Secure Run-Time Hardware Trojan Detection Using Lightweight Analytical Models

Burin Amornpaisannon, *Student Member, IEEE*, Andreas Diavastos, *Member, IEEE*, Li-Shiuan Peh, *Fellow, IEEE*, and Trevor E. Carlson, *Senior Member, IEEE*

***Abstract*—Hardware Trojans, malicious components that attempt to prevent a chip from operating as expected, are carefully crafted to circumvent detection during the predeployment silicon design and verification stages. They are an emerging threat being investigated by academia, the military, and industry. Therefore, run-time hardware Trojan detection is critically needed as the final layer of defense during chip deployment, and in this work, we focus on hardware Trojans that target the processor's performance. Current state-of-the-art detectors watch hardware counters for anomalies using complex machine-learning models, which require a dedicated off-chip processor and must be trained extensively for each target processor. In this work, we propose a lightweight solution that uses data from a single reference run to accurately determine whether a Trojan is slowing processor performance, across CPU configurations, without the need for new profiles. To accomplish this, we use an analytical model based on the application's inherent microarchitecturally independent characteristics. Such models determine the expected microarchitectural events across different processor configurations without requiring reference values for each application-hardware configuration pair. By comparing predicted values to actual hardware events, one can quickly check for unexpected application slowdowns that are the key signatures of many hardware Trojans. The proposed methodology achieves a higher true positive rate (TPR) compared to prior works while having no false positives. The proposed detector incurs no run-time performance penalty and only adds a negligible power overhead of 0.005%.**

***Index Terms*—Analytical modeling, embedded security, hardware Trojan detection.**

I. Introduction

THE INTERNET of Things (IoT) era has intensified the competition in the semiconductor industry to design and produce smart computing systems that can handle diverse and demanding applications, such as self-driving cars, smart cities, and wearable devices. These applications require smart computing systems to perform complex computation with high performance and power efficiency, which increases the chip complexity. Chip design companies are required to meet tight time-to-market timelines to stay competitive, forcing them to rely on external suppliers to overcome these challenges. One of the challenges of developing modern hardware is to ensure that they are secure and reliable. However, this current design flow exposes new security risks that have to be addressed. For example, malicious actors could exploit vulnerabilities in different components of the chip. Therefore, it is essential to apply rigorous testing and verification methods to identify and mitigate these risks before deploying the system to the end users.

Due to the tight time-to-market timeline and chip complexity challenges mentioned, chip designers provide their designs to and use products from various parties, such as leveraging third-party intellectual property (IP) blocks, using sophisticated CAD software built by other companies, and transferring their designs to foundries for fabrication. The problem is that these third-party companies can be untrusted and potentially inject hardware Trojans into the target design. Hardware Trojans are malicious components that attempt to prevent a chip from operating as expected and can be inserted in a chip at any phase of the design flow, from early register transfer level (RTL) design stage to tape-out [1]. They are an emerging threat being investigated by academia, military, and industry [1], [2], [3] and can cause catastrophic changes in chip functionality, leading to performance degradation, denial of service (DoS), and information leakage [4].

Countermeasures [5], [6] have been developed against hardware Trojans targeting various components, such as network-on-chip [7], [8], processors [9], as well as accelerators [10]. These countermeasures have also been developed for different phases of the chip development flow from presilicon validation to post-silicon testing and run-time verification. For example, functional verification ensures that the output of the design under test is as expected. Side-channel analysis observes side-channel characteristics of a circuit such as power and delay [11] in search of anomalies caused by Trojans. However, hardware Trojans are designed to be stealthy, embedded as tiny components in the chip, and rarely activated to circumvent these predeployment techniques. Run-time hardware Trojan detection is thus critically needed to serve as the final layer of defense that protects the chip during deployment.

State-of-the-art processor-based run-time hardware Trojan detection methodologies [9], [12] tend to observe microarchitectural events, such as the number of branch misses on a processor, to detect anomalous behavior. Unlike side-channel information (e.g., power), which can be disturbed by physical phenomena such as process variation and chip aging that

Manuscript received 25 April 2023; revised 19 July 2023; accepted 23 August 2023. Date of publication 15 September 2023; date of current version 22 January 2024. This work Supported by Singapore NRF under Grant NRF2018NCR-NCR002. This article was recommended by Associate Editor R. S. Chakraborty. *(Corresponding author: Andreas Diavastos.)*

Burin Amornpaisannon is with the School of Computing, National University of Singapore, Singapore.

Andreas Diavastos, Li-Shiuan Peh, and Trevor E. Carlson are with the Department of Computer Science, National University of Singapore, Singapore (e-mail: diavastos@gmail.com).

Digital Object Identifier 10.1109/TCAD.2023.3316113

can decrease its detectability, microarchitectural events tend to be consistent across chips with the same design. However, these prior works require time-consuming training of complex machine-learning detection algorithms that assume a trusted baseline with no Trojans during training. In some cases, like when the design is provided as a third-party IP block, a trusted baseline may not be available. More importantly, current state-of-the-art detection models are configuration-specific and cannot be reused across different processor configurations. Furthermore, they often require an additional off-chip secure processor to monitor the target processor during execution.

This work employs, for the first time, mechanistic models based on analytical modeling to address these challenges. Mechanistic models aim to predict key microarchitectural events of a given hardware–software pair. This data can be used to detect abnormal behavior due to the effects of hardware Trojans [12]. The key benefits of mechanistic models are that they can capture the fingerprint of an application regardless of the underlying microarchitecture while flexibly and accurately predicting microarchitecture-specific events using only high-level hardware configuration information. In addition, they do not require costly offline training or model updates when the processor configuration changes.

In this work, we introduce a lightweight hardware Trojan detector. The detector is connected to the target processor and implements analytical models that capture fundamental software characteristics to predict key microarchitectural events periodically without a trusted baseline. The predicted values are then compared with actual values from the target processor to identify anomalies in processor behavior and detect hardware Trojans that target the processor's performance. To the best of our knowledge, this work is the first that uses analytical modeling to detect hardware Trojans. The proposed solution achieves the highest true positive detection rate compared to prior work, with no false positives, even for hard-to-detect, stealthy hardware Trojans. The proposed detection unit is off the critical path, works in parallel with the processor, and only increases the power consumption by a negligible 0.005%.

II. Related Work

Prior run-time hardware Trojan detection methodologies observe key system attributes that capture the behavior of the processor, for example, from side-channel information or microarchitectural events, to detect anomalies. We classify prior works into those based on hardware performance counters and those that rely on chip information.

Hardware Trojan Detection in Processors Based on Microarchitectural Events: State-of-the-art run-time hardware Trojan detection techniques for processors rely on observing logical or side-channel information and use machine-learning or deep-learning algorithms to detect anomalies in the data. Vijayan et al. [9] observed representative flip-flops in a processor. The experiment shows that observing 12 flip-flops in the Leon3 processor is sufficient to achieve 90.9% accuracy and 0.40% power overhead accounting only for the monitoring module. Elnaggar et al. [12] proposed an approach that detects changepoints and classifies them from performance counter data. Elnaggar et al. [13] relied on half-space trees to detect anomalies in data streams from performance counters. The works [12], [13] heavily suffer from the limited subset of observable performance counters which hurts detectability and can lead to a high false positive rate (FPR) or low true positive rate (TPR) when observing an unsuitable microarchitectural event.

Hardware Trojan Detection in Processors Based on Chip Information: Other previous works also observe and detect anomalies from side-channel information using, for example, power, current, electromagnetic radiation, and temperature [14], [15], [16], [17]. However, even chips with the same design can have different observations due to process variations, different physical environments, and chip aging, complicating detection processes. Also, tiny hardware Trojans may be able to circumvent this approach as they do not generate observable anomalies in side channel information even if the Trojans are catastrophic. These techniques also require sensors of sufficient precision to be inserted on-chip.

FinalFilter [18] is a reconfigurable property checker that aims to protect security-critical properties instead of performance-related properties. This work is orthogonal to ours, and the two can be combined to provide broader protection. Processor protection unit (PPU) [19] observes the instruction's opcode, the number of clock cycles to execute an instruction, and specific internal signals to detect hardware Trojans. Nevertheless, its implementation highly depends on simple processor designs with limited scalability. Its complexity grows when a processor becomes more complicated.

III. Background

Microarchitectural events, such as a count of branch and cache events that occur, have been shown to accurately capture the characteristics of an application [20] and can be viewed as the fingerprint of a program from which hardware Trojans can be detected when they disturb the program signature reflected by these events. This section describes the microarchitecture-independent analytical models used in this work to capture the fingerprint of an application based on the application itself. In the rest of this section, we describe the two main analytical modeling techniques used in this work: 1) branch predictor modeling and 2) cache modeling.

A. Branch Predictor Modeling

An analytical model that captures the branch behavior of an application using linear branch entropy [21] is used in this work. This model provides a way to predict the branch misprediction miss rate of an application running on a specific hardware platform in a lightweight and flexible way while maintaining accuracy. The model has also been used as a component of a more complex processor analytical model [22] that can estimate the performance of out-of-order processors. Building and using the branch predictor model consists of two phases: 1) the preparation phase and 2) the miss rate estimation phase. The preparation phase also consists of two subphases: 1) application profiling and 2) model training.

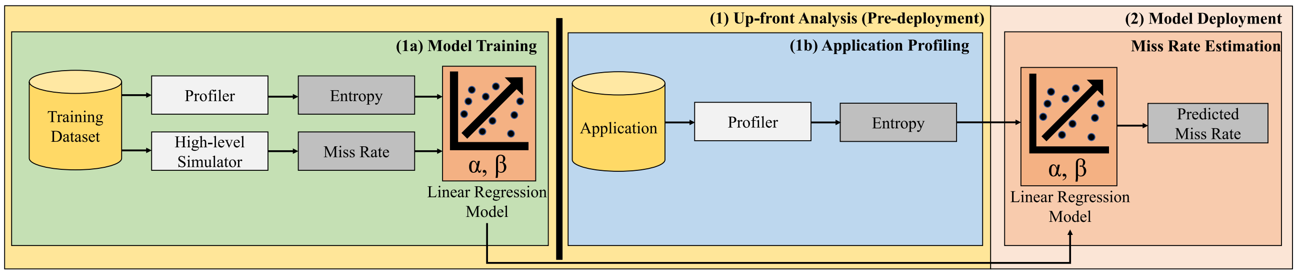


Fig. 1. Branch predictor modeling flow begins with (1) an up-front analysis phase to build an analytical model that can be used for predicting branch miss rates. The up-front analysis is composed of a (1a) model training and (1b) application profiling step. The model training step is used to generate linear branch entropy values and miss rates of the applications and train the linear regression model. During (1b) application profiling, the target application is profiled and a linear branch entropy profile is generated. During model deployment (2), the resulting linear regression model is used to predict branch miss rates based on the profile of the target application. This model is deployed and represents a lightweight model using two linear parameters of the specific branch predictor to accurately predict a branch prediction rate [21].

Fig. 1 demonstrates the entire flow of this framework

$$E_L(p(i,H)) = 2 \cdot \min(p(i,H), 1 - p(i,H)). \quad (1)$$

In the application profiling phase, where the application is run to collect microarchitectural event statistics, linear branch entropy $E_L(p(i,H))$ can be derived from a taken rate of a history pattern shown in (1). A taken rate of a specific local/global history pattern H, $p(i,H)$, is defined as a probability of a static branch i to be taken when the value of branch history equals H. The value of a linear branch entropy ranges from 0 to 1, where 0 is a predictable conditional branch, leading to a low miss rate, and 1 is the least predictable, leading to a high miss rate

$$E = \frac{1}{N} \sum_i \sum_H n(i,H) \cdot E_L(p(i,H)). \quad (2)$$

The average linear branch entropy, E, is then calculated using (2), using the weighted arithmetic mean, where $n(i,H)$ is the number of times the static branch i is executed and N is the total number of dynamic branches executed

$$M(E) = \alpha + \beta \cdot E. \quad (3)$$

The model training phase trains a linear regression model to predict branch miss rates. The branch miss rate of an application for a given branch predictor can be derived using (3). To construct this linear model, a training dataset for a particular branch predictor is needed, which contains the relationship between branch miss rate and linear branch entropy, which can be obtained from a high-level computer architecture simulator [23] or dynamic binary translation tool like Intel Pin [24]. α and β are then generated using a least-squares fit based on the training dataset.

During the miss rate estimation phase, the average linear branch entropy data from the target application is retrieved. The specific average linear branch entropy value is then chosen based on the type and size of the specific branch predictor to predict the miss rate. The predicted number of misses in a specific period can then be calculated by multiplying the predicted miss rate by the total number of branches during that period.

This branch predictor model provides a microarchitecture-independent branch profiling methodology that allows one to predict a branch miss rate without the need for a full processor design implementation or golden reference. The profile is also independent of a specific processor architecture as the branch statistics are only calculated solely based on the behavior of the program. Only the linear regression model is specific to a branch predictor, which can be trained using, for example, a high-level branch predictor simulator [25] without the need for its actual hardware implementation like RTL code or gate-level netlist.

B. Cache Modeling

To build an analytical model for caches, the least-recently used (LRU) stack processing algorithm [26] is used and was chosen as it can quickly predict the number of cache hits and misses analytically based on the LRU replacement policy as well as flexibly as it does not require reprofiling the same application for different processor configurations. The algorithm has been widely used in other works in different applications [27].

The LRU stack processing algorithm uses a stack to keep track of cache-line addresses accessing memory. The LRU stack distance of a cache-line address is the number of addresses between that cache-line address and the top of the stack. When a cache-line address is accessed, the counter for its particular LRU stack distance value is counted, and the cache-line address is moved up to the top of the stack to emulate the LRU replacement policy. By doing so, a histogram can be constructed from which one can calculate the number of cache hits and misses based on the size of a cache. This algorithm can be applied to instruction and data caches as they both work similarly on different types of data. This step is called the preparation phase.

Figs. 2 and 3 demonstrate how the LRU stack distance is computed and how to keep track of cache statistics. In this implementation, the buckets are associated to sizes 64, 128, 256, ..., 2^n B. The infinity bucket is used to count the number of addresses that have been first accessed and, thus, result in cold misses. In Fig. 2, when an address that has never been seen before is accessed, in this example address D, the address is placed on the top of the stack, and the infinity bucket is incremented by one. Similarly, in Fig. 3, when the address D is later accessed again, the address is moved to the top of the stack, and the distance between the last location and the top

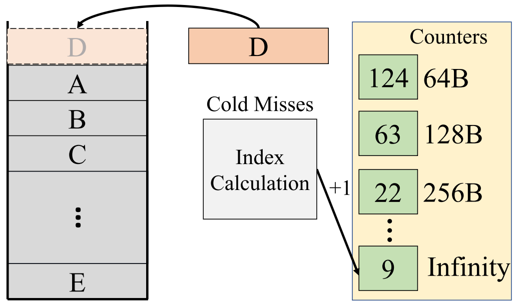


Fig. 2. Example of an LRU stack distance when an address D is accessed for the first time in the application run. The address D is put at the top of the stack, and as D has not been seen before, the counter in the infinity bucket is increased by one to count toward the number of cold misses. The values of the counters are then used to predict the number of cache hits and misses.

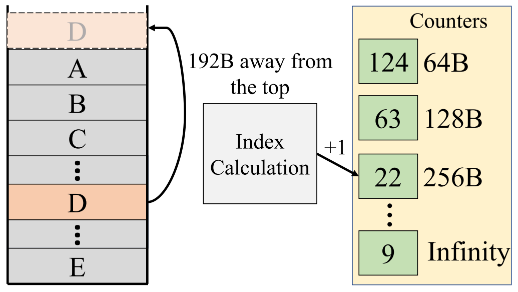


Fig. 3. Example of an LRU stack distance when an address D is accessed a second time. The address D is moved to the top of the stack, and the distance between the last location of D and the top of the stack is calculated. In this case, the counter in the 256 B bucket is increased by one as the total distance is 192 B, which is below 256 B but higher than 128 B. The total distance can be calculated by multiplying the number of entries away from the previous use of D by the cache line size. The values of the counters are then used to predict the number of cache hits and misses.

of the stack is calculated. The counter in the smallest bucket that is larger than the distance is incremented by one. In this case, the 256 B bucket is incremented as 192 B is bigger than 128 B and smaller than 256 B. The total LRU stack distance can be calculated by multiplying the number of entries away from the previous use of D by the cache line size.

To compute the number of cache hits and misses for a particular cache size, the number of accesses at the bucket sizes higher than the actual processor cache size is summed to determine the number of cache misses. Similarly, to find the actual number of cache hits for a particular cache size, the number of cache accesses at the sizes equal to and smaller than the processor cache size is summed. We call this step the cache event prediction phase.

Fig. 4 presents the process of cache miss event prediction. The histogram shows the number of accesses that occur in each bucket size. Each bucket represents the cache size needed for those accesses to result in a cache hit. In this case, the target processor has a 2-KiB data cache. The number of accesses that is in the buckets smaller or equal to 2 KiB will be summed up to get the number of cache hits. Similarly, the rest of the buckets are used to calculate the number of cache misses, as those accesses would require a cache that is larger than the cache used on the evaluated system at run time. Note that this histogram can be flexibly reused across different cache size

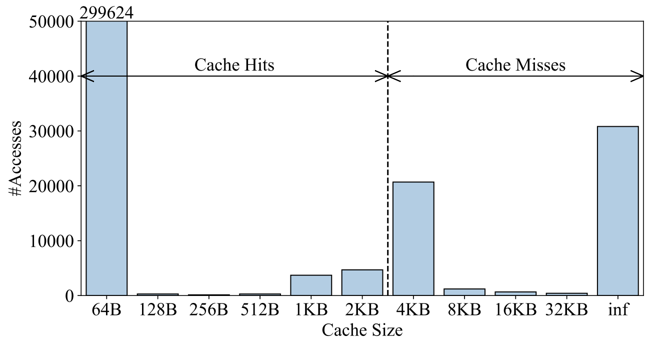


Fig. 4. Part of the dijkstra benchmark's data cache histogram from the LRU stack processing algorithm. Assuming that the target processor has a 2-KiB data cache, the bucket sizes that fall into the left area are the groups that contribute to the number of cache hits, and the bucket sizes in the right area are the groups that contribute to the number of cache misses.

configurations using this methodology without rerunning the preparation phase.

IV. Threat Model

Fig. 5 shows the threat model used in this work. In this threat model, hardware Trojans are assumed to be inserted into the processor during the RTL design stage by a malicious third-party design company. This company then gives an IP block of the Trojan-infected processor to a trusted chip design company. The proposed countermeasure is implemented and integrated with the chip design at this stage. The chip design company then integrates the Trojan-infected IP block with other Trojan-free components and runs a variety of tools to generate the final chip design for tape-out. These tools are assumed to be from well-known, certified EDA companies and are trusted. The final design is then fabricated at a trusted foundry.

Hardware Trojans are designed to be stealthy. Thus, they are assumed to be able to remain inactive during normal execution, which allows them to circumvent prelayout and post-layout verification. It is also assumed that they can avoid detection during chip testing. In addition, the developers who build and maintain software for the Trojan-infected processor are assumed to be trusted.

After fabrication, a user deploys the hardware-Trojan infected chip to execute their workloads, which could be critical applications [28]. During execution, the hardware Trojan is activated at a critical moment, like when the application is run in a nuclear or power facility, to attack the processor. Several types of attacks exist with their respective countermeasures. In this work, we focus on DoS and performance degradation attacks. Detecting application performance differences that are within normal operating ranges is outside the scope of this work.

This threat model is similar to the threat models used in state-of-the-art related works [9] and similar to the untrusted third-party vendor scenario, which is one of the five most common chip-level threat models [29]. The model ensures that the hardware components of the proposed methodology remain unmodified during the chip design process and allows for this work to focus on hardware Trojan issues.

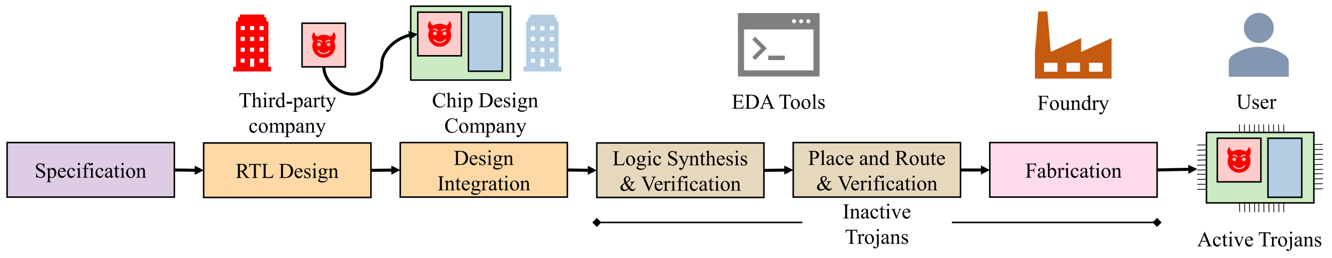


Fig. 5. Overview of the hardware Trojan threat model consisting of a variety of phases from specification to RTL design and integration to fabrication and packaging to customers. Hardware Trojans are assumed to be added by a third-party company at the RTL design stage inside an IP block. The IP block is then integrated to a bigger system by a trusted chip design company.

V. Methodology

Figs. 6 and 7 present an overview of both the software and hardware aspects of the proposed methodology, with the numbers in the figures corresponding to each of the steps of the methodology described below.

A. Software Flow

To ensure that the application will not be modified to bypass the hardware Trojan detector, and as an example methodology, the trusted software developers must first follow these steps.

1) The trusted chip developing company generates a private key that is only shared, through a secure channel, for example, based on a public-key cryptography algorithm, with trusted software companies that develop the applications.
2) The trusted software developers build their applications and generate binaries that must be profiled to capture the fingerprint of the application for use in branch and cache analytical models. The result is an application-specific but microarchitecture-independent profile that can be reused across different architectures and configurations. Thus, profiling is only done once for an application. The profile generated is the crucial information that will be used to predict microarchitectural events by the proposed hardware Trojan detector. Unlike prior work [9] that requires low-level knowledge of the target processor (e.g., RTL code and gate-level netlist), our proposed methodology only requires high-level configuration information of the processor design, such as data cache and instruction cache sizes, which is known by the hardware and software vendors.
3) The generated profile is sent to the message authentication code (MAC) algorithm to generate a profile tag using the private key provided by the chip company. The MAC algorithm ensures that the untrusted processor does not modify the application profile sent to the proposed Trojan detector.
4) Finally, the application binary, the profile, and the profile tag are sent to the user to run the application.

B. Hardware Flow

To ensure that the binary and profile are not tampered with and hardware Trojans are detected when they are activated, the hardware follows these steps.

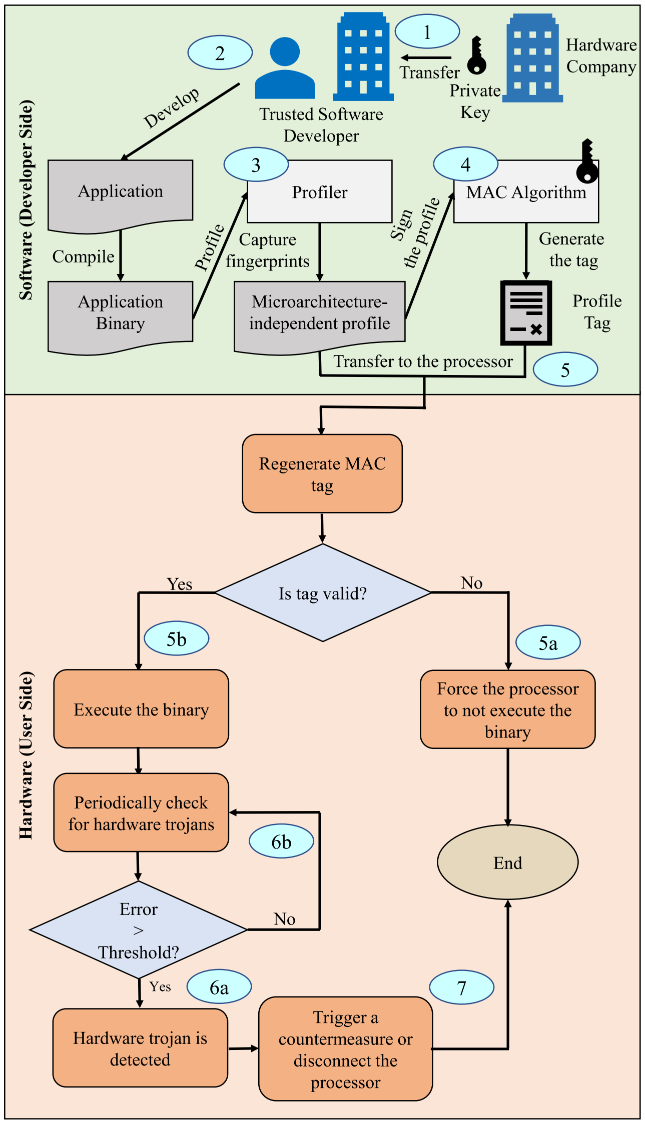


Fig. 6. Overview of the proposed methodology consisting of two sides: developer side and hardware side. The developer side is where the software developer prepares their application for distribution to users. The hardware side receives an application binary with its profile and profile tag to run the application and checks for hardware Trojans.

1) The profile and profile tag are first given to the secure detector through the untrusted processor to verify their correctness. The secure detector also holds the same secret key as the software developers. The secure detector regenerates the profile tag using its private key

and compares it with the tag sent from the software developers. Note that the MAC algorithm used is provided as an example to demonstrate this methodology. One algorithm can be used if they are deemed more secure for their use cases.

a) If the tags differ, meaning that either the tag or profile has been tampered with, the untrusted processor is stopped and not allowed to execute the binary.

b) If the tags are the same, the processor is allowed to execute the binary.

2) The Trojan detector uses the profile to generate data for detecting Trojans when the target application is loaded into the processor. The processor then executes the binary with the detector, periodically checking for hardware Trojans by predicting microarchitectural events based on the given profile for every N instruction (instruction window).

a) If the number of microarchitectural events is outside the upper-bound and lower-bound values of the predicted microarchitectural events, a hardware Trojan is detected.

b) If the difference in performance is lower than the threshold, execution continues.

3) When a hardware Trojan is detected, the secure detector can trigger a countermeasure, for example, disconnecting the processor from other IP blocks or turning the processor off.

C. Performance Counter Interception Attack Prevention

The MAC algorithm is used in our proposed methodology (see Fig. 6) to generate a verifiable tag for each application. The MAC is then used to verify that the untrusted processor is sharing a correctly generated (and trusted) application profile from the original software developer and that it has not been tampered with. The trusted processor can validate this profile during run time to allow the untrusted processor to operate.

If the processor contains a hardware Trojan that can intercept performance counter values sent to the detector, it can modify the profile to match the fake performance counter values to bypass this detection methodology. This type of Trojan is similar to the one proposed in a recent work [9]. Instead, through the use of a verification tag MAC, the correctness of the profile can be verified, and this type of hardware Trojan will be blocked in this methodology. It can no longer continue to run as both the correct performance counter values for the entire application run, as well as a matching profile needs to be presented. This is not possible when hardware Trojans are operating as they do not have knowledge of the application and the software developer who originally generated the application profile is trusted.

D. Hardware Design

Fig. 7 shows the overview of the proposed hardware design. A lightweight Trojan detector is integrated into the chip to predict the number of microarchitectural events in the processor pipeline based on the application profile to detect

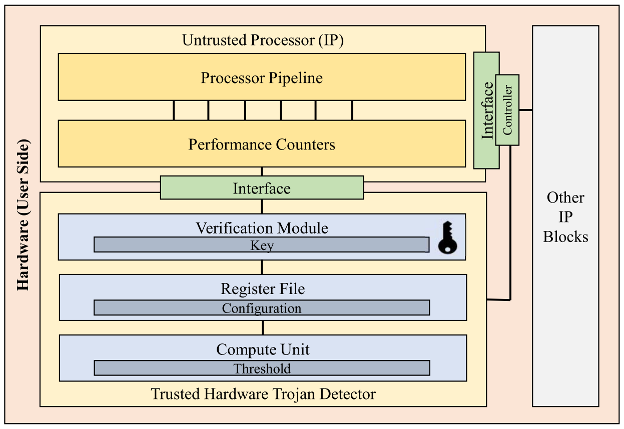


Fig. 7. Overview of the proposed hardware design. The untrusted processor IP is connected to the trusted hardware Trojan detector via an interface. The processor may also be connected to other IP blocks through a controller which is controlled by the detector.

anomalies. In modern processors, the number of microarchitectural events can be found in the performance counters that keep track of important events (e.g., branch misses and cache misses). Therefore, the Trojan detector is directly connected to the performance counter module through the IP interface. The untrusted processor may also connect to other IP blocks via an interface that contains a controller that enables the secure detector to control the processor. Connecting the detector directly to the processor at the hardware level is beneficial as the interface used to transfer performance counter data is not as limited as reading the data at the software level.

The untrusted processor normally comes in the form of an IP block from a third-party company. IP blocks generally do not allow hardware developers to have access to the implementation of the design, i.e., RTL code, to ensure that proprietary components and designs are not exposed to people outside the company. However, common interfaces are typically exposed to the hardware developers to allow them to integrate them with other components or IP blocks. The methodology is suitable for this setting as it does not require any information from the RTL code and only requires the secure detector to have access to the performance counters of the processor to be validated.

The Trojan detector consists of three key components: 1) verification module; 2) register file module; and 3) compute unit. The verification module implements the MAC algorithm to check the correctness of the profile and contains the private key. The register file module and the compute unit implement the linear branch model and cache event prediction.

The register file module stores the α and β parameters for the linear branch model and data and instruction cache size parameters for the LRU cache model. These parameters can be hard-coded by trusted hardware developers as they are fixed in each processor. As the performance counters do not reset their values after each read request, the register file module is also required to store performance counter values read for the previous detection period. These previous values will then be used with the current performance counter values that are read in the current detection period to calculate the number of events that occur in the current instruction window by subtracting the current performance counter values from the previous ones. The register file module also stores the acceptable ranges of microarchitectural events generated before running the application.

The compute unit calculates the branch model's miss rate estimation, the cache model's cache event prediction, and the range calculation. For the range calculation, it first generates a pair of upper- and lower-bound values for each predicted value based on the threshold and confidence interval. These values are then stored in the register file module to compare the actual value from the performance counter with this pair at run-time. The range calculation phase is done during the programming phase of the processor before running an application.

During the detection phase when an application is being run, the secure detector periodically observes hardware performance counters in each instruction window. This presents an opportunity for the secure detector to be in an idle state to save power when it is not the time to make an observation. The observed data is then compared with the predicted data. If the error between the observed and predicted data is higher than the specified threshold, meaning that an anomaly was found, the secure detector can send a signal that a hardware Trojan has been detected, for example, to trigger a countermeasure.

Due to the high accuracy of modern branch predictors and the high data reuse of applications, the number of branch and cache misses is generally relatively small. Therefore, tracking misses can lead to a higher number of false positives, as a slight difference between the analytical models and actual hardware can create a significant error value. For this reason, we use the number of branch and cache hits in this work.

For example, assuming that there is an application that has 100 K memory accesses and has 1% and 2% cache miss rates from actual hardware and analytical model, respectively, the difference in the cache miss rates may seem small, but, in fact, the actual misses are 1000 and 2000 misses, respectively, leading to 50% error (as seen by measuring the miss rate) in total. In contrast, by considering a number of hits instead, the same application has 99% and 98% cache hit rates, respectively, meaning that it has 99K and 98K actual cache hits which lead to only 1% error.

E. Upper- and Lower-Bound Calculation

We use confidence intervals to calculate acceptable microarchitectural event ranges based on the analytical models' predicted values. However, since an application consists of different phases that may have largely different microarchitectural event characteristics, our methodology will need to determine when applications switch between phases. There are a number of different methods to determine application phases, from the use of basic block vectors [30], to LRU stack distance [31] or even both of them at the same time [20]. Instead, in this work, we first apply a changepoint detection algorithm [12] to detect phase changes in the LRU stack distance [31] information during application profiling as this information is readily available. Next, a confidence interval is calculated for each phase and type of microarchitectural event to determine upper- and lower bounds. A threshold value is then applied to ensure that the range handles errors that can be introduced by inaccuracies in the analytical models. The secure detector approximates the standard deviation by applying the Range Rule of Thumb [32], which simplifies the hardware implementation.

TABLE I
ROCKET CHIP [33] CONFIGURATION USED IN THIS WORK. THE NUMBER OF PERFORMANCE COUNTERS INCLUDES ONLY THE CONFIGURABLE ONES

Component	Parameter
ISA	RV64GC
L1I Cache	LRU, 4-way 16 kB
Core	5-stage in-order
L1D Cache	LRU, 4-way 16 kB
Branch Predictor	gshare
L2D Cache	Inclusive, 8-way 512 kB
Technology Node	22 nm
No. of perf. counters	15

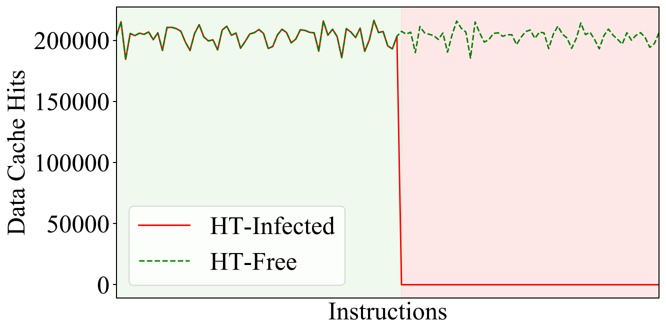
VI. EVALUATION

A. Experimental Setup

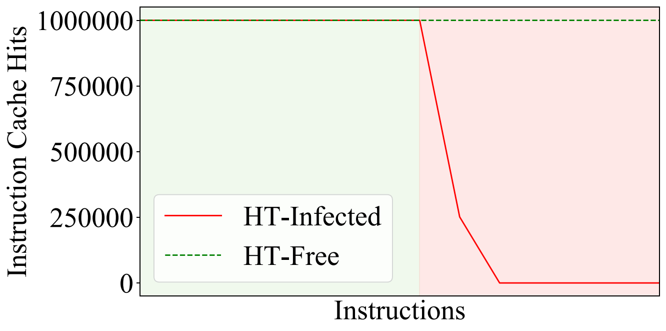
We evaluate the proposed methodology on Rocket Chip [33] a 5-stage, in-order RISC-V processor (see Table I for configuration details). The secure detector reads the control and status register (CSR) module, which contains configurable hardware performance counters. The core is synthesized using Synopsys Design Compiler version P-2019.03 targeting a 22-nm technology node and runs at 250 MHz. Note that the secure detector connected to the processor is off the critical path and does not affect the maximum frequency. Power analysis is performed with Synopsys PrimePower version P-2019.03. We use the Firesim FPGA platform [34], an open-source cycle-accurate FPGA-accelerated full-system hardware simulation platform, to conduct the experiments in this work. We test a selection of MiBench embedded benchmarks [35] using 99.5% confidence intervals with 4% thresholds, respectively. The instruction window size is 1 million instructions.

The gem5 simulator [36] is used as an ISA frontend for the branch predictor and cache analytical models. As the gem5 simulator models branch predictor effects on the instruction cache, instructions that are loaded and flushed later in the pipeline due to branch misprediction can add noise to the data trace and are ignored in this work. Only committed instructions are analyzed to avoid noise. No additional information from a detailed simulator is used. The gem5 simulator is used as a faster methodology for profile data collection and is shown to have relatively high accuracy. One can increase the accuracy of the profiling data by using cycle-level models but at the cost of a significantly longer run time.

We use a linear branch model with parameters α (–0.189) and β (52.322) obtained from [21] based on the 2011 Championship Branch Predictor (CBP) competition training dataset [25] based on the x86 ISA. One can train a linear model as described in Section III to generate the model parameters for a specific branch predictor. A cache access is modeled to happen at every fetched instruction and data value instead of

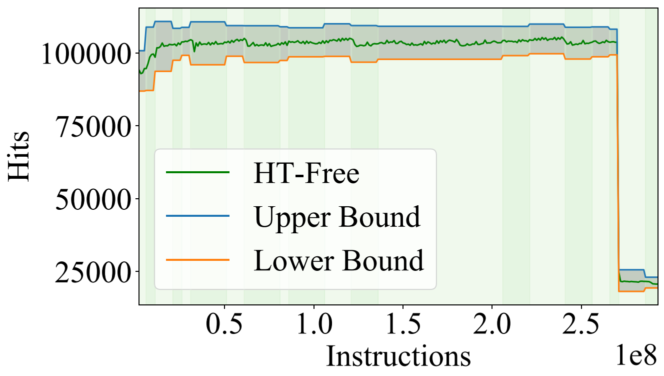


(a)

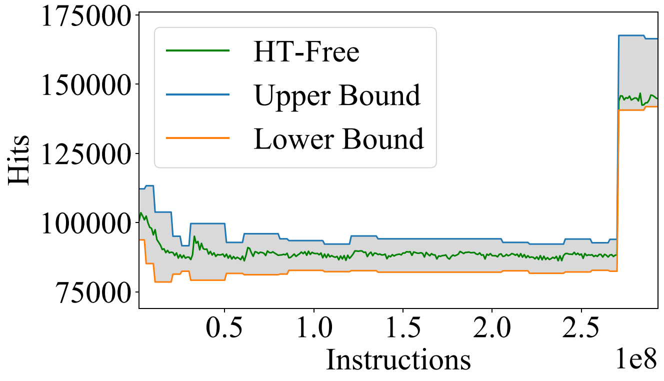


(b)

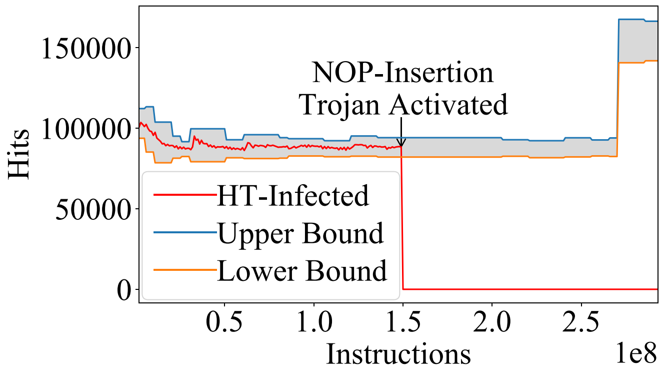
Fig. 8. Effects of the Trojans. The green area indicates the period that the Trojans are inactive. The red area indicates the period that the Trojans are active. (a) *NOP-insertion Trojan* inserts NOP instructions into the processor pipeline reducing processor activities including memory requests, thus significantly decreasing data cache hits. (b) *Cache-disabling Trojan* disables the instruction cache and the number of cache hits slumps. Note that the number of instruction cache misses also goes up significantly.



(a)



(b)



(c)

Fig. 9. Predicted bounds for cache and branch events and an example of how the proposed methodology detects a Trojan in the *basicmath* benchmark with 99.5% confidence intervals and 4% threshold. (a) Data cache hits during a Trojan-free run with the transition of execution phases, represented as a change in color, detected by the changepoint detection algorithm. Each phase has the same upper and lower bounds. (b) Branch prediction hits for a Trojan-free execution. (c) Detection of a hardware Trojan-infected execution by observing the number of branch prediction hits. As the NOP-insertion Trojan affects the number of branch prediction hits significantly, the proposed methodology can detect the Trojan in this workload with 100% accuracy.

every fetched memory and data block to imitate the behavior of Rocket Chip. The same methodology can also be applied to processors that fetch block-level data.

We evaluate three hardware Trojans, two of which are built based on Trust-Hub [4], with each Trojan triggered in the middle of the benchmark run (Fig. 8).

1) *NOP-insertion Trojan* inserts NOP instructions at the front-end to launch a DoS attack [12]. This is similar to PIC16F84-T200 from Trust-Hub.
2) *Cache-disabling Trojan* forces every cache access to miss in order to disable the instruction cache [12]. This is similar to s35932-T300 from Trust-Hub.
3) *NOP delay Trojan* is a Trojan that is built to evaluate the effectiveness of our solution in a milder interference scenario by periodically inserting NOP instructions 40% of the time when active to slow down the core.

B. Methodology Demonstration

Fig. 9 demonstrates how the proposed technique works, using the analytical cache model shown in Fig. 9(a) and analytical branch predictor model illustrated in Fig. 9(b), to define the detection boundaries, as well as how the analytical branch prediction model is able to detect an NOP-insertion Trojan shown in Fig. 9(c).

Fig. 9(a) shows the number of L1 data cache hits over time. The L1 data cache histograms are used with the changepoint detection algorithm to detect phases, which are then used to calculate acceptable ranges for all models. As illustrated, phase detection helps the methodology to adjust the width of the acceptable ranges in a fine-grained way to maintain appropriate ranges for each phase of the program, ensuring that the ranges are not too wide to overlook anomalies from hardware Trojans and not too narrow to suffer from a high FPR.

The model only requires the branch prediction algorithm (gshare in this work) and its size as input. Combined with

TABLE II
TPR, FPR, AND THE PERFORMANCE IMPACT EACH TROJAN HAS ON THE APPLICATION. DoS STANDS FOR DENIAL OF SERVICE, MEANING THAT THE WORKLOAD DOES NOT COMPLETE

Benchmark	Trojan No.	TPR (%)	FPR (%)	Perf. Impact (%)
basicmath	1	100.00	0.00	DoS
	2	100.00	0.00	685.70
	3	100.00	0.00	25.02
dijkstra	1	100.00	0.00	DoS
	2	100.00	0.00	668.06
	3	100.00	0.00	24.96
fft	1	100.00	0.00	DoS
	2	100.00	0.00	557.80
	3	100.00	0.00	25.83
qsort	1	100.00	0.00	DoS
	2	100.00	0.00	352.42
	3	100.00	0.00	28.38

TABLE III
VARIATION OF DATA CACHE (D$) AND BRANCH HITS (BR.) IN THE TWO LARGEST PHASES IN EACH APPLICATION, SHOWN AS MAXIMUM DEVIATION PERCENTAGE (THE DIFFERENCE BETWEEN THE MAXIMUM AND AVERAGE VALUES IN THAT PHASE). *Size* IS THE PORTION OF INSTRUCTIONS FOR EACH PHASE RELATIVE TO THE TOTAL NUMBER OF INSTRUCTIONS OF THE APPLICATION. NOTE THAT *qsort* ONLY HAS ONE PHASE

	1^{st}			2^{nd}		
	D$	Br.	Size	D$	Br.	Size
basicmath	1.15%	1.87%	23.97%	2.64%	5.96%	6.85%
qsort	11.55%	9.79%	100.00%	-	-	-
dijkstra	1.88%	0.09%	4.20%	6.79%	0.35%	4.20%
fft	0.20%	1.39%	83.33%	0.75%	8.86%	8.55%

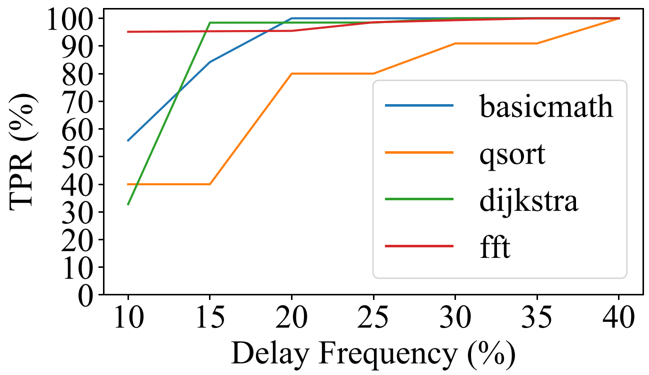


Fig. 10. TPRs with different delay frequencies to observe the detector's sensitivity. 10% delay frequency means the Trojan periodically delays the core by an additional 10% of the time (when the Trojan is active, not the total time).

TABLE IV
ERROR RATE INDUCED BY TROJAN 3 FOR DIFFERENT DELAY FREQUENCIES. THE NUMBERS ON THE LEFT AND RIGHT OF EACH PAIR INDICATE THE ERRORS IN DATA CACHE HITS AND BRANCH HITS, RESPECTIVELY. INSTRUCTION CACHE HITS ARE NOT AFFECTED

Delay Frequency	basicmath	qsort	dijkstra	fft
10%	1.08% / 7.15%	4.17% / 4.28%	3.99% / 3.98%	6.09% / 0.98%
15%	1.69% / 10.90%	7.12% / 7.13%	6.11% / 6.14%	9.03% / 1.64%
20%	2.35% / 15.14%	9.14% / 9.80%	8.47% / 8.54%	9.47% / 4.62%
25%	3.07% / 19.60%	11.65% / 12.17%	10.83% / 10.96%	12.07% / 8.09%
30%	4.28% / 24.18%	14.90% / 15.30%	13.49% / 13.69%	15.66% / 11.83%
35%	8.15% / 27.25%	18.28% / 18.53%	16.49% / 16.58%	19.43% / 15.77%
40%	12.23% / 30.47%	20.93% / 21.22%	19.68% / 19.88%	23.38% / 19.89%

appropriate branch information from the application, the model predicts branch miss rates periodically throughout the run that are used to calculate the number of acceptable branch hits in the entire run shown as a gray area in Fig. 9(b). Any value outside this area is considered anomalous due to interference from a hardware Trojan.

In Fig. 9(c), the red line shows the number of branch hits after the NOP-insertion hardware Trojan is activated. Our methodology can detect this with 100% accuracy for this workload, as the number of branch hits falls out of the gray zone. In contrast, this Trojan can entirely circumvent a state-of-the-art changepoint detection algorithm [12], as it does not generate a change in the data stream observable by the detection algorithm.

C. Evaluation Results

Table II shows that the proposed methodology, for the workloads evaluated, can detect all of the anomalous data points while having no false positives due to the significant deviation of the predicted and the monitored microarchitectural events. This is also evident from the significant performance impact each Trojan has on the application, as shown in the table.

1) Sensitivity Study: The sensitivity of the proposed Trojan detector depends on how wide the boundaries of the monitored microarchitectural events are. These boundaries are defined by the workload variability during execution, which depends on the inherent characteristics of each application. Table III shows the variations within the normal operating ranges of the applications. A variation is defined as the difference in percentage between the maximum and average values in a specific phase. *qsort* has only one phase and has a large variation in data cache and branch hits, 11.55% and 9.79%, respectively. This means that the data cache boundaries in that phase must cover at least 11.55% above and below the average value to have zero false positives, which is the reason why the TPRs increase at the slowest rate in Fig. 10.

From Fig. 10, the delay frequency of the NOP delay Trojan is changed to measure the accuracy when the effect of this Trojan becomes milder. 10% delay frequency means the Trojan periodically delays the core by an additional 10% of the time when the Trojan is active. The proposed methodology achieves at least 84% TPR when the delay frequency is at 15% in *basicmath, dijkstra*, and *fft*, which leads to the total errors ranging from 1.69% to 9.03% in data cache hits and 1.64% to 10.90% in branch hits as shown in Table IV. For *qsort*, the proposed methodology requires at least 20% delay frequency, leading to around 9% errors in both types of events, to achieve more than 80% TPR. This shows that our methodology can detect even a stealthy Trojan that periodically attacks the core.

One may argue that a hardware Trojan that causes, say, 1% performance impact on the application run can circumvent this proposed methodology as the Trojan does not significantly affect the number of microarchitectural events. However, 1% performance impact is within the expected variance of the analytical models and is out of the scope of this work.

2) Power Overhead: The Rocket Chip CPU consumes 257 mW while the detector and MAC unit consume 0.0044

TABLE V
DETECTION DELAY AS REPORTED BY PRIOR WORKS [9], [12], [13] COMPARED TO THE PROPOSED METHODOLOGY

Methodology	Detection Delay
Elnaggar et al. [13]	71.6 μs
Elnaggar et al. [12]	7800.0 μs
Vijayan et al. [9]	80.8 μs
This work	0.2 μs

TABLE VI
BEST TPR AND FPR AS REPORTED BY PRIOR WORKS [9], [12], [13] COMPARED TO THE PROPOSED METHODOLOGY

Methodology	Trojan No.	TPR (%)	FPR (%)
Elnaggar et al. [13]	1	94.0	4.0
	2	100.0	1.0
Elnaggar et al. [12]	1	Fail	Fail
	2	99.9	0.0
Vijayan et al. [9]	1	95.0	0.0$^{(1)}$
	2	85.0	0.0$^{(1)}$
This work	1	100.0	0.0
	2	100.0	0.0

(1) Based on reported precision

and 0.0079 mW, respectively, leading to 0.005% power overhead. The low power is due to clock gating, as the detector is mostly idle in between detections. Also, complex computations during the preparation phases of the analytical models are done at the software flow, prior to running the application. This work provides significantly lower power overhead as the design is built specifically for the methodology and integrated into the chip while the prior works [9], [12], [13] rely on an off-chip desktop processor to execute their detection algorithm.

3) Detection Delay: The proposed methodology provides significantly lower detection delay at 0.2 μs compared to prior works as shown in Table V. This is due to the fact that the proposed methodology integrates the secure detector inside the chip, and the secure detector does not need to execute a complex machine-learning algorithm like the previous works. Having a lower detection delay allows a preventive mechanism to have a faster response to the effects of hardware Trojans.

4) Comparison With the State of the Art: Table VI compares this work with the state-of-the-art run-time hardware Trojan detection methodologies based on their published results on Trojans 1 and 2. The methodology from Elnaggar et al. [12] is unable to detect Trojan 1 as it does not generate a significant change in the observed CPU performance counters. Nevertheless, they were able to achieve 99.9% TPR and 0% FPR on Trojan 2. Note that it achieves 99.9% TPR only when the appropriate performance counter is picked, as the work has a limited performance counter interface that allows the detection system to collect only two performance counter data values at a time. The methodology from Elnaggar et al. [13] achieves 94.0% and 100.0% TPRs while having 4.0% and 1.0% FPRs in Trojans 1 and 2, respectively. Similarly, the methodology from Elnaggar et al. [13] suffers from the same performance counter limitation as Elnaggar et al. [12]. Vijayan et al. [9] achieved 95% and 85% TPR on the Trojans 1 and 2, respectively, while maintaining a 0% FPR. Our proposed methodology achieves a high true positive detection rate with no false positives while having negligible power overhead, unlike these prior works which require an additional secure off-chip co-processor to run their detection algorithms.

VII. CONCLUSION

This work proposes a hardware Trojan detection methodology based on analytical modeling that can predict several microarchitectural events using only high-level processor configuration information without a golden reference. The predicted values are then compared to the actual values from performance counters to detect hardware Trojans. The methodology achieves a high true positive detection rate compared to prior works, with no false positives and negligible power overhead of 0.005%.

REFERENCES

[1] M. Xue, C. Gu, W. Liu, S. Yu, and M. O'Neill, "Ten years of hardware Trojans: A survey from the attacker's perspective," *IET Comput. Digit. Technol.*, vol. 14, no. 6, pp. 231–246, Sep. 2020.

[2] J. Francq and F. Frick, "Introduction to hardware Trojan detection methods," in *Proc. Design, Autom. Test Europe Conf. Exhibit. (DATE)*, 2015, pp. 770–775.

[3] S. Ray, W. Chen, and R. Cammarota, "Invited: Protecting the supply chain for automotives and IoTs," in *Proc. 55th ACM/ESDA/IEEE Design Autom. Conf. (DAC)*, 2018, pp. 1–4.

[4] H. Salmani, M. Tehranipoor, and R. Karri, "On design vulnerability analysis and trust benchmarks development," in *Proc. IEEE 31st Int. Conf. Comput. Design (ICCD)*, 2013, pp. 471–474.

[5] H. Li, Q. Liu, and J. Zhang, "A survey of hardware Trojan threat and defense," *Integration*, vol. 55, pp. 426–437, Dec. 2016.

[6] S. Charles and P. Mishra, "A survey of network-on-chip security attacks and countermeasures," *ACM Comput. Surveys*, vol. 54, no. 5, p. 101, May 2021.

[7] K. Wang, H. Zheng, and A. Louri, "TSA-NoC: Learning-based threat detection and mitigation for secure network-on-chip architecture," *IEEE Micro*, vol. 40, no. 5, pp. 56–63, Sep./Oct. 2020.

[8] S. Charles and P. Mishra, "Reconfigurable network-on-chip security architecture," *ACM Trans. Design Autom. Electron. Syst.*, vol. 25, no. 6, p. 53, Aug. 2020.

[9] A. Vijayan, M. B. Tahoori, and K. Chakrabarty, "Runtime identification of hardware Trojans by feature analysis on gate-level unstructured data and anomaly detection," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 25, no. 4, p. 33, May 2020.

[10] S. Bhunia and M. Tehranipoor, *The Hardware Trojan War: Attacks, Myths, and Defenses*. Cham, Switzerland: Springer, Dec. 2018.

[11] M. Tehranipoor and F. Koushanfar, "A survey of hardware Trojan taxonomy and detection," *IEEE Design Test Comput.*, vol. 27, no. 1, pp. 10–25, Jan./Feb. 2010.

[12] R. Elnaggar, K. Chakrabarty, and M. B. Tahoori, "Hardware Trojan detection using changepoint-based anomaly detection techniques," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 12, pp. 2706–2719, Dec. 2019.

[13] R. Elnaggar, K. Chakrabarty, and M. B. Tahoori, "Run-time hardware Trojan detection using performance counters," in *Proc. IEEE Int. Test Conf. (ITC)*, 2017, pp. 1–10.

[14] F. K. Lodhi, S. R. Hasan, O. Hasan, and F. Awwadl, "Power profiling of microcontroller's instruction set for runtime hardware Trojans detection without golden circuit models," in *Proc. Design, Autom. Test Europe Conf. Exhibit. (DATE)*, 2017, pp. 294–297.

[15] S. Narasimhan, W. Yueh, X. Wang, S. Mukhopadhyay, and S. Bhunia, "Improving IC security against Trojan attacks through integration of security monitors," *IEEE Design Test Comput.*, vol. 29, no. 5, pp. 37–46, Oct. 2012.

[16] J. He, X. Guo, H. Ma, Y. Liu, Y. Zhao, and Y. Jin, "Runtime trust evaluation and hardware Trojan detection using on-chip EM sensors," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, 2020, pp. 1–6.

[17] C. Bao, D. Forte, and A. Srivastava, "Temperature tracking: Toward robust run-time detection of hardware Trojans," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 10, pp. 1577–1585, Oct. 2015.

[18] C. Sturton, M. Hicks, S. T. King, and J. M. Smith, "FinalFilter: Asserting security properties of a processor at runtime," *IEEE Micro*, vol. 39, no. 4, pp. 35–42, Jul./Aug. 2019.

[19] J. Dubeuf, D. Hély, and R. Karri, "Run-time detection of hardware trojans: The processor protection unit," in *Proc. 18th IEEE Eur. Test Symp. (ETS)*, 2013, pp. 1–6.

[20] T. E. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout, "BarrierPoint: Sampled simulation of multi-threaded applications," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, 2014, pp. 2–12.

[21] S. De Pestel, S. Eyerman, and L. Eeckhout, "Linear branch entropy: Characterizing and optimizing branch behavior in a micro-architecture independent way," *IEEE Trans. Comput.*, vol. 66, no. 3, pp. 458–472, Mar. 2017.

[22] S. Van den Steen et al., "Analytical processor performance and power modeling using micro-architecture independent characteristics," *IEEE Trans. Comput.*, vol. 65, no. 12, pp. 3537–3551, Dec. 2016.

[23] N. Binkert et al., "The Gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[24] C.-K. Luk et al., "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM-SIGPLAN Symp. Program. Lang. Design Implement. (PLDI)*, 2005, pp. 190–200.

[25] "Championship branch prediction," presented at 2nd JILP Workshop Computer Archit. Competitions (JWAC-2), 2011. [Online]. Available: https://jilp.org/jwac-2/

[26] R. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, Jun. 1970.

[27] D. Eklov and E. Hagersten, "StatStack: Efficient modeling of LRU caches," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, 2010, pp. 55–65.

[28] S. Adee, "The hunt for the kill switch," *IEEE Spectr.*, vol. 45, no. 5, pp. 34–39, May 2008.

[29] Z. Huang, Q. Wang, Y. Chen, and X. Jiang, "A survey on machine learning against hardware Trojan attacks: Recent advances and challenges," *IEEE Access*, vol. 8, pp. 10796–10826, 2020.

[30] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. ACM Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2002, pp. 45–57.

[31] X. Shen, Y. Zhong, and C. Ding, "Predicting locality phases for dynamic memory optimization," *J. Parallel Distrib. Comput.*, vol. 67, no. 7, pp. 783–796, Jul. 2007.

[32] X. Wan, W. Wang, J. Liu, and T. Tong, "Estimating the sample mean and standard deviation from the sample size, median, range and/or interquartile range," *BMC Med. Res. Methodol.*, vol. 14, pp. 1–13, Dec. 2014.

[33] K. Asanović et al., "The rocket chip generator," Electr. Eng. Comput. Sci., Univ. California, Berkeley, CA, USA, Rep. UCB/EECS-2016-17, 2016.

[34] S. Karandikar et al., "FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2018, pp. 29–42.

[35] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. IEEE Int. Workshop Workload Characterization (WWC-4)*, 2001, pp. 3–14.

[36] J. Lowe-Power et al., "The gem5 simulator: Version 20.0+," 2020, *arXiv:2007.03152*.

Burin Amornpaisannon (Student Member, IEEE) is currently pursuing the Ph.D. degree with the National University of Singapore, Singapore, under the supervision of Dr. T. E. Carlson and Dr. L.-S. Peh.

His research interests include understanding physical attack vectors, efficient neuromorphic computing, and computer architecture in general.

Andreas Diavastos (Member, IEEE) received the Ph.D. degree in computer architecture from the University of Cyprus, Nicosia, Cyprus, in 2018.

He did his Postdoctoral Fellowship with the Computer Architecture Group, National University of Singapore, Singapore, and then joined the Universitat Politècnica de Catalunya, Barcelona, Spain, as a Distinguished Researcher. His research interests include processor and accelerator architectures, hardware–software co-design, parallel programming and execution models, automatic parallelization, and high-performance computing. He developed the SWITCHES parallel runtime system that includes the first auto-tuning system for static schedules for task data-flow applications on multi- and many-core systems.

Li-Shiuan Peh (Fellow, IEEE) received the B.S. degree in computer science from the National University of Singapore (NUS), Singapore, in 1995, and the Ph.D. degree in computer science from Stanford University, Stanford, CA, USA, in 2001.

She joined NUS as a Provost's Chair Professor with the Department of Computer Science, with a courtesy appointment in the Department of Electrical and Computer Engineering in September 2016. Previously, she was a Professor of Electrical Engineering and Computer Science with the Massachusetts Institute of Technology (MIT), Cambridge, MA, USA, and was on the faculty of MIT since 2009. She was also the Associate Director for Outreach of the Singapore–MIT Alliance of Research and Technology from 2015 to 2016. Prior to MIT, she was on the faculty of Princeton University, Princeton, NJ, USA, since 2002. Her research focuses on networked computing, in many-core chips as well as mobile wireless systems.

Dr. Peh received the NRF Returning Singaporean Scientist Award in 2016, the ACM Distinguished Scientist Award in 2011, the MICRO Hall of Fame in 2011, the CRA Anita Borg Early Career Award in 2007, the Sloan Research Fellowship in 2006, and the NSF CAREER Award in 2003.

Trevor E. Carlson (Senior Member, IEEE) received the bachelor's and master's degrees from Carnegie Mellon University, Pittsburgh, PA, USA, in 2002 and 2003, respectively, and the Ph.D. degree from Ghent University, Ghent, Belgium, in June 2014,

He was a Postdoctoral Fellow with Uppsala University, Uppsala, Sweden, in 2017, He is an Assistant Professor with the National University of Singapore, Singapore. He has over 16 years of computer systems and architecture experience in both industry and academia. His research interests include efficient general-purpose processing, secure systems, AI acceleration, and simulation methodologies. He co-develops the Sniper Multi-Core Simulator, which is being used by hundreds of researchers in academia and industry, to evaluate the performance and power efficiency of next-generation systems.

Dr. Carlson's work has received six best paper or best paper nominations in conferences, such as the International Symposium on Microarchitecture (MICRO) and the International Symposium on Performance Analysis of Systems and Software.