically generated security properties of prior work. Using our event types, we build a library of typed property templates and develop a specification miner for use with the typed templates.

To generate the security critical properties of interest, our miner must be able to handle complex templates, such as those involving four or more variables, register equality to a constant, and terms written over register slices. To the best of our knowledge no existing miner provides all of these features. We build our miner on top of the Texada specification mining tool [8]. We modify Texada to accept typing information from traces and to reason effectively about register slices. We add pre- and post-processing steps to provide the needed typing information to traces of execution, apply filters to reduce the complexity of the search, and compose related properties to generate a concise set of properties that lend themselves to simple English descriptions and are critical to security.

We demonstrate the use of our typed property templates and type-aware specification miner by mining security specifications of three open source RISC processors: OR1200, Mor1kx, and RISC-V. Using our library of typed templates, we are able to automatically mine 25 of the 28 known security critical properties on OR1200. UNDINE also finds new security critical properties that require temporal logic to express. We provide an example exploit for one such property we mine on Mor1kx.

II. Tool

A. Overview

Figure 1 provides an overview of the UNDINE workflow. In a preliminary step, the processor design is simulated to generate traces of execution. Traces provide input to UNDINE, which works in three steps: preprocessing, mining, and postprocessing. During preprocessing UNDINE converts the traces of execution to traces of typed events and then applies a filter. During mining UNDINE takes filtered, typed event traces and a typed property template, and produces a set of security critical properties. During postprocessing UNDINE synthesizes properties to produce a manageable set of properties that can be understood by the user and are critical to the security of the processor.

We will use the following security property as a motivating example while describing this process:

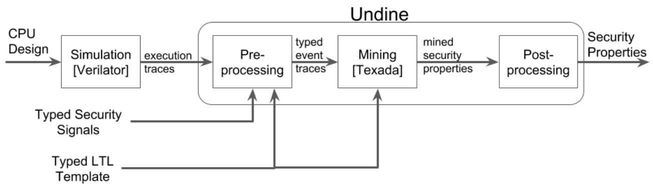

Fig. 1. An overview UNDINE, which uses a modified version of the Texada specification miner.

```
assert property
((~((ex_insn & 'hFFFF0000) >> 16 == 8192))
 || (id_flushpipe == 1));
```

This property was developed manually in prior work [1] and states that when a syscall instruction is being executed the instruction pipeline should be flushed at the instruction decode phase. It is critical to security because a syscall causes a change in privilege level, and the instruction following a syscall in the pipeline, which will not be part of the system call, should not be executed at the elevated privilege level.

B. Simulation

A *trace of execution* is produced by simulating the register transfer level (RTL) specification of the design under consideration. The value of each signal in the RTL model is logged at every clock cycle. More formally, a trace T is an ordered sequence of time-stamped, signal–value pairs:

$$\begin{aligned} T =& [(q,x)_t, (r,y)_t, (s,z)_t, \ldots, \\ & (q,x')_{t+1}, (r,y')_{t+1}, (s,z')_{t+1}, \ldots, \\ & (q,x'')_{t+2}, (r,y'')_{t+2}, (s,z'')_{t+2}, \ldots], \end{aligned}$$

where q, r, s represent state-holding signals in the design, and x, y, z represent numeric values. Tick-marks indicate the passage of a single clock cycle: if x represents the value of register q at time t, x' represents the value of register q at time $t+1$. We will use this notation throughout the paper. Where context makes the meaning clear, we will sometimes overload terms and use q, r, s to mean both the register and its value.

Each signal–value pair in the trace is an *event*. A standard value change dump (VCD) file as produced by many simulators suffices as a trace of execution. An excerpt from the VCD file produced by simulating the OR1200 processor is shown in Listing 1.

C. Typed Events

Central to the design of UNDINE is the notion of a *typed event*. We define five event types.

- register–register (RR): $(q_1 == q_2)$. Two registers, q_1 and q_2, have the same value.

```
or1200_ctrl.ex_insn == 1234
or1200_ctrl.id_flushpipe == 1
.. // this denotes a change in time
or1200_ctrl.ex_insn == 4321 // etc.
```

Listing 1. Excerpt of a trace of execution from the OR1200 processor.

$$\begin{aligned} LTL &\doteq \mathbf{G}(\phi) \\ \phi &\doteq \phi \rightarrow \phi \mid \neg\phi \mid \phi \mid \phi \wedge \phi \mid \phi \vee \phi \\ &\quad \mid \phi \ \mathbf{U}\ \phi \mid \mathbf{X}\ \phi \mid e \\ e &\doteq \mathtt{RR} \mid \mathtt{DR} \mid \mathtt{RV} \mid \mathtt{SV} \mid \mathtt{BV} \end{aligned}$$

Fig. 2. The grammar of typed LTL properties. The temporal operators ($\mathbf{G}, \mathbf{U}, \mathbf{X}$) have the standard definitions of Globally, Until, neXt. The event types (RR, DR, RV, SV, BV) are as defined in section II-C.

- delta–register (DR): $(q' == q + y)$. Register q changes by some value y in the next clock cycle.
- register–value (RV): $(q == y)$. Register q has value y.
- slice–value (SV): $(q[i:j] == y)$. A slice of register q has value y.
- bit–value (BV): $q[i:i+1] == y$. The i^{th} bit of register q has value y.

Returning to the example, a slice of the register `ex_insn` is compared to a value, in this case using a bit mask. (This comparison checks whether the instruction in the execute phase of the pipeline is a syscall.) This is expressible as an equivalent slice–value event. The `id_flushpipe` is an example of a register–value event. The original property can be restated using our typed events.

```
assert property
 (~ex_insn[31:16] == 8192 // SV event
  || id_flushpipe == 1);   // RV event
```

Event types inform the specification mining in two ways: 1) Registers in the design are associated with a particular type and will only appear in events of the correct type; 2) Property templates are written in terms of typed events and only property instances with the correct typing will match a given template.

D. Grammar of LTL Properties

UNDINE mines for properties by looking for possible instantiations of a given template. It is not limited to a predefined set of property templates, but rather takes the template as an input from the user. A user is free to create their own template or choose one from the library of templates we developed. Figure 2 defines the language of properties expressible in UNDINE.

E. Preprocessing

The preprocessor takes as input a set of execution traces and produces a filtered set of typed event traces ready for specification mining. There are two tunable parameters to the preprocessor that determine the trace transformation:

- typing information for signals in the design, and
- register slice size.

In prior work, Zhang et al. found that there is a subset of registers in the design that are associated with properties critical to security [7]. We dub these the *security-critical registers*. In UNDINE we extend this idea further. We note that a security-critical register will be used in a property in a predictable way. For example, the `id_flushpipe` register from our running example is used in a `RV` event. In the security-critical properties developed in prior work [7], [1], [5] the `id_flushpipe` register appears only in events of type `RV`. The first parameter to the preprocessor is the typing information for the security-critical registers in the design.

Hardware designs often use bit packing, for example, storing 32 individual control bits in a single 32-bit register. Another design tactic is to encode semantic information in a slice of a register, as when the highest-order 16 bits of the 32-bit `ex_insn` register determine whether the instruction is a syscall instruction. Security properties are often concerned with the control and semantic information available at the sub-register level. To enable this, the second parameter to the preprocessor is the register slice size: the preprocessor will break every register into its component slices of the given size. As we discuss in Section II-H, register slicing also reduces the time complexity of property mining for any given template.

The trace of execution is converted to a filtered trace of typed events as follows. At each clock cycle in the trace, each register is split according to the register slice size parameter. Each register is then labeled with its type as given in the signal typing parameter. Next, a set of *derived events* are added to each clock cycle in the trace. The derived events are calculated as follows. For every event type that appears in the property template, for every register of the appropriate type, the set of possible derived events is added to the trace. The execution trace from Listing 1 might look like the following after slicing and typing information have been applied.

```
ex_insn[15:0] == 1234  // SV
ex_insn[31:16] == 0    // SV
id_flushpipe == 1      // RV
```

Finally, at each clock cycle, registers in the execution trace that are of a type that does not correspond to any of the event types in the property template under consideration are removed from the trace.

F. Mining

After preprocessing, the filtered typed event traces and the property template are pased to the specification miner.

UNDINE is built on top of the Texada LTL Specifications Miner [8]. Texada takes in a trace of events and a property template and produces all property instantiations of the given template that are true of the event trace. We modify Texada to handle typed events and typed LTL property templates. With event types, potential properties that would otherwise match the property template, but have a type mismatch can be discarded early. The event types provide an effective filter at both the preprocessing and the mining stage.

Two additional modifications to Texada include discarding registers with uninitialized values and adding support to recognize and effectively handle sliced registers.

G. Postprocessing

The postprocessing step combines and simplifies properties to produce a more manageable set of final properties. First, all properties that contain an implication are sorted by antecedent. Properties that have the same antecedent are grouped into a new property in which all the consequents are ANDed together. Second, properties containing an implication are sorted by consequent and all properties with the same consequent are grouped into a new property in which all the antecedents are ORed together. Finally, the properties are simplified using the Z3 SMT solver [9].

H. Complexity

As with most specification miners the time complexity of UNDINE is exponential in the number of unique terms in the property template under consideration [10]. Its complexity is given by e^T, where e is the number of unique events in the set of traces being mined and T is the number of events in the template.

Register slice size affects the run time of UNDINE in two ways. If the slice size is aligned with semantic and control components of a register the templates required to capture the desired security properties tend to be simpler. If, on the other hand, the slice size is too big or too small the number of unique events in the template (T) grows. On the other hand, smaller slices always reduces the number of unique events in a trace (e).

III. EVALUATION

A. Property Templates

We developed a library of nine typed LTL templates that describe the patterns common to security critical properties for open source, RISC, pipelined processors. These are described in Table I. The first eight templates in the library come from studying security critical properties developed, either manually or semi-automatically, in prior work [7], [5], [1]. The ninth template comes from our own study of the processor design specifications.

Columns three and four of Table I list how many properties each template produced when mining the OR1200 processor (Section III-F provides details on our evaluation set up). Column five lists how many of the known security critical properties of prior work ([7], [5], [1]) were found by each template. In total our templates find 25 of the 28 security critical properties of prior work. The 3 properties not found require a bit shift that is determined dynamically, which is not supported by the UNDINE grammar.

Template 9 exercises the **U** (until) LTL operator and is necessary for finding properties that ensure the processor is

TABLE I
TYPED PROPERTY TEMPLATES

ID	Typed Template	Mined Properties	Postproc'd Properties	Known Properties
1	$\mathbf{G}(\mathtt{RR}_a)$	2	2	1
2	$\mathbf{G}(\mathtt{SV}_a \rightarrow \neg\mathtt{RR}_b)$	46843	32	2
3	$\mathbf{G}(\mathtt{SV}_a \rightarrow \mathtt{SV}_b)$	8134	376	2
4	$\mathbf{G}(\mathtt{SV}_a \mid \neg\mathtt{SV}_b)$	5794	431	1
5	$\mathbf{G}((\mathtt{SV}_a \ \& \ \mathtt{SV}_b) \rightarrow \mathtt{RR}_c)$	1026262	19	14
6	$\mathbf{G}(\mathtt{SV}_a \rightarrow \mathtt{DR}_b)$	13088	4	1
7	$\mathbf{G}((\mathtt{SV}_a \ \& \ \mathtt{SV}_b) \rightarrow \mathtt{BV}_c)$	204138	3	1
8	$\mathbf{G}(\mathtt{SV}_a \rightarrow (\mathtt{SV}_b \mid \mathtt{RR}_c))$	525322	648	1
9	$\mathtt{RR}\ \mathbf{U}\ \mathbf{G}(\mathtt{BV})$	134	134	0

initialized correctly. We discuss this template further in the next section.

B. Mining with Temporal Templates

Often, properties are defined for the processor starting at the first clock cycle after reset. These assume that processor state is initialized correctly; if it is not, security may be compromised without violating any property. Specifying the sequence of events required for secure initialization requires temporal operators, something not handled by prior work on defining security properties.

Using template (9) ($\mathtt{RR}\ \mathbf{U}\ \mathbf{G}(\mathtt{BV})$), we mine properties on the mor1kx processor and find seven groups of registers that must be equal to each other until the initialization period has ended (until $\neg$`reset`). These properties are listed in Table II. The first six properties describe registers that are free to change their values after reset; the seventh property describes registers that must always be equal and could therefore have been captured with the simpler $\mathbf{G}(\mathtt{BV})$ template. In Section III-E we use property (5) from Table II as a case study and examine how a violation of the property can lead to an exploitable security vulnerability.

C. Performance with Security Optimizations

We examine the performance benefits of introducing typed events. Table III compares mining time for each template for each of three versions of UNDINE: Naive, Security Signal, and Typed. The Naive implementation uses traces containing events from all registers, allows any signal to be associated with an event of any type, and does not include type checking in the miner. The Security Signal implementation uses traces with only registers associated with security critical properties, allows any signal to be associated with an event of any type, and does not include type checking in the miner. The Typed implementation uses traces with only registers associated with security critical properties, uses only events in which signals have the right type, and includes type checking in the miner. In all except the most trivial cases, mining is prohibitively expensive without any typing.

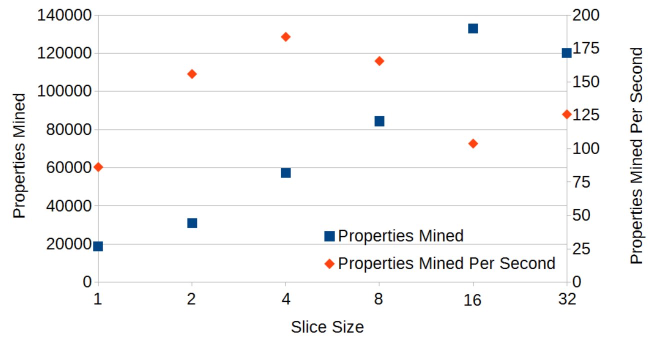

Fig. 3. Slice size affects the mining rate and quantity of output properties.

D. Performance with Slicing

The register slice size is a parameter to the preprocessor and is adjustable by the user. Smaller slice sizes lead to fewer possible unique events for a given trace, giving a performance boost to the miner. However, changing the slice size in either direction can affect the number of property instantiations for any given template as well as the rate at which properties can be mined. We explore this trade-off in Figure 3 using example template ($\mathtt{BV}\ \mathbf{U}\ \mathbf{G}(\mathtt{SV} \rightarrow \mathtt{RR})$.

E. Example Exploit

In the mor1kx processor, the exception, output, and basic status registers must be equal until initialization completes. Property (5) (Table II) captures this requirement. The lowest order bit of the basic status register indicates whether the processor is in supervisor mode or not. After initialization the basic status register holds the current status unless an exception has occurred, in which case the status register is saved to the exception status register. We add a bug to the control module of the mor1kx processor that violates this property. The bug changes the initial value of the exception status register and modifies the exception status register update to update all bits except the lowest order bit. As soon as an exception occurs, the correct value of the supervisor mode bit is lost. We exploit this bug to put the register in supervisor mode while executing user level code after an exception.

F. Number of Properties

We evaluate UNDINE on three open source RISC processors: mor1kx, OR1200, and RISC-V. Using template (5) $\mathbf{G}((\mathtt{SV}\ \&\ \mathtt{SV}) \rightarrow \mathtt{RR}$ we mine each of the three processors until a stable set of properties is reached.

On OR1200 and mor1kx the execution traces were a series of assembly instructions chosen from all legal instructions that could be compiled to run in simulation, a linux boot, the built-in test suites of the system-on-a-chip, and a hello world C program (for a system call) compiled to run on bare-metal. On RISC-V, it was three benchmarks: quicksort, towers, vector-vector-add and a hello world C program (for a system call) compiled to run on bare-metal. Figure 4 shows how the set of properties converges to a steady state as the trace length

TABLE II
PROPERTIES MINED USING TEMPLATE (9) ON MOR1KX

Prop ID	Registers equal until reset	Description
1	ctrl_lsu_adr_o[31:0], dbus_dat_o[31:0], du_dat_o[31:0], mfspr_dat_o[31:0], pc_decode_to_execute[31:0], pc_fetch_to_decode[31:0], spr_bus_dat_i[31:0], spr_bus_dat_o[31:0]	Many address and data fields
2	decode_rfa_adr_o[4:0], decode_rfb_adr_o[4:0], decode_rfd_adr_o[4:0],	Decode stage register file address registers
3	fetch_rfa_adr_o[4:0], fetch_rfb_adr_o[4:0], wb_rfd_adr_o[4:0]	Fetch stage register file address registers
4	ctrl_rfd_adr_o[4:0], execute_rfd_adr_o[4:0], fetch_rfa_adr_o[4:0], wb_rfd_adr_o[4:0]	Other register file address registers
5	spr_esr[15:0], spr_sr[15:0], spr_sr_o[15:0]	Status registers and exception status register
6	ctrl_epcr_o[31:0], pc_execute_to_ctrl[31:0]	Program counter and exception program counter
7	du_dat_i[31:0], snoop_adr_i[31:0]	Debug ports to databus (globally true)

ID	Typed Template	Naive Implementation	Security Signals	Typed Templates
1	$\mathbf{G}(\texttt{RR}_a)$	t/o	6.823	0.600
2	$\mathbf{G}(\texttt{SV}_a \rightarrow \neg\texttt{RR}_b)$	t/o	38.682	1.777
3	$\mathbf{G}(\texttt{SV}_a \rightarrow \texttt{SV}_b)$	t/o	122.186	7.826
4	$\mathbf{G}(\texttt{SV}_a \mid \neg\texttt{SV}_b)$	t/o	71.694	6.856
5	$\mathbf{G}((\texttt{SV}_a \ \& \ \texttt{SV}_b) \rightarrow \texttt{RR}_c)$	t/o	t/o	217.988
6	$\mathbf{G}(\texttt{SV}_a \rightarrow \texttt{DR}_b)$	t/o	122.186	18.445
7	$\mathbf{G}((\texttt{SV}_a \ \& \ \texttt{SV}_b) \rightarrow \texttt{BV}_c)$	t/o	t/o	515.168
8	$\mathbf{G}(\texttt{SV}_a \rightarrow (\texttt{SV}_b \mid \texttt{RR}_c))$	t/o	t/o	1521.896
9	$\texttt{RR} \ \mathbf{U} \ \mathbf{G}(\texttt{BV})$	t/o	27.995	0.987

TABLE III
TIME IN SECONDS TO MINE TYPED TEMPLATES BY IMPLEMENTATION (T/O INDICATES TIMEOUT AT 4 HOURS)

TABLE IV
NUMBER OF PROPERTIES AFTER MINING AND AFTER APPLYING TWO POSTPROCESSING STEPS USING TEMPLATE (5).

	Mining	Consequent	Antecedent
OR1200	597838	234096	22
mor1kx	755530	135378	26
RISCV	278960	104370	8

and number of traces increase on the RISC-V processor. The OR1200 and mor1kx processors exhibited similar trends.

The figures show the number of properties produced without postprocessing. Table IV shows how postprocessing reduces the final number of properties produced by UNDINE.

Runs to steady state finish in under 15 minutes. Table V shows the time it takes UNDINE to complete a steady state run for a given architecture, broken out into different stages.

TABLE V
TIME IN SECONDS FOR EACH OF PREPROCESSING, MINING, AND POSTPROCESSING FOR STEADY-STATE RUNS BY ARCHITECTURE

	Preprocessing	Mining	Postprocessing
OR1200	34.24	171.36	7.87
mor1kx	1.00	105.79	14.58
RISCV	173.20	842.13	3.26

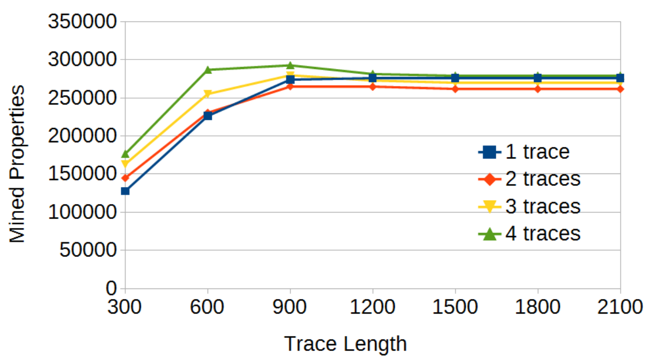

Fig. 4. Mining on RISCV converges to a steady set of properties as trace length and number of traces increase.

IV. RELATED WORK

Generating Security Critical Properties for Hardware Designs: SCIFinder semi-automatically generates security critical properties for a hardware design [7]. Whereas that tool produces properties expressible in propositional logic, our tool can produce both propositional and temporal properties expressed in linear temporal logic.

Extracting Functional Properties from Hardware Designs: General properties that do not focus on security have been found by looking for instances of known patterns, such as one-hot encoded signals [2], [11]. An alternative approach is to use data mining to infer properties from traces of execution of the design [3], [4], [12]. More recent work has focused on mining temporal properties from execution traces [13], [14], [15], [16]. A combination of static and dynamic analysis can extract properties expressed over words [17].

Mining Specifications for Software: The seminal work in specification mining comes from the software domain [18] in which execution traces are examined to infer temporal specifications in the form of regular expressions. Subsequent work used both static and dynamic traces to filter out less useful candidate specifications [19]. More recent work has tackled the challenges posed by having imperfect execution traces [20], and by the complexity of the search space [21], [22], [10]. Daikon, which produces invariants rather than temporal properties, learns properties that express desired semantics of a program [23].

Generating Security Properties of Software: In contrast to the hardware domain, in the software domain a number of papers have developed security specific specification mining tools. These tools use human specified rules [24], observe instances of deviant behavior [25], [26], [23], or identify instances of known bugs [27].

V. Conclusion

In this paper we proposed an assertion mining tool, Undine, which automatically mines securty critical LTL properties from RTL specifications of processors. The tool produces manageable numbers of properties which if violated by a processor leave possible vulnerabilities over which exploits can be readily demonstrated. Undine runs in minutes, is usable across different architectures and can be easily parameterized if necessary. We also propose a typing system for processor events for security critical LTL properties.

Acknowledgments

We would like to thank Rui Zhang for insightful comments as we developed this work, and the anonymous reviewers for their helpful feedback. This material is based upon work supported by the National Science Foundation under Grants No. CNS-1651276 and CNS-1816637, by the Semiconductor Research Corporation, Intel, and by a Google Faculty Research Award. Any opinions, findings, conclusions, and recommendations expressed in this paper are solely those of the authors.

References

[1] 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 *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM, 2015.

[2] 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.

[3] 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.

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

[5] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin, "Security checkers: Detecting processor malicious inclusions at runtime," in *IEEE Int'l Symposium on Hardware-Oriented Security and Trust (HOST)*, 2011.

[6] ——, "Evaluating security requirements in a general-purpose processor by combining assertion checkers with code coverage," in *IEEE Int'l Symposium on Hardware-Oriented Security and Trust (HOST)*, 2012.

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

[8] C. Lemieux, D. Park, and I. Beschastnikh, "General LTL specification mining," in *Int'l Conf. on Automated Software Engineering*. IEEE, 2015.

[9] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008.

[10] M. Gabel and Z. Su, "Symbolic mining of temporal specifications," in *Int'l Conf. on Software Engineering*. ACM, 2008.

[11] E. E. Mandouh and A. G. Wassal, "Automatic generation of hardware design properties from simulation traces," in *International Symposium on Circuits and Systems*. IEEE, 2012, pp. 2317–2320.

[12] L. Liu, D. Sheridan, V. Athavale, and S. Vasudevan, "Automatic generation of assertions from system level design using data mining," in *Int'l Conf. on Formal Methods and Models for Codesign*. ACM/IEEE, 2011.

[13] W. Li, A. Forin, and S. A. Seshia, "Scalable specification mining for verification and diagnosis," in *Design Automation Conference*. ACM, 2010.

[14] A. Danese, N. Dalla Riva, and G. Pravadelli, "A-team: Automatic template-based assertion miner," in *Design Automation Conference*. IEEE, 2017.

[15] A. Danese, T. Ghasempouri, and G. Pravadelli, "Automatic extraction of assertions from execution traces of behavioural models," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*. IEEE, 2015, pp. 67–72.

[16] A. Danese, F. Filini, T. Ghasempouri, and G. Pravadelli, "Automatic generation and qualification of assertions on control signals: A time window-based approach," in *IFIP/IEEE International Conference on Very Large Scale Integration-System on a Chip*. Springer, 2015, pp. 193–221.

[17] L. Liu, C. Lin, and S. Vasudevan, "Word level feature discovery to enhance quality of assertion mining," in *International Conference on Computer-Aided Design (ICCAD)*. IEEE/ACM, 2012, pp. 210–217.

[18] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," *ACM Sigplan Notices*, vol. 37, no. 1, pp. 4–16, 2002.

[19] W. Weimer and G. C. Necula, "Mining temporal specifications for error detection," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2005, pp. 461–476.

[20] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: mining temporal API rules from imperfect traces," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 282–291.

[21] M. Gabel and Z. Su, "Javert: fully automatic mining of general temporal properties from dynamic traces," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 339–349.

[22] G. Reger, H. Barringer, and D. Rydeheard, "A pattern-based approach to parametric specification mining," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2013, pp. 658–663.

[23] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 35–45, Dec. 2007. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2007.01.015

[24] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou, "AutoISES: Automatically inferring security specifications and detecting violations," in *Proceedings of the 17th Conference on Security Symposium*, ser. SS'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 379–394. [Online]. Available: http://dl.acm.org/citation.cfm?id=1496711.1496737

[25] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically patching errors in deployed software," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 87–102. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629585

[26] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim, "Cross-checking semantic correctness: The case of finding file system bugs," in *Symposium on Operating Systems Principles*. ACM, 2015.

[27] F. Yamaguchi, F. Lindner, and K. Rieck, "Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning," in *Conference on Offensive Technologies (WOOT)*. USENIX, 2011.