Temporal Tracing of On-Chip Signals using Timeprints

Rehab Massoud*, Hoang M. Le*, Peter Chini+, Prakash Saivasan+, Roland Meyer+ and Rolf Drechsler*,†
*University of Bremen, +Technical University of Braunschweig, †DFKI GmbH, Germany

ABSTRACT

This paper introduces a new method to trace **cycle-accurately** the temporal behavior of on-chip signals while operating in-field. Current cycle-accurate schemes incur unacceptable amounts of data for logging, storage and processing.

Our key idea to enable efficient yet cycle-accurate tracing, is to bring *timing* to the front as a main traced artifact. We split the signal tracing into consecutive (back-to-back) finite trace-cycles. Within a trace-cycle, a signal's value-change instance gets assigned an encoded timestamp. At the end of each trace-cycle, these encoded timestamps are aggregated into a logged *timeprint*, which summarizes the temporal behavior over the trace-cycle.

To retrieve the accurate timing, we reconstruct the exact instances from a timeprint via a SAT query. The experiments demonstrate how unprecedented lightweight tracing can be applied, and how timeprints enable the verification of cycle-accurate properties and the detection of sporadic temperature effects.

ACM Reference Format:
R. Massoud et al. 2019. Temporal Tracing of On-Chip Signals using Timeprints. In *The 56th Annual Design Automation Conference 2019 (DAC '19), June 2–6, 2019, Las Vegas, NV, USA.* USA, 6 pages. https://doi.org/10.1145/3316781.3317920

1 INTRODUCTION

In late May 2018, the National Transportation Safety Board (NTSB) issued the preliminary report about the first instance of an autonomous car killing a pedestrian [3]. The report included information about the timing of when object identifications happened. The timings provided were very coarse; and if the delay that caused the crash was dependent on signals exchanged between modules, the exact timing of the signals' firings would be crucial. Nowadays, reporting such events is carried out by the systems about themselves, as in space-telemetry or automotive-diagnostics and telematics. Accurate, independent and non-intrusive tracing of on-chip signals is needed to determine what happened exactly. This becomes even more vital for determining liability when signals are exchanged between systems provided by different suppliers.

In this paper, we target the scenario where a cycle-accurate post-mortem analysis is performed when an unexpected failure occurs during in-field **deployment**. On the cycle-accurate level, variant forms of *Runtime Verification* (RV) [9] and monitoring [10], can be utilized to check a subset of the **specified behaviors**. Specified behaviors are those defined at design-time by the system specifications. Unfortunately, in practice, there are **unspecified behaviors**, as the specifications are not always complete. Additionally, there are specified behaviors that cannot be formalized and synthesized as on-chip monitors (either due to limited expressiveness of monitoring logic or limited on-chip area). This is illustrated on the right side of Figure 1: during deployment, RV can only cover a subset of the specified behaviors and none of the unspecified behaviors.

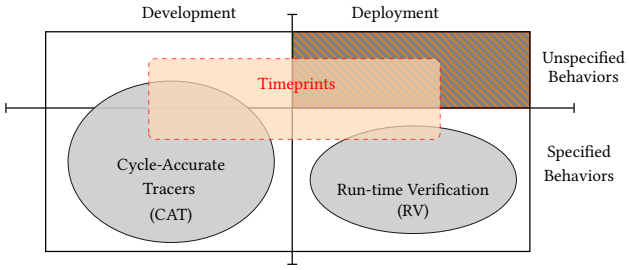

Figure 1: Cycle-Accurate Tracing Taxonomy

During the **development** phase, *cycle-accurate tracing* (CAT) is heavily applied in Real-Time (RT) embedded software domain using dedicated debuggers/tracers, such as [1, 4]. This category of tracers focuses on software operating on a processor or a micro-controller, where tracing depends on the existence of an instruction set and program counters that proceeds the execution. For an arbitrary on-chip signal, other solutions like logic and protocol analyzers and/or scan-chain-based methods for FPGA/ASIC Systems on Chip (SoC) such as [5, 17] exist. These provide different forms of functional and sometimes cycle-accurate tracing. Unspecified behaviors might be captured during *development* by such existing CAT (meaning the circle of CAT might reach to that partition), as shown on the left of Figure 1. However, for any generic on-chip signal this is limited to **development time**. During **deployment**, none of these CAT methods can provide continuous cycle-accurate tracing in-field, due to limited trace-buffer's area and/or notorious logging and storage requirements [23], which easily exceed several Gigabytes per second.

In summary, no existing approach can provide continuous cycle-accurate tracing that can cover **unspecified behaviors** during **deployment**. This is where the proposed method comes to serve. Our *timeprints* are designed to keep an independent, compressed, cycle-accurate temporal record/trace of what happened on chip. This is vital given our increasing dependence on digital systems, of which the executions are conducted on-chip without leaving a trace. Timeprints' very low bit-rate suffices for efficient logging, storage and processing during deployment. Hence, they have the potential to change how we think about in-field cycle-accurate tracing. Our contribution can be summarized as:

- a *timeprints*-based light-weight cycle-accurate tracing method, including a formulation and a reduction to SAT of the timeprint reconstruction problem, and
- an implementation of the approach using realistic experiments, showing how after-deployment cycle-accurate sporadic behavior tracing can be efficiently checked.

DAC '19, June 2–6, 2019, Las Vegas, NV, USA

ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317920

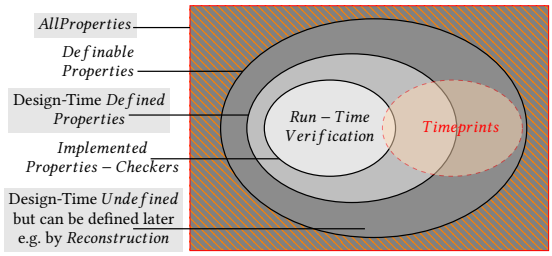

Figure 2: Cycle-Accurate Properties Classification

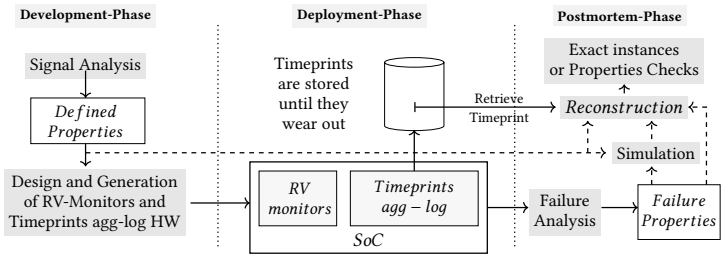

Figure 3: Timeprints Life Cycle

2 OVERVIEW

Timeprints enable efficient tracing by trying **not** to log the information –properties– we already know about the signal. What gets logged is something that can enable us to check *accurately* what happened when doubt is raised. So, we use the *properties* already known to be holding, to check the properties we are suspecting, or even to restore the exact occurrences when we have no clue at all. Figure 2 presents a clarifying classification of signal's properties from cycle-accurate perspective.

After the signal to be traced is identified, properties of that signal are used in the development phase, to decide about which monitors/checks shall be implemented. The *Signal-Analysis* label on the upper left side of Figure 3, shows this step at the *design-phase*. This would result in a subset of the describable (definable) properties, which can be formally defined down to the cycle-accurate level, as illustrated by the subsets in Figure 2. Among the defined-properties, some can be checked using hardware synthesized monitors, constituting another subset of the defined properties. Such monitors verify in run-time that certain properties hold about the traced signal [10], or reason about combinations/history of them as in [11]. They can also be used in WCET analysis as in [12], and can even correct/enforce the behavior if the property is violated during runtime [13]. Due to on-chip space limitation, only a subset of the synthesizable monitors gets implemented. The darkest area in the *Timeprints* set in Figure 2, represents the definable properties which were **undefined at design-time** on the cycle-accurate level, and timeprints can enable tracing them.

In our methodology flow, the *defined properties*, resulting from the signal-analysis step do not only result in the run-time verification monitors, but are also used to decide the size and encoding of the timestamps, as shall be seen later in Sections 3.3, 4.3 and 5.1. Like the RV monitors, the timeprints generator (timestamps aggregation and logging) is implemented in hardware, as in the middle of Figure 3. Light-weight timeprints are logged all the time during the execution, with constant rate and transmitted to some central database, see Figure 3.

The idea of timestamps aggregation was first introduced in [16], to compress the logged data into footprints. Instructions footprints-aided analysis was used in [18]. Periodic logging [24], analysis-based message-selection [8], and selective data capture and compaction [25], were all utilized to overcome the unacceptable accurate data logging requirements. These are all development-phase methods, that require on-chip trace buffers and/or many reruns.

When the system is running, hardware monitors (if used) will be checking the implemented defined properties satisfaction. Then, when an un-expected failure happens, or any doubts about certain point of operation in the past is raised, the respective timeprints are consulted in a *postmortem-phase*, as in th right if Figure 3.

To reconstruct the exact instances that can lead to the timeprint at hand, we encode the possible timestamps' aggregations and the timeprint into an All-SAT problem. We then utilize the verified defined-properties about the execution in the SAT problem encoding to prune the huge search space. The properties already known to hold because the hardware monitors checking them indicate their satisfaction, can be encoded into the SAT-solver input. For example: missing deadlines, as a defined property, would be captured by RV; while smaller delays that might indicate a security threat [14], can skip such RV check. This RV check can still be used to prune the SAT-problem search space; then the reconstructed instances would show the smaller delays. Dashed lines in Figure 3 indicate optional usage in the reconstruction step. Failure properties obtained from analyzing the failure that took place can also be encoded. This renders the reconstruction effort acceptable, and hence the whole method very practical.

Going back from a latest traced state, or check-point, using the formal description of the system, was used in [19]. But this formal reconstruction is neither time-accurate nor sporadic behavior preserving; because it requires several reruns. Formal reconstruction of scenarios that can lead to an existing failed state was also introduced in [6]; where a candidate simulation is provided as a possible failure scenario, but without any guarantee relating this scenario to what actually took place on-chip.

A timeprints' trace enables reconstructing the exact instances, in a way that can show non-monitored delays, and/or give an evidence of a specific failure scenario or threat hypothesis. By encoding properties, the reconstruction process can directly check, potential *simulation* scenarios for plausibility or likelihood, as in the top-right of Figure 3. Timeprints can also in many cases prove or disprove exact properties about specific execution traces. This makes them serve as a potential witness, which can be consulted when needed, for transparent liability assignment. Timeprints are not meant to replace existing methods, rather they are inspected to reason about the details of cycle-accurate behavior when needed.

In the next section, we clarify by example the timeprints generation and reconstruction steps.

3 DIDACTIC EXAMPLE

Consider the scenario of autonomous car crash; where the obstacle was identified, the car slowed down but still hit the obstacle. We assume that every module exactly behaved according to the logged diagnostic information, which narrowed the suspicions down to the exact timing of sending signal S_t, sent by chip C_1 to chip C_2.

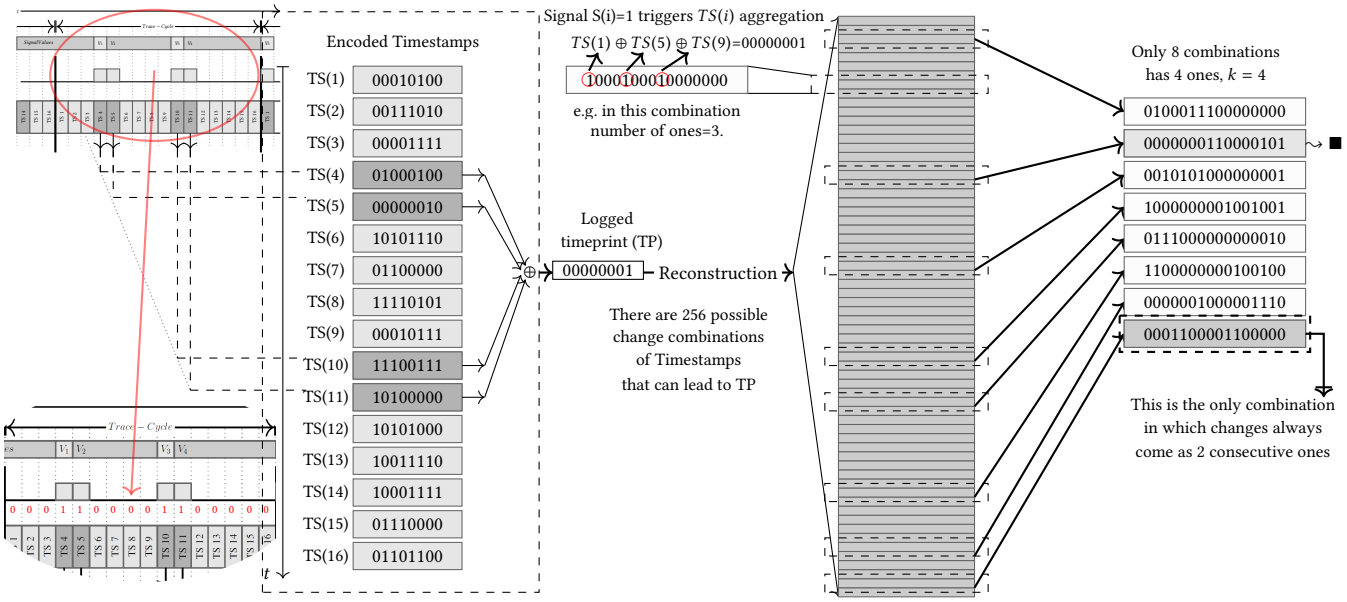

Figure 4: Signal Values Changes, Corresponding Timestamps Aggregation and Reconstruction

Our methodology starts with having a designated discrete signal S_t to trace. We split the tracing task of S_t into *trace-cycles* that contain m clock-cycles each. In our example, $m = 16$. In Figure 4, on the left-hand side, a trace-cycle j is depicted, where four changes in S_t's value $V_1, \ldots, V_4$ took place. If S_t was sent by C_1 *before* a specified deadline, that lies at the d^{th} clock-cycle in the trace-cycle j, then C_2 is responsible for the overall system's delayed response. But if S_t was sent *after* the deadline, C_1 is responsible.

A common way to record the behavior of a signal like S_t, is to log the precise timings. This means we log a number with $\log(m)$ bits each time the value of S_t changes. In Figure 4, we have 4 changes. Hence, we log $4 \cdot 4 = 16$ bits. In general, when the number of changes is denoted by $k \in \mathbb{N}$, we need to log $k \cdot \log(m)$ bits each trace-cycle. Thus, the amount of logged information depends linearly on k. As k varies from one trace-cycle to another, the amount of logged bits also does. This makes processing or searching through logged information difficult. Using one pin for logging, during a trace-cycle of length m, the maximum number of bits that can be logged each trace-cycle is m. If we need $\log(m)$ bits for each change, we can record at most $m/\log(m)$ of them.

3.1 Timeprint Logging

For the method, we identify clock-cycles by encoded timestamps. These are bitvectors of a fixed dimension (or size) b, where $m \geq b \geq \log(m)$. In Figure 4, we identify the i-th clock-cycle by a timestamp $TS(i)$. We use $b = 8$ bits for the 16 timestamps.

Initially, we use a b-bit hold vector, with all bits set to 0. When a change in S_t occurs, a corresponding *change-signal* $S(i)$ goes to 1, causing the corresponding timestamp to be aggregated –here added– to the currently hold vector. At the end of the trace-cycle, this results in a b-bit vector TP, called the timeprint. Since the vector is a summary of the traced signal's behavior, our method logs TP. In Figure 4, we aggregate/add the timestamps $TS(4)$, $TS(5)$, $TS(10)$, and $TS(11)$. The resulting timeprint TP is the sum of these timestamps: $TP = (0, 0, 0, 0, 0, 0, 0, 1)$. TP is shown in the middle of Figure 4, as an 8-bit vector. Note that modulo 2 addition is just bitwise XOR.

Bitwise XOR function can be implemented easily on hardware and is bit-width preserving. When we use the XOR as an aggregation function, the number of changes is lost. So, we record the precise number of changes that happened during a trace-cycle in a counter k that is increased each time a change shows up. Since there are at most m changes, we can encode k into $\log(m)$ bits. We show later, that k plays a central role in the reconstruction problem. During a trace-cycle, our method always logs $b + \log(m)$ bits in total: b for the timeprint TP, $\log(m)$ for the counter k. This enforces a constant number of logged bits each trace-cycle, irrespective of k.

3.2 Reconstruction Problem

In a trace-cycle, the only information logged by our method are: the timeprint and the number of changes in the traced signal. This may create ambiguity: there might be different change-signals leading to the same timeprint. Finding these is called *reconstruction problem.*

For the aggregated timeprint $(0, 0, 0, 0, 0, 0, 0, 1)$ in Figure 4, there are more solutions besides the actual *change-signal* (we'll call it for simplicity *signal*): $TS(4) + TS(5) + TS(10) + TS(11)$, for example, $TS(1) + TS(5) + TS(9)$, illustrated at the top middle of Figure 4. In total, there are 256 signals whose timestamps sum up to TP. The number of reconstructions (SAT-query solutions) increases with the decrease in the timeprint size. Until now we have not yet taken into account all of the logged data. Since we also know that the actual signal has exactly k changes, we can exclude those violating this requirement. This leads to a formal problem description:
"Given a timeprint TP and a number k of changes, find all signals with k changes, whose timestamps add up to TP."

For our example, limiting the number of changes to 4 decreases the number of candidate signals from 256 to only 8. A list of these is shown on the right-hand side of Figure 4. It is worth noting that the reconstruction problem crucially depends on the chosen

timestamps. Linear independent timestamps would cause no ambiguity but the timeprint width would then be m-bit which is too big. On the otherhand, too compressed timestamps would increase the degree of ambiguity.

3.3 Temporal Properties

To isolate the actual signal in the reconstruction problem, we utilize information obtained from the *defined and verified* properties.

Back to our original scenario, where the traced signal S_t was sent from chip C_1 to chip C_2, we know that these are timings of writes of new S_t values; so if we know that such writes always last for one cycle (by some verified specifications) after which S_t goes to zero and remains there until a new value of S_t is written, we can conclude that all changes in the signal (represented by ones), has to occur in two consecutive clock cycles; and hence the last shaded row is actually the one that happened. In our example, it was not an accident that we had at the end one possibility and we became sure of what accurately took place. We utilized the property (of 2 consecutive value's changes), to choose a set of timestamps that would, in most cases, result in a unique reconstruction for each couple of pairs (4.3 and 5.1.2 discuss timestamps encoding). However, sometimes we cannot exclude all the non-actual traces. For example, if one more cycle delay was allowed, the second possibility (2^{nd} row, marked with ■) would have been a candidate.

Often, we do not need to find the actual signal, rather it is enough to check whether all signals that might lead to the logged timeprint, satisfies or breaks a certain safety property. Consider the case of meeting a deadline. Assume that the deadline is given by $i = 8$. In Figure 4, we can see that all 8 possible reconstructed signals have a 1-bit already before the 8-th position. This means that all traces involving 4 changes that aggregate to TP meet the deadline, no matter which one actually took place. So, it may happen that the information given by the temporal property does not suffice to isolate a single signal reconstruction (i.e. a single cycle-accurate instances). But often, we only want to know whether there is a trace that satisfies or breaks a certain temporal property.

4 FORMULATION AND ENCODING

We formalize the logging procedure, prove it to be a sound abstraction, and present a solution for the reconstruction problem. The underlying domain is $\mathbb{F}_2$, the field consisting of 0, 1 with addition and multiplication modulo 2. Let $n \in \mathbb{N}$. We write $\mathbb{F}_2^n$ for the n-dimensional vector space over $\mathbb{F}_2$. This is the set of n-bit vectors where addition is bitwise XOR.

We trace signals over *trace-cycles* of *length* m, with $m \in \mathbb{N}$. These are periods containing m clock-cycles. Let such a trace-cycle be fixed. Formally, a *signal* is a map $S : [1..m] \to \{0, 1\}$, where $S(i) = 1$ indicates a change of the traced-signal in the i-th clock-cycle. By Sig, we denote the set of all such signals.

Fix $b \in \mathbb{N}$ with $m \geq b \geq \log(m)$. We define the logging relative to an *encoding*. An encoding is an injective map $TS : [1..m] \to \mathbb{F}_2^b$, which assigns each clock-cycle a unique *timestamp*. The logging procedure abstracts a signal S to a pair (TP, k), where $TP \in \mathbb{F}_2^b$ is called *timeprint* and k is the number of changes in S. Such a pair is called a *log entry*. The set of all possible log entries is $Log = \mathbb{F}_2^b \times [1..m]$.

Formally, for a fixed encoding TS, the logging procedure implements the function $\tilde{\alpha}_{TS} : Sig \to Log$. Given a signal S, the function returns a log entry $\tilde{\alpha}_{TS}(S) = (TP, k)$ with $TP = \sum_{i:S(i)=1} TS(i)$ and $k = |\{i \mid S(i) = 1\}|$. Note that the timeprint TP is the sum over those timestamps where the traced signal changes. Moreover, there are exactly k changes in S.

4.1 Soundness

We show that the logging procedure constitutes a sound abstraction of signals. To this end, we establish a Galois insertion between the domain of signals and the domain of log entries.

Formally, the domain we abstract from is the powerset lattice $\mathcal{P}(Sig)$. The abstract domain is the powerset lattice over log entries, $\mathcal{P}(Log)$. Fix an encoding $TS : [1..m] \to \mathbb{F}_2^b$. For the abstraction function, we apply the logging procedure to sets of signals. Let $\alpha_{TS} : \mathcal{P}(Sig) \to \mathcal{P}(Log)$ be the lifting of $\tilde{\alpha}_{TS}$, defined by $\alpha_{TS}(F) = \bigcup_{S \in F} \{\tilde{\alpha}_{TS}(S)\}$, where $F \in \mathcal{P}(Sig)$.

For the concretization $\gamma_{TS} : \mathcal{P}(Log) \to \mathcal{P}(Sig)$, we first define an auxiliary function $\tilde{\gamma}_{TS} : Log \to \mathcal{P}(Sig)$. It maps a log entry (TP, k) to its preimage under $\tilde{\alpha}_{TS}$, the set $\{S \in Sig \mid \tilde{\alpha}_{TS}(S) = (TP, k)\}$. Then γ_{TS} is the lifting of $\tilde{\gamma}_{TS}$ to sets of log entries, defined as above.

Both functions, α_{TS} and γ_{TS} are monotonic by definition: We have that $\alpha_{TS}(F) \subseteq \alpha_{TS}(G)$ for $F \subseteq G$ sets of signals, and similar for γ_{TS}. Moreover, the functions establish a Galois insertion as shown in the next lemma. The proof follows from the above definitions.

Lemma 1. *Let TS be an encoding. For each $F \in \mathcal{P}(Sig)$ we have $F \subseteq \gamma_{TS}(\alpha_{TS}(F))$. Moreover, for each $V \in \mathcal{P}(Log)$ we have the equality $V = \alpha_{TS}(\gamma_{TS}(V))$.*

4.2 Signal Reconstruction

Since the logging procedure is an abstraction, there might be different signals that result in the same log entry. Finding all signals that get abstracted to a particular entry is what we refer to as the signal reconstruction problem. We give a formal definition of the problem and an idea of an efficient SAT-based solution.

Let $(TP, k) \in Log$ be the output of the logging procedure. Reconstructing all signals S that get abstracted to (TP, k) is the task of computing the preimage of (TP, k) under the map $\tilde{\alpha}_{TS}$. We define the *Signal Reconstruction* problem as follows:

Signal Reconstruction (SR)	
Input:	Encoding $TS : [1..m] \to \mathbb{F}_2^b$, timeprint $TP \in \mathbb{F}_2^b$, $k \in \mathbb{N}$.
Task:	Find all signals S with $\tilde{\alpha}_{TS}(S) = (TP, k)$.

We can state an equivalent form of SR in terms of linear algebra. Let $A = [TS(1) \mid \cdots \mid TS(m)] \in \mathbb{F}_2^{b \times m}$ be the matrix consisting of all timestamps. SR is equivalent to finding all solutions $x \in \mathbb{F}_2^m$ of the system $Ax = TP$, where x has exactly k entries set to 1. Each solution x represents a signal S with $\tilde{\alpha}_{TS}(S) = (TP, k)$ and vice versa.

Variants of this problem are well-known in coding theory [22]. Moreover, SR is known to be NP-hard [7] and it is therefore unlikely that it can be solved in polynomial time.

To solve SR, we suggest a SAT-encoding. Usually, the input to a SAT solver is a formula in CNF. However, we use an extension which also allows clauses of XORed variables. Those are particularly helpful when encoding linear equations. A solver supporting this input format is *Cryptominisat* [21].

For the encoding, we introduce m variables $x_1, \ldots, x_m$. These are the bits of a solution x in the system $Ax = TP$. Intuitively, setting x_i to 1 amounts to a signal that has a change in the i-th clock cycle. This means that timestamp $TS(i)$ is *chosen* and summed

up to arrive at TP. The linear equations $Ax = TP$ are modeled by b clauses $C_1, \ldots, C_b$. A variable x_i occurs in C_j if the j-th bit of $TS(i)$ is 1. All the variables in C_j are XORed. If the j-th bit of TP is 0, the clause C_j gets negated, a feature supported by *Cryptominisat*. Hence, the clause represents exactly the j-th equation of $Ax = TP$.

What is left to encode is the cardinality constraint over the variables. We have to choose exactly k out of the m variables and set them to 1. A naive encoding would use $\binom{m}{k+1} + \binom{m}{m-k+1}$ clauses, resulting in an intractable SAT query. We use the efficient and compact cardinality encoding proposed in [20], which introduces $O(m \cdot k)$ additional variables and needs only $O(m \cdot k)$ clauses to express the constraint.

4.3 Choice of Timestamps

The choice of the timestamps has influence on the ambiguity occurring within the logging procedure and thus on the time needed to solve SR. Intuitively, a sparse choice of timestamps allows only for few possibilities to sum up to the timeprint. It decreases the number of solutions of $Ax = TP$, making it easier to find all of them. However, we can only allow sparsity up to a certain extend as the number of logged bits would grow.

Ideally, we would choose a timestamp encoding that avoids ambiguity at all. This can be achieved by constructing an encoding $TS : [1..m] \rightarrow \mathbb{F}_2^b$, where $TS(1), \ldots, TS(m)$ are linearly independent vectors. Then, the system $Ax = TP$ has a unique solution and SR can be solved quite fast. For example, a *one-hot encoding* would be of this type. However, choosing m linearly independent vectors requires that the dimension of $\mathbb{F}_2^b$ is m, hence $b = m$. But then, the number of bits we need to log depends linearly on m, contradicting our goal to establish a space-efficient logging procedure.

We can achieve a trade-off in the choice of timestamps by requiring linear independence only up to a depth d. That means each subset of timestamps $T \subseteq TS([1..m])$ of size d is linearly independent. As d grows, the number of solutions to SR decreases, but the number of logged bits b increases. Currently, we fix $d = 4$ and approximate TS and b using a practical heuristic. Computing an encoding with the smallest possible b is an open problem for future research.

5 APPLICATION

We first discuss timeprint design parameters, and how they affect the reconstruction time. Then two different experiments, showing how timeprints can be used, are presented.

5.1 Timeprint Design Parameters

5.1.1 Trace-cycle length m. The choice of m affects directly the amount of logging. When b is the bit-width of the timeprint, the bit rate required for logging is: $(b + log(m))/m$ multiplied by the maximum clock-rate. Increasing m would decrease the log-rate, but the average number of changes k, would increase, increasing the number of solutions, and the reconstruction time. Each row at Table 1 (to the left) represents the reconstruction time for certain trace-cycle length m and number of changes k. At the end of each row, the log-rate R required for a signal of 100 MHz is given.

5.1.2 Timestamp Encoding. In Random-constrained timestamp generation: timestamps are generated randomly, while checking for Linear Independence of depth 4 (LI-4). We use generic heuristics, which starts by smallest possible timestamp fulfilling LI-4; then we increment and check that the condition still holds. This encoding leads to smaller b and less average reconstruction times than the random-constrained encoding, see at the right Table 2.

5.1.3 Encoding Temporal Properties. As mentioned earlier, the additional encoding of temporal properties of a signal into SR helps pruning the search space and reduces the SAT solving time. When the property is a violation (satisfaction) of some safety property, then we only require one SAT-solution (UNSAT) to prove (disprove) that a violation has (not) happened. For illustration we consider two simple properties, $P2$: 2 consecutive timestamps would appear at least once, and Dk: at least k changes happens before deadline D.

Columns marked with c-SAT corresponds to solving time of SR (i.e. only using cardinality constraints c=k). The columns marked with $P2$ and Dk show the solving time under additional constraints imposed by $P2$ and Dk of $k = 3, D = 32$, respectively. For each columns pair, the first gives time until reaching the first candidate, while the second (c-SAT.10) gives time until the 10^{th} satisfying solution, or until no more solutions is found, if there are already less than 10. As can be seen, $P2$ is less efficient than Dk, because it is also a weaker property. Using $P2$ and Dk, together, reduces the solving time even more than Dk alone (column $c + Dk + P2$). Notice that, in Tables 1 and 2, we just used some specific timeprints, and that the time is affected by the timeprint itself, and not only the properties[1]. We can model properties defined in [15].

5.2 Experiments

5.2.1 CAN Bus Communication. The exact timing of the actual messages exchange on the bus in Controller Area Network (CAN), used in automotive for inter-modules communications, is vital in determining the actual transmission time that took place in the real-world. A sample of CAN messages log (as usually reported by the software) is shown here, where timestamps are at the left:

```
2.257008s GearBoxInfo(1020)d 1 01             size  -> 58
2.253552s EngineData(100)d 8 00 00 19 00 00 00 00 00-> 125
2.256312s ABSdata(201)   d 6 00 00 00 00 00 00      -> 105
2.260804s Ignition_Info(103) d 2 01 00              -> 67
...
```

If we name these messages as $m_1, m_2, \ldots m_4$, then m_1 would appear on the CAN bus as (where ones corresponds to the bus's recessive state, and zeros to the bus's dominant state):
00111111110000000100000001000000010110000110111111111111
The CAN bus idle state is 1, hence m_1's first bit 0 is the start bit, and then comes the ID (1020=01111111100), then the data size, ... etc[2]. To log timeprints during the in-field message exchange on a 5 Mbps bus, a trace-cycle length of 1000 clock-cycles, and timestamps width of 24, were chosen. This means 5 timeprints and their respective number of change k were logged every second, i.e. 170 bps (5*(24+10)). Such available data from CAN messages logs can be encoded in our SAT-reduction. We built a tool, that directly takes CAN messages, and other temporal properties as input, and encodes the corresponding clauses to the SAT solver input.

In our experiment, two modules exchanging the message m_2 were involved in a late car response, and m_2's transmission time would determine who is responsible for the delay. The CAN messages listing above was from the transmitter. At the receiver, m_2, was received as:

[1]Results were obtained on an Intel i7-7500U CPU @ 2.70GHz x 4 with 15.6 GiB memory.
[2]Details are in the ISO-11898:2003 Standard. We ignore bit-stuffing here for simplicity.

m,k	b	c-SAT.1	c-SAT.10	c+P2.1	c+P2.10	c+Dk.1	c+Dk.10	c+Dk+P2.1	c+Dk+P2.10	R
64/3	b=13	0m0.010s	0m0.085s	0m0.026s	0m0.092s	0m0.023s	0m0.027s	00m0.049s	0m0.022s	20.97 MHz
64/4		0m0.048s	0m0.149s	0m0.014s	0m0.192s	0m0.023s	0m0.044s	0m0.032s	0m0.036s	
64/8		0m0.019s	0m0.101s	0m0.021s	0m0.175s	0m0.029s	0m0.222s	0m0.045s	0m0.221s	
64/32		0m0.032s	0m0.023s	0m0.013s	0m0.058s	0m0.013s	0m0.024s	0m0.041s	0m0.030s	
128/3	b=16	0m0.124s	0m0.709s	0m0.127s	0m0.817s	0m0.108s	0m0.170s	0m0.125s	0m0.131s	12.5 MHz
128/4		0m0.116s	0m0.610s	0m0.274s	0m0.928s	0m0.085s	0m0.223s	0m0.157s	0m0.313s	
128/8		0m0.118s	0m0.774s	0m1.070s	0m2.122s	0m0.156s	0m0.460s	0m0.259s	0m0.948s	
128/16		0m0.087s	0m0.149s	0m0.035s	0m0.228s	0m0.036s	0m0.261s	0m0.102s	0m0.174s	
512/3	b=22	0m1.714s	1m50.343s	0m29.912s	0m33.847s	0m1.479s	0m1.514s	0m0.427s	0m0.393s	4.3 MHz
512/4		0m42.638s	1m50.312s	1m3.351s	2m2.762s	0m1.563s	0m1.551s	0m1.901s	0m1.892s	
512/8		0m44.025s	1m57.296s	0m52.319s	2m46.793s	0m4.226s	0m14.590s	0m8.770s	0m21.824s	
1024/3	b=24	1m36.234s	18m29.931s	4m7.147s	4m5.050s	0m2.747s	0m2.624s	0m1.380s	0m1.687s	2.3 MHz
1024/4		3m42.684s	18m0.121s	10m28.096s	24m48.517s	0m4.567s	0m5.443s	0m23.424s	0m21.586s	
1024/8		3m33.685s	15m14.368s	7m37.488s	22m10.561s	0m8.891s	1m23.997s	0m4.949s	2m3.061s	

TS encoding:		Random-Constrained Timestamps			
m/k	b	c-SAT	c+P2	c+Dk	c+Dk+P2
512/3	b=31	2m57s	0m0.340s	4m39.727s	0m0.270s
512/4		33m17	13m33.423s	3m26.192s	1m51.532s
1024/3		11m39s	0m1.063s	22m26.209s	0m0.987s
TS encoding:		Incremental Timestamps			
m/k	b	c-SAT	c+P2	c+Dk	c+Dk+P2
512/3	22	0m1.714s	0m29.912s	0m1.479s	0m0.427s
512/4	22	0m42.638s	1m3.351s	0m1.563s	0m1.901s
1024/3	24	1m36.234s	4m7.147s	0m2.747s	0m1.380s

Table 1: Different m,k reconstruction time (to the left)
Table 2: Different timestamps encoding schemes (above)

```
2.253596s EngineData(100)d 8 00 00 19 00 00 00 00 00-> 125
```

The deadline was at the absolute timestamp 2.253580s. The logged timeprint, corresponding to the trace-cycle which started at 2.253400s was retrieved. The exact whole trace-cycle reconstruction took 38.279s, and showed the message started at the 823^{rd} clock-cycle (corresponds to 2,253,564.6us) and ended after the deadline at 2,-253,589.6us. Given the actual failure time window (from 2.253533s to253600s), reconstruction took only 3.082s. Encoding the property that this message transmission happened on the wire before the deadline in that window gave UNSAT in 1.597s.

Being that small enables simple and efficient logging and transmission hardware (hence no trace buffers are required) , and allows saving data of hours in few Gigabytes. We used CANoe CAN-analyzer Demo9 from Vector to generate a full scenario of exchanged CAN messages; over which we applied manual delays.

5.2.2 Temperature Compensated Refresh Effects Detection. In this experiment, to obtain cycle accurate log of memory access, we implemented a *timeprints-agg-log*, (aggregation/logging) hardware, and connected it to the address-signals of the AHB-AMBA bus. Timeprints were logged via a simplified USB-UART transmitter. Our *timeprints-agg-log* HW was, together with a LEON3 processor, implemented on a Nexys3 FPGA board. The whole system including the timeprints-generation was also simulated by Questa-Sim RTL cycle-accurate simulator.

Comparing the logs from HW to the simulated timeprints, obtained while running the same software image on both, we identified simulation configuration error, where the memory wait states were wrong in the SRAM model from the Gaisler simulation library [2]. After fixing this error, the number of changes k, in all trace-cycles became exactly the same (in simulation and log). However, the timeprints themselves started to differ after around 3000 clock-cycles (3 trace-cycles, as we chose $m = 1024$). Re-running several times, at different temperatures, the mismatch in timeprints started from as early as the third trace-cycle, to as late as the 28^{th}.

Starting from the trace provided by the simulation, we encoded it with the property that: one change instance is delayed by one-clock cycle. Doing this enabled us to identify the exact delay clock-cycle, each time a mismatch was found. From these exact instances, we noticed that, during the execution, this one clock-cycle delay happens earlier if temperature is higher. The data-sheet of the memory chip, mentions a temperature compensated refresh rate, but it does not specify accurately its dependence of the chip temperature, which increases on its own by the execution itself; i.e. it even differs for different instruction sequences being run. This illustrates directly, how properties undefined at design-time could be traced on the cycle-accurate level, using *timeprints*.

6 CONCLUSION

The paper presented a novel *Timeprints*-based tracing methodology, which provides cycle-accurate temporal traces of on-chip signals execution. Being of extreme light-weight, *Timeprints* open the door for the first time, to a wide range of deployment-phase accurate timing-properties verification. Being consistent and cycle-accurate, provides a reliable transparent evidence for the actual temporal behaviors taking place on-chip, while operating in-field.

ACKNOWLEDGMENT

This work is supported by the DAAD, University of Bremen (SyDe graduate school and CRDF) and the BMBF grant SELFIE (grant no. 01IW16001). We also thank Khaled Naga (Avelabs) for his insights on CAN bus experiment.

REFERENCES

[1] 2018. ARM CoreSight and ETM. http://www.arm.com.
[2] 2018. Gaisler Research. LEON3 synthesizable processor. http://www.gaisler.com.
[3] 2018. https://www.ntsb.gov/news/press-releases/Pages/NR20180524.aspx.
[4] 2018. System Navigator Probe. http://www.mips.com.
[5] 2018. www.xilinx.com/products/design-tools/chipscopepro.html.
[6] C. Ahlschlager and D. Wilkins. 2004. Using Magellan to Diagnose Post-Silicon Bugs, Synopsys Verification Avenue Technical Bulletin, vol.4, no.3.
[7] E. Berlekamp, R. McEliece, and H. van Tilborg. 1978. On the inherent intractability of certain coding problems. *IEEE Transactions on Information Theory* 24, 3 (1978).
[8] Debjit Pal et al. 2018. Application Level Hardware Tracing for Scaling Post-Silicon Debug. In *(DAC 2018)*.
[9] E. Bartocci et al. 2018. *Introduction to Runtime Verification.* Springer.
[10] E. Bartocci et al. 2018. *Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and Applications.* Lectures on RV, Springer.
[11] J. Schumann et al. 2015. R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems. In *Runtime Verification 2015; Vienna; Austria.*
[12] N. Decker et al. 2018. Online analysis of debug trace data for embedded systems. In *DATE, 2018.*
[13] S. Pinisetty et al. 2014. Runtime enforcement of timed properties revisited. *Formal Methods in System Design* (Dec 2014).
[14] M. Hamad, Z. Hammadeh, S. Saidi, V. Prevelakis, and R. Ernst. 2018. Prediction of Abnormal Temporal Behavior in Real-Time Systems. In *(SAC 2018).*
[15] B. Lisper and J. Nordlander. 2012. A Simple and Flexible Timing Constraint Logic. Berlin, Heidelberg.
[16] R. Massoud, J. Stoppe, D. Große, and R. Drechsler. 2017. Semi-formal Cycle-Accurate Temporal Execution Traces Reconstruction. In *FORMATS.*
[17] S Mitra, S. A. Seshia, and N. Nicolici. 2010. Post-Silicon Validation Opportunities, Challenges and Recent Advances. In *DAC.* ACM.
[18] S. B. Park, T. Hong, and S. Mitra. 2009. Post-silicon bug localization in processors using instruction footprint recording and analysis (IFRA). *TCADIC* (2009).
[19] F. M. De Paula. 2012. Backspace: Formal Analysis for Post-Silicon Debug Traces. In *Phd. Thesis, University of British Colombia.*
[20] C. Sinz. 2005. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In *CP (Lecture Notes in Computer Science)*, Vol. 3709. 827–831.
[21] M. Soos, K. Nohl, and C. Castelluccia. 2009. Extending SAT Solvers to Cryptographic Problems. In *SAT (Lecture Notes in Computer Science)*, Vol. 5584. Springer.
[22] A. Vardy. 1997. Algorithmic Complexity in Coding Theory and the Minimum Distance Problem. In *STOC.* ACM, 92–109.
[23] B. Vermeulen and K. Goossens. 2014. *Debugging Systems-on-Chip.* Springer, New York.
[24] JS. Yang and NA. Touba. 2008. Enhancing Silicon Debug via Periodic Monitoring. In *Proc. of Symposium on Defect and Fault Tolerance.*
[25] Joon-Sung Yang and Nur A. Touba. 2013. Improved Trace Buffer Observation via Selective Data Capture Using 2-D Compaction for Post-Silicon Debug. In *TVLSI.*