applied sciences

MDPI

Article

MATANA: A Reconfigurable Framework for Runtime Attack Detection Based on the Analysis of Microarchitectural Signals

Yuxiao Mao *, Vincent Migliore * and Vincent Nicomette *

LAAS-CNRS, Université de Toulouse, CNRS, INSA, 31400 Toulouse, France
* Correspondence: yuxiao.mao@laas.fr (Y.M.); vincent.migliore@laas.fr (V.M.); vincent.nicomette@laas.fr (V.N.)

Featured Application: The MATANA framework is particularly suited for critical embedded systems that want to design and integrate flexible attack detection mechanisms based on microarchitectural signal observation.

Abstract: Microarchitectural attacks exploit target hardware properties to break software isolation techniques used by the processor. These attacks are extremely powerful and hard to detect since the determination of the program execution's impact on the microarchitecture is at the same time not precisely understood and not easily observable at the software layer. Some approaches have attempted to benefit from existing hardware to better understand and detect the microarchitectural attacks (i.e., Hardware Performance Counters or Arm CoreSight), but such hardware was not meant to be used for cybersecurity, with reduced choice on observable signals and limited throughput of information. In this paper, we propose MATANA, an open and adaptive reconfigurable hardware/software co-designed framework. Combining fine-grained analysis of microarchitectural signals and software support, MATANA allows to design and assess detection mechanisms for attacks by characterizing their microarchitectural effects—in particular, microarchitectural attacks, but also some high-level attacks such as return-oriented programming attacks. The paper also describes a prototype implementation, built with a RISC-V softcore processor Rocket running Linux 4.15 on a Virtex-6 FPGA. We successfully used MATANA to analyze cache side-channel attacks and build attack detection logic from two different perspectives: instruction-based and memory-access-based. We also successfully detected return-oriented programming attacks by exhibiting a specific behavioral pattern on the microarchitecture.

Keywords: attacks detection; hardware/software co-design; side-channel attacks; return-oriented programming; RISC-V

Citation: Mao, Y.; Migliore, V.; Nicomette, V. MATANA: A Reconfigurable Framework for Runtime Attack Detection Based on the Analysis of Microarchitectural Signals. *Appl. Sci.* **2022**, *12*, 1452. https://doi.org/10.3390/app12031452

Academic Editors: Guy Gogniat, Vianney Lapotre and Maria Mushtaq

Received: 11 January 2022
Accepted: 26 January 2022
Published: 29 January 2022

1. Introduction

With the emergence of attacks targeting hardware microarchitecture, including Cache Side-Channel Attacks (CSCAs) such as Flush + Reload [1] and Prime + Probe [2], and transient execution attacks such as Spectre [3] and Meltdown [4], a deep understanding of the system's microarchitecture and the study of how the microarchitecture is affected by the software execution has become a very important research issue. Microarchitectural information describes the inner state of the hardware, which is supposed to be transparent to the software. This microarchitectural information includes both storage information, such as a register indicating whether the current execution is a speculative execution, and timing information, such as the execution time of one access to the cache. Many microarchitectural attacks consist in collecting and analyzing microarchitectural information in order to gain knowledge about other processes sharing the same hardware. For example, in Flush + Reload, the attacker measures the access time to a specific memory address, they can then infer whether this address is in the cache and deduce whether the victim process accessed the same address or not. This use of microarchitectural information breaks

the hypothesis of interprocess isolation and can be used by an attacker to discover and exploit vulnerabilities.

Different prevention mechanisms, from pure hardware to pure software solutions, have already been proposed in the literature [5,6]. They mainly involve disabling the optimizations, enforcing resources isolation, or enforcing constant-time execution. However, these mechanisms are not sufficient for several reasons. As microarchitectural attacks often rely on hardware-level optimization, an always-active prevention mechanism may impede the performances of the processor or the software, which may be unacceptable in some situations. Furthermore, hardware modifications, such as the partitioning of hardware resources, also require privileged software to identify the nature of different software and manage the use of these partitions, which may be quite complicated in complex environments. Finally, it is very difficult to design prevention mechanisms that cover all the attack surface at the microarchitecture level, mainly because this attack surface is not clearly identified currently. As a consequence, it is important to design and implement detection mechanisms that complement the prevention mechanisms.

From the attack detection point of view, the fact that the microarchitectural information is hardly accessible to software makes it difficult to implement efficient and easy-to-use countermeasures against the microarchitectural attacks, and more generally, to characterize the footprint of attacks on the microarchitecture. Existing runtime analysis and detection of microarchitectural attacks mainly rely on Hardware Performance Counters (HPCs) [7]. Available on most modern processors, HPCs store counts of microarchitectural events of the processor using a set of special-purpose registers. These HPCs can be analyzed by software in order to build some behavior model of either malicious or legitimate processes. Even if HPCs have been used to successfully detect some cache attacks on cryptographic algorithms (RSA, AES, . . .), they suffer from inherent limitations: First, almost all of the proposed detection algorithms are based on software detection. Compared with dedicated hardware solutions, the cost can be prohibitive in a complex environment composed of numerous processes running in parallel. Second, proposed techniques usually lack generality and adaptability. In particular, they use a limited set of attacks to validate their proposals. Overall, none of these solutions are sufficient to address the multiplicity of the attacks and their mutation capabilities. In this paper, we propose a reconfigurable framework based on the analysis of microarchitectural signals for the detection of several classes of attacks, especially microarchitectural attacks, called MATANA. Our solution is based on a reconfigurable fabric, deeply integrated into a processor fabric, aimed at capturing meaningful internal signals by executing a set of detection algorithms and alerting the system in case of an attack.

To assess the validity of the approach, i.e., the use of a fully reconfigurable fabric for detection purposes, we evaluated our solution with a frequency gap between the detection module and the processor. We implemented MATANA into the RISC-V Instruction Set Architecture (ISA) [8] softcore processor Rocket [9] with Linux support on the Xilinx ML605 Field Programmable Gate Array (FPGA) platform. We successfully analyzed and designed detection mechanisms for two classes of attacks that exploit different mechanisms: a class of attacks that exploit microarchitecture without altering the control flow of a program (i.e., CSCAs); a class that alters the control flow (i.e., Return-Oriented Programming [ROP] attack). For the first class of attacks, we based our detection on instruction patterns and memory access patterns; we found no false positive for the first approach and a few ones for the second with better sensitivity. For the second class of attacks, we detected alteration on the control flow by defining some metrics and thresholds on attack-free benchmarks and applied them to evaluate attack benchmarks. We also successfully detected attacks with no false positives. The source code of our framework, including MATANA hardware architecture, software stack, and the use cases described in Section 5, is available at https://gitlab.laas.fr/matana (accessed on 10 January 2022).

The contributions of the paper are as follows:

- A novel microarchitectural signal runtime monitoring and analysis framework, which allows better understanding of interactions between software and microarchitecture at runtime;
- A methodology that enables, from the analysis of purposely chosen microarchitectural signals, the detection of various families of attacks via the characterization of their microarchitectural footprints;
- An open source framework implementation that can run on real hardware with Linux kernel;
- A deep presentation and analysis of two use cases in which we successfully built an appropriate detection logic for CSCAs and ROP attacks.

The paper is organized as follows. Section 2 is dedicated to the state of the art and presents some technical solutions dealing with the extraction of microarchitectural information and their relevance for intrusion detection. Section 3 provides an overview of MATANA and presents the considered threat model, the integration process into a target system, and the global detection methodology. Section 4 presents the detailed architecture of MATANA, both hardware and software. Section 5 describes two use cases of attack detection, i.e., CSCAs and ROP attacks on a RISC-V target system. Section 6 discusses the advantages and limitations of our framework, and Section 7 draws some conclusions and perspectives.

2. Related Work

As this paper is aimed at designing attack detection mechanisms at the microarchitecture level, the related work in this section focuses on existing methods that aim at collecting and analyzing microarchitectural signals and their use for attack detection.

2.1. Hardware Event Counters

HPCs are special-purpose registers located inside the processor that count the number of occurrences of microarchitectural events, such as L1 cache miss, instruction retired, bus access, branch misprediction, etc. In most modern processors, the number of events that can be monitored ranges from twenty to more than a few hundreds, but only a limited set can be selected at a time, around 2 to 8 for many processors. The HPCs were mostly designed to measure the performance of software running on top of a given processor and to help in optimizing the software. Although HPCs are not designed for security, there is significant use of HPCs for microarchitectural attack detection in several research works since they are easily accessible by software [7,10]. The proposed approaches involve profiling the software with the counted events in order to either identify the characteristics of attacks (so-called signature-based approach [11,12]), identifying the characteristics of a legitimate software (so-called anomaly-based approach [13]), or a mix of them [14]. The profiling can be performed with machine learning algorithms [10,11,13,14] or using some heuristics [12].

Similar to HPC, Chen et al. proposed CC-Hunter [15], which adds event monitoring logic in shared hardware and counts whenever one process uses specific hardware resources that are already occupied by another process. It allows the detection of Covert Timing Channels (CTCs), which use the leak of microarchitectural timing information to secretly communicate between different processes. PMUe [16] proposed the detection of ROP attacks based on modified HPCs. It contains a hardware logic recognizing short instructions separated by branch misprediction events, which is a typical ROP pattern.

However, existing hardware event counters approaches only focus on counters in a small and predefined list, without discussing a systematic approach to evaluate various microarchitectural events and find what events can efficiently contribute to detect attacks.

2.2. Instruction Tracing

Arm CoreSight [17] is another microarchitectural monitoring technology available for off-the-shelf processors. Arm CoreSight is a debug and trace technology inside Arm's System-on-Chip (SoC). Its Program Trace Macrocell [18] collects instruction traces by means

of tracing key instructions, such as branch, exception, and synchronization. These traces can be accessed via the bus or debug port. To the best of our knowledge, there is no microarchitectural attack detection using Arm CoreSight but some papers mention its use for research works in Dynamic Information Flow Tracking (DIFT) [19] and code reuse detection [20]. The proposed approaches consist of capturing representative traces inside the processor, then decompressing and analyzing these traces inside the FPGA. Compared with hardware event counters, Arm CoreSight provides much fine-grained information about the software's execution, and the analysis can proceed in parallel inside the FPGA to reduce the execution time of the detection process. However, limited by the bus structure, the FPGA can only obtain little information when running continuous analysis.

Similar hardware monitoring techniques that target the fine-grained instruction and data trace of the processor have also been proposed in different academic research works. Several tag-based monitoring techniques for DIFT have been notably proposed. The Raksha architecture [21] consists in modifying the pipeline to include tag related memory, tag propagation, and verification logic, while FlexCore [22] and Harmoni [23] use a dedicated component. Trace-based monitoring techniques have also been proposed to allow collection and analysis of the execution trace. LBA [24] performs the analysis on a general-purpose core; PHMon [25] performs the analysis on a coprocessor; while REHAD [26] describes a solution to detect Flush + Reload attack by means of a dedicated component. Among these works, only FlexCore and REHAD discuss the use of reconfigurable hardware for flexible detection logic structure, and only REHAD targets microarchitectural attack detection. However, even in REHAD—the detection algorithm design phase—the generality of the approach for several classes of attacks and an exhaustive evaluation of the attack detection are not provided.

2.3. Hardware Signal Probing

To observe more fine-grained arbitrary signals inside the SoC, hardware signal probing techniques, such as Xilinx ChipScope [27] or Intel SignalTap [28] can be used. Such probes can be directly connected to the chosen signal: sample signal values at each clock cycle, store them in a buffer, and send this buffer via debug port when a trigger condition is reached. Such probes are commonly used to debug the hardware, where debug information is received and analyzed by another system (Host). Although it benefits from fine-grained and flexible monitoring capabilities, the data transfer throughput is still limited by the debug port. Hardware signal probing techniques can be a solution for passive off-chip analysis, but are not suitable for the continuous monitoring of large amount of fine-grained signals. Perhaps due to this reason, we do not have knowledge of any application of this technology for security purposes.

However, some less flexible propositions—such as the monitoring of a small amount of selected microarchitectural signals directly in hardware—have been proposed to enforce the security of a specific hardware component. For instance, LiD-CAT [29] monitors cache-related signals and performs detection with a state machine constructed based on a formal description of cache information leakage attacks.

The study of the current research works shows, in our opinion, that there is a real need for a more open and free framework that can monitor a large amount of microarchitectural signals in real time, based on real software and hardware, to design and evaluate different detection techniques for some specific attacks. MATANA is designed to fill this gap. A comparison of MATANA with previously presented solutions can be found in Table 1.

Table 1. Comparison of different attack detection systems based on the analysis of microarchitectural signals. *Reconfigurability* is classified *high* if some reconfigurable fabric is used, *limited* if some configuration bits that change the hardware's behavior exist, and *no* if the reconfigurability is not possible or not discussed. Note: CTC = Covert Timing Channel, CSCA = Cache Side-Channel Attack, ROP = Return-Oriented Programming, DIFT = Dynamic Information Flow Tracking.

	Implemented Protection against		**Monitored Signals Type**	**Reconfigurability**	**Methodology Provided**
	Microarch Attacks	**Control-Flow Attacks**			
HPC-based [10–14]	unspecific	no	event	limited	no
CC-Hunter [15]	CTCs detection	no	event	limited	no
PMUe [16]	no	ROP detection	event	no	no
Wahab et al. [19]	no	DIFT	instruction	high	no
Lee et al. [20]	no	ROP detection	instruction	high	no
Raksha [21]	no	DIFT	instruction + tag	limited	no
FlexCore [22]	no	DIFT	instruction	high	no
Harmoni [23]	no	DIFT	instruction + tag	limited	no
LBA [24]	no	DIFT	instruction	no	no
PHMon [25]	no	DIFT	instruction	limited	no
REHAD [26]	CSCAs detection	no	instruction	high	no
LiD-CAT [29]	CSCAs detection	no	cache internal	no	no
MATANA	unspecific	unspecific	any	high	yes

3. Overview of MATANA

This section provides a detailed overview of our methodology to detect attacks by characterizing their microarchitectural fingerprint using MATANA. We first introduce the assumptions and threat model considered in the paper; then, we describe the required modifications of the target system (both software and hardware) and present our iterative methodology to design detection algorithms suited for detecting some specific class of attacks.

3.1. Assumptions and Threat Model

We consider an attacker with regular user privileges. The threat model considered is the following: (1) the attacker does not have any physical access to the system; (2) the attacker can execute any userland attack and victim process; (3) the attacker has full control of the attack process; (4) the attacker cannot directly read the memory content of the victim process but can bypass interprocess isolation mechanisms using microarchitectural information. The detection algorithms are implemented both in hardware and software (inside the target system kernel). As a consequence, we also make the assumption that the kernel of the target system is trusted. As we assume that the attacker does not have any physical access to the system, the nonintrusive side-channel attacks based on physical access—such as power measurement analysis—are out of the scope of this paper, as they do not modify the system state and their behavior stays invisible at the microarchitectural level. Fault injection attacks are also out of the scope of this paper, as they could damage the hardware detection logic (such as laser beams that could create bit flips in the detection logic).

3.2. MATANA Design Flow

Figure 1 presents the design flow to integrate MATANA into a target system. MATANA follows a white box approach, where a precise description of the microarchitecture is required as well as a description of the target attack that must be detected. The design flow is split into four main steps:

1. From the description of the microarchitecture of the target and the description of the attack, selection of the relevant internal signals that are modified when the target attack is performed, and design of the associated detection algorithm. More details on the methodology are provided in Section 3.3.

2. Patch of the target microarchitecture to output the signals (identified in step 1) towards the Detection Module. This Detection Module must also be connected to the system bus to communicate with software and connected to the processor interrupt system. More details are provided in Section 4.
3. Patch of the system kernel to allow communications from/to the Detection Module and the handling of interrupt requests.
4. Integration of the patched software and hardware into the final target system.

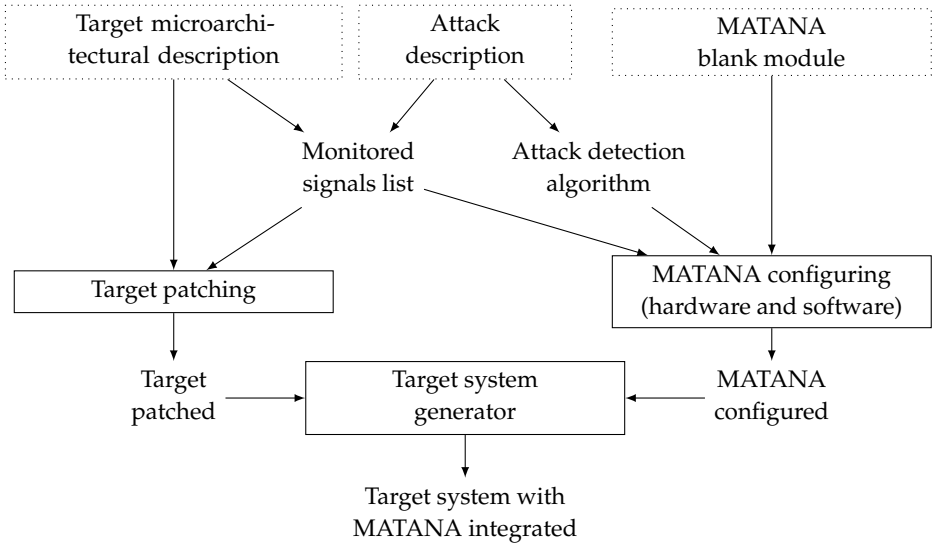

Figure 1. Design flow for integrating MATANA into a target system.

3.3. Methodology for Designing Attack Detection Algorithms

Figure 2 describes our iterative methodology to design and implement attack detection using MATANA. The methodology is composed of five steps:

1. **Signal Selection**. This step aims at selecting the most appropriate monitored signals for attack detection. As the methodology is iterative, this selection may be refined at each iteration. At first, a large number of different signals can be chosen, ranging from some generally useful signals such as the instruction flow (since microarchitectural attacks are based on relatively small instruction patterns) to some attack specific signals such as cache and memory events. For the following iterations, the selection of monitored signals may be adjusted according to the experimentation results in step 4 by removing some less-efficient signals or adding some related signals.
2. **Probe Connection**. The probe connection step consists in choosing a proper clock domain crossing technique used for connecting each selected signal to the Detection Module, with consideration of information quantity, area overhead, and detection accuracy. This step is only required if the Detection Module is not running at the same frequency as the processor. This distinction is important since, in the case of a reconfigurable Detection Module, the frequency may be much smaller that the processor's core. We illustrate the detection method at low frequency in Section 5.2. For more information regarding transmission techniques, the reader can refer to Section 4.3.
3. **Detection Logic Design**. This step aims at designing the detection algorithm intended to be implemented into the Detection Module. The inputs of the algorithm are the signals selected in step 1, and the output is an alert when an attack occurs. The algorithm itself can be designed based on the expected behavior of target components, the properties of target attacks, or be adapted from any previously known detection algorithms. Thanks to the fast reconfigurability offered by MATANA, a "trial and error" methodology is possible to investigate different detection methods for a given attack. Since no assumption is required on the type of algorithm, simple techniques

such as heuristic-based pattern matching to more sophisticated ones based on state machine or machine learning can also be evaluated.

4. **Benchmarking**. This step aims at carrying out some experiments to assess the relevance of the attack detector (with signals, connections, and algorithms defined in previous steps) by means of some appropriate benchmarks. For this purpose, MATANA is equipped with an Operating System (OS) aimed at running sufficient benchmarks to evaluate the accuracy of the detection. In our approach, we chose one benchmark that represents an effective attack, and the others were picked from state of the art benchmarks to cover the majority of applications, i.e., mainly CoreMark and Embench. Let us note that without any benchmark, MATANA can also evaluate the stability of the detector since the OS is always running in background. If the detection logic implemented exhibits too many false positives or false negatives, it is necessary to go back to step 1 or 3 and to improve either the selection of signals or the design of the detection logic itself.
5. **Deployment**. When the detection logic exhibits a good detection rate, a final step consists in optimizing the solution for its deployment in final products. To our viewpoint, the best choice for the final product is a reconfigurable Detection Module for detection flexibility, with fixed monitored signals integrated into the target component in order to reduce performance impact. For a Detection Module that is supposed to detect multiple attacks, the size of reconfigurable fabric should be sufficient to contain at least each algorithm and ideally multiple algorithms at the same time. For that purpose, some additional place is needed to allow the evolution of detection algorithms and the monitored signals should be the concatenation of all required signals. Other classical optimization must also be done in this step, such as reducing hardware resource utilization, optimizing for frequency and power consumption, and revisiting the performance impact of software support, while maintaining adequate detection efficiency.

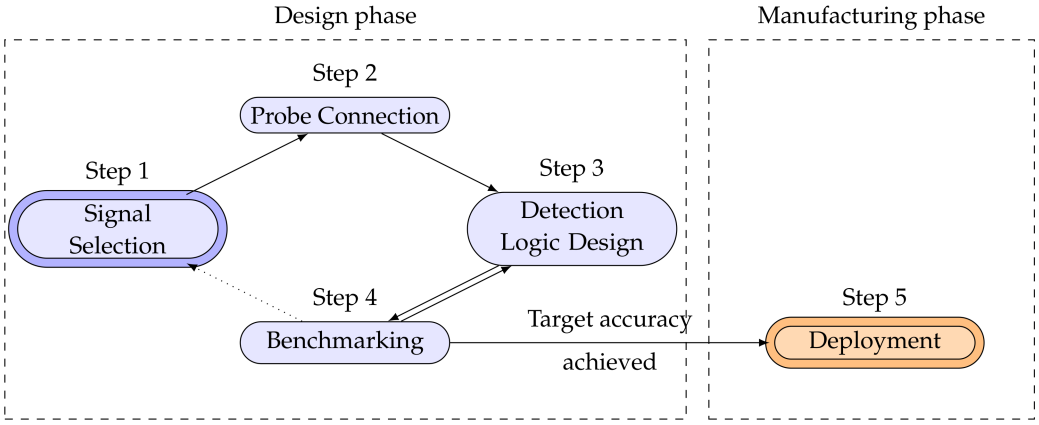

Figure 2. Methodology for attack detection using MATANA.

MATANA can also serve as an experimentation platform to evaluate the effectiveness of attack mitigation for a given hardware component (ideally a soft IP core) in a real execution environment. For example, it is possible to integrate a new cache within the SoC and analyze the correlation between the microarchitectural signals and some victim secret using steps 1 to 4.

4. MATANA Detailed Architecture

MATANA architecture is presented in Figure 3. MATANA is built on top of a target system and is composed of one or multiple probes, a Detection Module, and connections.

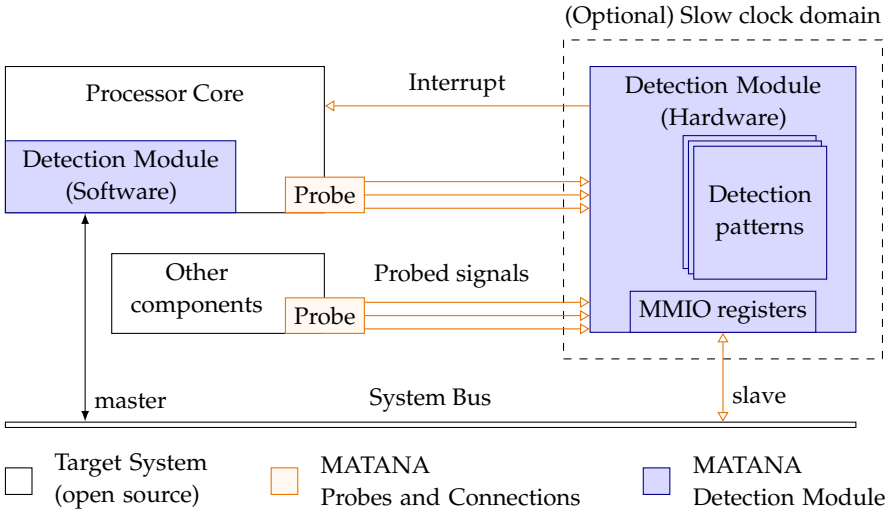

Figure 3. MATANA architecture.

4.1. Hardware Architecture

Probes are integrated on target components that are supposed to be monitored in order to extract their internal signals' values (called probed signals in the following). The target components must allow additional connections. The presence and placement of these probes is crucial in the architecture because the quality of the detection highly depends on the analysis of these signals. If the different probes are adequately placed on the target components, they can provide a large cycle-accurate data set that gives a precise view of the system state, which is essential to build appropriate detection logic.

The Detection Module is a combination of hardware and software, and is the core component in which the detection algorithms are implemented. It has been designed as a stand-alone module at both the hardware and software level, for better modularity of the system. The hardware part of the Detection Module (called hardware Detection Module in the following) is implemented in a fully reconfigurable fabric to perform different custom runtime analyses on probed signals. It is equipped with a set of Memory-Mapped I/O (MMIO) registers and memory zone, and provides its intermediate results to the software part of the module (called software Detection Module in the following) located in the system kernel. Additionally, the hardware Detection Module can send interruptions to the processor by means of the interrupt link, in case the hardware requires an immediate intervention of the software—for example, when an attack is detected. It is up to the software Detection Module to decide whether it is a real attack and what action to take, as it benefits from a more global and accurate view of the execution environment. We did not add any other signal from the Detection Module to the target components, because this would require modification of the target components so that they can handle these signals.

By default, the probed signals are directly connected to the hardware Detection Module. This dedicated Probe Connection (which does not require any shared bus) guarantees high throughput and low-latency data transfer that is unaffected by other system activities. However, when the hardware Detection Module is configured in a slower clock domain, other Probe Connections are needed, as presented in Section 4.3.

For the flexibility in the design phase, i.e., to allow the evaluation of different detection algorithms based on different probed signals, as described in Section 3.3, the target components, hardware Detection Module, and its connections are located on a reconfigurable fabric such as an FPGA. Other parts of the target system may be outside of it, as long as the synchronization is properly configured. Let us underline that this framework is independent of the ISA, which means that many systems are compliant with MATANA as long as (1) the target components can output some internal signals and (2) access to a system bus and an interrupt link are provided to the reconfigurable part.

The design phase is intended to easily assess different detection algorithms. At the end of this phase, i.e., when some relevant detection algorithms are identified and adopted, the integration of these algorithms into final products (manufacturing phase, as described in Figure 2) only necessitates that the hardware Detection Module and some specific required I/O are located in the reconfigurable fabric to preserve flexibility (as shown in dotted rectangle in Figure 3). This configuration allows the integration of probes into high-frequency target components, thus providing a balance between design effort, SoC performance, and detection flexibility for future attacks. The experiments we carried out seem to confirm this choice, as we show in Section 5 that we are able to design efficient detectors with few and standard signals for two completely different classes of attacks. Let us note that the set of probed signals may evolve according to the processor's version to include new relevant signals and increase the accuracy of the detection. Anyway, it remains possible for previous versions to build a detector for new attacks (even if not perfect), thanks to the implemented probes available in this version, by purposely reconfiguring the detection algorithms implemented in the Detection Module.

4.2. Software Support

The hardware Detection Module may be used alone in order to collect and analyze the probed signals and provide results to the outside, using the debug port, for further offline analysis. This would be very similar to hardware signal probing techniques but slightly better because the analysis of the different signals enables the output of more meaningful information. However, hardware-only analysis is too restrictive to our viewpoint because (1) some detection logics are quite difficult to implement in hardware; (2) it cannot benefit from important information that can be retrieved from the global state of the runtime environment; and (3) when an attack is detected, the hardware Detection Module can only perform limited actions defined by the physical logic implemented, which may be insufficient for a complex system. For that purpose, we argue that the software part in the Detection Module of MATANA framework is important.

The software Detection Module is located in the system kernel, and, as mentioned in the threat model, is considered trusted. Its main functionalities consist of (but are not limited to) the following: (1) providing additional high-level information to the hardware Detection Module, such as the current Process Identifier (PID); (2) performing runtime data analysis that is not easily feasible or too costly at the hardware level; (3) dynamically adjusting the configuration of the hardware Detection Module by setting values to MMIO registers; (4) handling interruptions sent by hardware Detection Module; (5) performing postrun data collection and analysis.

Let us note that this framework does not require any modification on userland programs executed (either benign or malicious), e.g., to include tags or special instructions. It guarantees that MATANA is compatible with existing software environments and, as only few modifications are required at kernel level, it can be widely used even for nonembedded applications.

A typical use case scenario of MATANA is as follows: the hardware Detection Module processes probed signals with information and the configuration sent by the software Detection Module and places the analysis result into the MMIO registers. Then, by means of an interruption or a polling mechanism, the software Detection module can fetch the MMIO register's content and perform a more powerful analysis.

4.3. Probe Connection Types for Low-Frequency Attack Detection

The hardware Detection Module can be integrated into the target components and share the same communication channels to the software. However, when deploying MATANA on the final system, placing the hardware Detection Module separately from the rest of the system for better modularity, using a reconfigurable fabric, and benefiting from the flexibility on detection algorithms implemented are important to deal with attacks that are constantly evolving. For that purpose, we have to consider that the hardware Detection

Module may be asynchronous and run at a lower frequency than the target components. In that case, the Detection Module's I/O must be properly synchronized. For that purpose, we propose several synchronization approaches for probed signals that aim at reducing the hardware logic complexity while preserving sufficient information transmitted for attack detection.

The proposed Probe Connection types are shown in Figure 4, with a target component's frequency *divN* times higher than the hardware Detection Module's frequency:

A *Parallel Connection* collects all signals in *divN* adjacent cycles and sends them in parallel to the hardware Detection Module. This connection type maximizes the preservation of the information collected and pinpoints the state of the signals at a specific time, but requires more synchronization logic and internal logic to process it.

A *Preprocessing Connection* preprocesses and buffers the probed signal, then sends the processed data to the hardware Detection Module. It provides a balance between the hardware resource utilization and the amount of useful data transmitted, and can reduce analysis effort in the hardware Detection Module. For example, this logic can count some hardware event occurrences or extract the instruction traces, and provide the possibility to implement existing detection solutions based on a hardware event counter or instruction tracing.

An *Extraction Connection* is a special and simple case of Preprocessing Connection that picks one interesting signal in a window of *divN* samples. It can be used where the signal value rarely changes or where some loss of data is acceptable with respect to the accuracy of the detection. It preserves the time of occurrence of events, which is useful for detecting time-based information leakage in microarchitectural attacks, and this property is not guaranteed by the use of a simple FIFO.

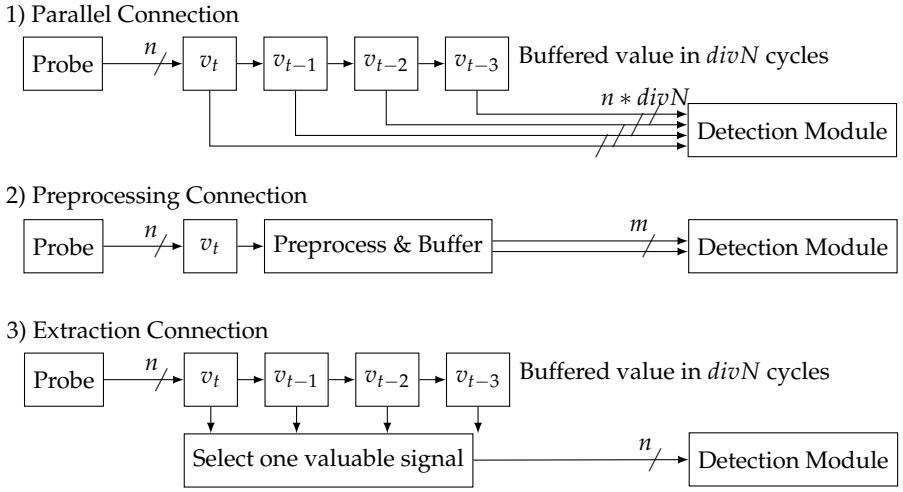

Figure 4. Probe connection types for low-frequency hardware Detection Module.

Let us note that these connection types can be mixed, i.e., different signals from the same probe can use different connection types.

5. Use Cases of Attack Detection with MATANA

In this section, we present our experimentation platform and two use cases of attack detection using MATANA. The detection strategies are constructed following the steps described in Section 3. The first use case considers a common microarchitectural timing attack—the so-called Prime + Probe; the second use case considers an attack altering the control flow, i.e., ROP attack. We demonstrate that, with quite simple heuristics, our methodology allows the identification of a malicious behavior even in the case of a complex system equipped with an OS executing a wide range of typical activities.

5.1. Platform Setup

A prototype of MATANA has been implemented on a Xilinx ML605 evaluation board equipped with a Xilinx Virtex-6 FPGA running a Linux kernel 4.15.

We chose to build our target system with the Chipyard SoC generation framework v1.3 [30], which contains a RISC-V ISA softcore processor Rocket [9], since it is open source, actively maintained, and easily reconfigurable. Rocket is an in-order processor core, we configured it to 64 bit, medium size (with support of virtual memory, no Floating-Point Unit (FPU)), an L1 instruction cache, and an L1 data cache).

Figure 5 presents the hardware structure of the target system. The hardware part of MATANA is completely integrated in Chipyard (represented in blue color in Figure 5). To ease the integration process, all MATANA-related files have been written in chisel [31], a Scala-based hardware design language used by Chipyard that can be compiled to Verilog code, especially to ease the signals connection inside the target system and be still compatible for different SoC parameters. The probing is mainly done inside the Rocket core to monitor the processor's internal signals, such as the instructions executed, the memory addresses accessed, and some hardware events.

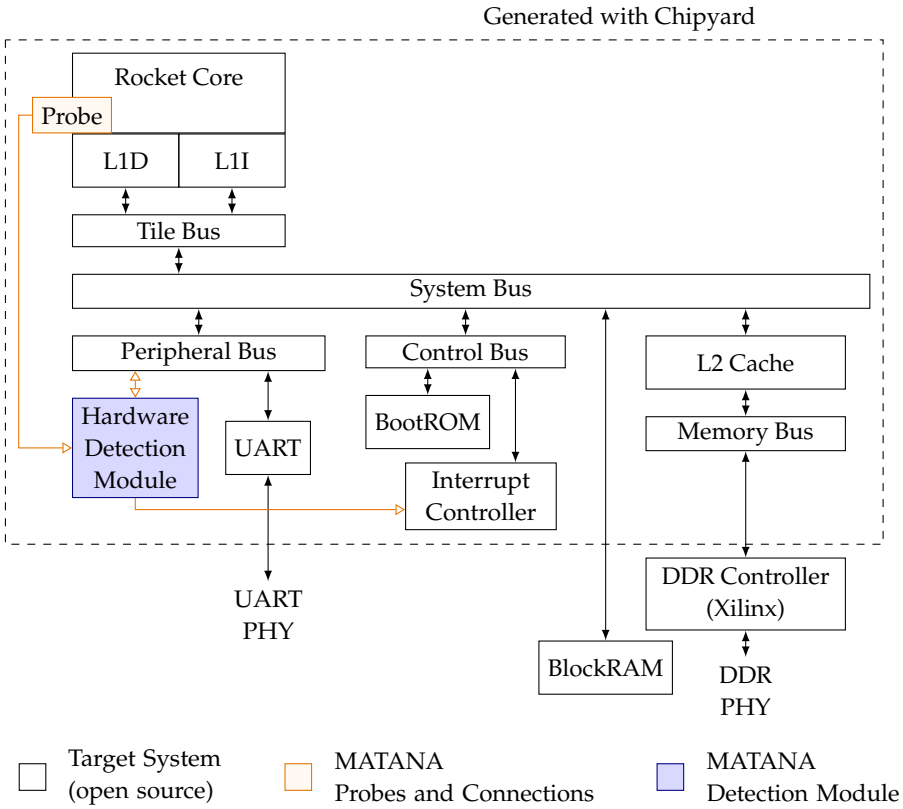

Figure 5. Hardware architecture of MATANA prototype implemented on FPGA.

We also made several choices to ease and increase the accuracy of the evaluation process: (1) The detection logic has been implemented as a userland program for testing purposes, as it allows the test of different software implementations without regenerating the kernel image and rebooting the system; it is intended to be integrated into the Linux kernel after the testing phase in order to benefit from the protection at kernel level. (2) A Linux kernel module has been developed to manage the communications between the hardware and software Detection Module. (3) The Linux kernel has been patched to automatically capture the current PID at context switch.

The Linux kernel binary was generated based on the freedom-u-sdk project v1.0 [32], version 4.15, with integration of MATANA. The software part of our prototype was compiled with GCC toolchain version 9.2.0 for cross compilation; the toolchain was compiled

with the option `rv64imac` since the hardware part of our target system does not have any FPU.

5.2. Use Case: Prime + Probe Cache Attack Detection

In this section, we present the application of our methodology to detect a practical Prime + Probe attack using MATANA.

5.2.1. Technical Background

Memory cache is the hardware component that is designed to speed up the processor's access to the main memory. It is built on the temporal and spatial locality principle of memory access, i.e., the same location and the nearby location of a recently accessed location is likely to be used again in the near future. If data access is required by the processor, and the data are not in the cache, the corresponding data and nearby data are read from the memory and loaded into the cache for likely future access (so-called a cache miss). If the data is in the cache, the access is quicker (so-called a cache hit). The correspondence between the main memory and the cache is usually based on physical addresses, as shown in Figure 6. For the L2 cache of our prototype, the lower bits of the address correspond to the offset in a cache line (the basic unit of the cache, usually 64 bytes), the medium bits of the address correspond to the cache set index (64 sets), and the higher bits of the address constitute the tag used to determine if a given cache line contains data from this specific memory address location. One set can have multiple cache lines; for example, our L2 cache is 8-way associative, which means that each cache set contains 8 cache lines. One address that is mapped to a given cache set can then be placed on one of the 8 cache lines based on a replacement policy, for example, Least Recently Used (LRU).

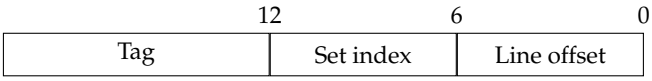

Figure 6. Cache indexing for 64-set cache and 64-byte cache line.

Prime + Probe is a powerful CSCA that benefits from this access acceleration property of the cache. It only requires that the attacker's process and the victim's process share the same physical cache at the same time. For example, processes running on different CPUs of the same system that share the Last Level cache can satisfy this requirement [33]. The global attack works in three phases: the preparation phase, the attack phase itself, and the analysis phase.

In the preparation phase, the attacker needs to locate one or multiple cache sets of interest to monitor and, for each set, find 8 addresses (for a 8-way associative cache) that are located in the given set but with different tags. The attack phase's assembly code is shown in Listing 1. The first step is called Prime, in which the attacker accesses in turn the 8 addresses previously identified. Due to the replacement policy of the cache, all 8 cache lines of this selected set are occupied by the attacker's data. Then, in a second step, so-called Wait, the attacker's process waits for the victim process to execute. During this step, if the victim uses the set, some victim's data are loaded in the set and evict the attacker's data. The third step is so-called Probe, in which the attacker accesses the 8 previously identified addresses and measures the total access time. The access is arranged in the form of a linked list (i.e., the value of the next address is stored in the contents of the previous address) and in inverse order of Prime phase to prevent the prediction of accesses and the self-replacement due to LRU. If the victim process did not use the set monitored by the attacker in the Wait step, then all the 8 attacker's data are still in the cache, and access to these data generates 8 cache hits, which corresponds to a fast access. If the victim process used this set, then some attacker's data were evicted and the access to these data generates a cache miss, which corresponds to a slow access. By repeating Prime, Wait, and Probe (Probe can be served as a Prime for the next round of Prime + Probe), the attacker learns a sequence of accesses of the victim process on monitored sets, which allows them to deduce some of the victim's secret information in the Analysis phase.

Listing 1. One round of Prime + Probe operation on RISC-V.

```
    rdcycle a2
    ld      a5,0(a5)      ; Prime: 8 accesses
    ld      a5,0(a5)
    ld      a5,0(a5)
    ld      a5,0(a5)
    ld      a5,0(a5)
    ld      a5,0(a5)
    ld      a5,0(a5)
    ld      a5,0(a5)
    rdcycle a4
    ...                   ; Wait for victim to execute
    rdcycle a2            ; Measure access time
    ld      a5,0(a5)      ; Probe: 8 accesses
    ld      a5,0(a5)      ; Can serve as Prime for next round
    ld      a5,0(a5)
    ld      a5,0(a5)
    ld      a5,0(a5)
    ld      a5,0(a5)
    ld      a5,0(a5)
    ld      a5,0(a5)
    rdcycle a4            ; Measure access time
    ...                   ; Next round of Prime + Probe
```

CSCAs can also be used as part of other attacks. For example, Spectre and Meltown attack expose information leaked from other microarchitectural side channels to the cache, then use CSCA to read the secret information; rowhammer [34] attack needs to frequently evict memory lines out of the cache, which corresponds to the Prime phrase of CSCAs, in order to perform direct access to main memory. For these reasons, we argue that providing detection solutions for CSCA is particularity valuable.

5.2.2. Determination of a Suitable Attack Benchmark

The validation phase (i.e., step 4 in our methodology) requires a suitable attack benchmark parallel to more common benchmarks. In addition to a set of schoolbook implementations of Prime + Probe, we would like to assess that our detection algorithm is able to secure the execution of a sensible application by detecting Prime + Probe attack at runtime. In our case, we focused on the Prime + Probe attack of the last round of AES described in [2] and implemented the attack on our target system with the help of an adapted version of the Mastik tool [35]. This attack is applied to the C implementation of AES in OpenSSL 1.1.1k, which is based on the well-known T-Table optimization. This does not reduce the generality of the approach since the Prime + Probe gadget remains but is integrated into a more sophisticated attack.

To test different attack scenarios, we implemented two versions of this attack: (1) a *SingleProcess* version where the attack code and the victim code run in the same process; (2) a *SharedMemory* version where the attack code and the victim code run in separate processes and exchange plaintexts and ciphertexts via a shared memory. In both versions, the attack works as follows. First, the attacker performs a Prime on the cache. Second, the attacker provides a random plaintext to the victim and waits until the victim provides the encrypted plaintext. Third, the attacker checks the cache with a Probe. Notice that with our setup, the full key recovery is possible with 3000 different random plaintexts for the *SingleProcess* version.

For the schoolbook Prime + Probe benchmark, we implemented the standard cache line profiling, i.e., loop of a Prime—an access to a specific address and a Probe (which also acts as a Prime operation). There are three versions of this program, one that probes only set 0, one that probes set 1, and one that probes all 64 sets.

5.2.3. Design of the Detection Algorithm

In this section, we present the different heuristic patterns that we identified for Prime + Probe attack detection by following the aforementioned methodology described in Section 3.3. For this experiment, the reconfigurable hardware Detection Module is set at 1/16 of the processor's frequency. The clock cycle of the hardware Detection Module is called *DM cycle* in the following. For each benchmark (including attack-related ones), the following steps were followed: (1) reset the internal state of the detection module and start monitoring; (2) execute the program under test; (3) stop monitoring and collect information stored in MMIO registers. To evaluate the impact of the experimental setup, we also executed the setup with no program executed, and to test the impact of the OS, we executed the standard system call `sleep` as in other benchmarks. During the run, the software Detection Module sends the PID at context switch to the hardware Detection Module to help identify kernel-related activities.

The benchmarks are built by means of the same toolchain described in Section 5.1. The benchmarks include (1) all 3 CoreMark runs, (2) all 11 CoreMark-PRO programs for single context, (3) all 19 Embench programs with time measurement using `rdcycles`, (4) the Dhrystone benchmark, and (5) the Stream benchmark with `DSTREAM_ARRAY_SIZE=32768`. These benchmarks are selected in order to represent a large range of common algorithms that make use of the CPU and the main memory.

1. **Signal Selection.** Three categories of signals seem relevant to detect Prime + Probe attacks. First, instruction-flow-related signals, i.e., the instruction currently executed as well as the instruction valid signal in the processor's pipeline are useful to identify instruction patterns. Second, as cache side-channel attacks perform a lot of carefully chosen memory access, the signals associated to the memory access seem relevant. We chose the virtual address sent in the processor's pipeline and the physical address obtained in the L1 data cache. Third, inspired by some detection techniques based on HPCs proposed in several research papers [12,14], several cache events (cache miss, cache access, and cache TLB miss) are also relevant.
2. **Probe Connection.** The instruction signals are collected by means of a Parallel Connection that is more suited in order to preserve the maximum value and time information. The address and the cache event signals are collected by means of an Extraction Connection, in such a way that we only keep one single valid address and one single cache event per *DM cycle*. This choice was purposely made because the memory accesses take multiple cycles in a cache hit and even more in a cache miss. This was confirmed by our experimentation, showing that one value per *DM cycle* is enough to preserve sufficient information for detection.

3/4. **Detection Logic Design./Benchmarking.** Prime + Probe attacks are closely linked to the time measurement, which can be performed by means of various instructions that are dependent on the target processor ISA. By analyzing the code presented in Listing 1, we note that there are always two time measurement instructions (so-called timer instructions in the following) relatively close to each other used to measure the time to access 8 memory addresses. To identify this pattern, we developed two different heuristics: *InstTimer*, which analyses the use of specific instructions; *MemAccess*, which tries to identify patterns in memory access. For evaluation purposes, each heuristic has been associated to a counter to evaluate the occurrence of the corresponding pattern during the benchmark's execution. Table 2 provides the evaluation of the two heuristics for the proposed benchmarks.

For *InstTimer*, we set a detection window and count the use of specific timer instructions within. The detection window starts at the first detection of a timer instruction; if another specific instruction occurs during the window, we increment the *InstTimer* counter. For the Target processor ISA (i.e., RISC-V), the set of instructions are `rdcycle`, `rdtime`, and `rdinstret`. As we can see in Table 2, this approach is very effective and does not exhibit false positives.

Table 2. Evaluation of the Prime detection using the instruction-flow-based (*InstTimer*) and memory-access-based (*MemAccess*) pattern counters. For each program, the column *nb. of detection* is the counter value at the end of the execution, and *mean period* is the mean cycles between two counter's increment. Cycles are expressed as *DM cycles*. Sets of benchmarks are noted with * and only the mean value is represented.

Program	Cycles	InstTimer		MemAccess	
		Nb. of Detection	Mean Period	Nb. of Detection	Mean Period
CoreMark *	53,452,817	0	N/A	161	332,005
CoreMark-PRO parser	67,632,009	0	N/A	29,174	2318
CoreMark-PRO others *	1,163,695,064	0	N/A	7854	148,166
Embench *	43,957,696	0	N/A	162	271,344
Dhrystone	11,242,585	0	N/A	175	64,243
STREAM	40,513,182	0	N/A	140	289,380
Attack SingleProcess	108,337,476	572,429	189	1,339,213	81
Attack SharedMemory	724,951,250	553,060	1311	926,154	783
Cache Profiling set 0	976,975	39,985	24	118,522	8
Cache Profiling set 1	816,452	39,985	20	116,022	7
Cache Profiling all set	15,594,885	1,299,297	12	5,294,473	3
No program	112,200	0	N/A	77	1457
Sleep 1	168,206	0	N/A	96	1752

For *MemAccess*, since the number of cache lines in a set is 8 in our implementation, the pattern detection tries to identify the presence of 8 consecutive memory accesses typically used in Prime. More precisely, we extract from the memory address the cache tag and cache set index, and increment the counter when the target set has been recently used but not the tag. For this second heuristic, we also obtained interesting results: as it can be observed in Table 2, this heuristic exhibits some false positives, but the occurrence of the pattern is much more important in case of an attack. In addition, the mean number of cycles between two increments of *MemAccess* is much shorter than for *InstTimer*, which means that *MemAccess* is also very sensitive. This can be an interesting approach when, for instance, an untrusted black box software must be executed and we want to carefully analyze its behavior related to cache attack. Let us note that:

- Regarding *InstTimer*, the timer-instruction-based pattern may be limited when cache attacks are not based on such timer instructions, but adopt a less-accurate timing measurement strategy by using other mechanisms such as a kernel library or a counter in a loop in a different core. To cover this kind of attack, it is possible to check the physical address of the library function (provided by the software Detection Module) or the loop counter (found using instruction-based or address-based pattern matching) and look for access to these suspicious addresses instead of timer instruction.
- Some attacks, such as rowhammer, only use cache eviction and do not need any time measurement. Thus, the timing-instruction-based approach is not effective and the memory-access-instruction-based approach should be investigated instead.
- The *MemAccess* is indirectly an estimation of the misuse of the cache. Basically, accessing data within a specific set but with addresses with different tags very often breaks both temporal and spatial locality principles. As a consequence, an investigation on the adaptation of software to carefully use the cache would be an interesting approach to take advantage of the sensitivity of *MemAccess* to secure the execution, especially for critical systems that have dedicated software.
- We also tested other heuristics, such as detection strategies based on cache event occurrence per *DM cycle* or based on consecutive load instruction to access a list of address. However, as these detection patterns exhibited false positive and negative rates that were not satisfying, they were not sufficiently relevant to discriminate

between attack benchmarks and normal benchmarks, and are not presented here. Let us underline that we took advantage of MATANA's flexibility to quickly assess the accuracy and precision of given approaches, which is the strength of our methodology.

5.2.4. Area Footprint

Table 3 provides the area footprint of the solution in 3 scenarios: *InstTimer* only, *MemAccess* only, and *InstTimer + MemAccess*. The area of the unprotected SoC is presented for comparison purposes. The full implementation (i.e., *InstTimer + MemAccess*) basically increases the number of Flip-Flops used by 12%, and the number of LUTs by 5%. The main area overhead is due to the synchronization for the Detection Module connections and the Parallel Connection required by *InstTimer* for the instruction flow signals. Some room for improvement is possible by revisiting the synchronization strategy and detection algorithms. We can also note that the combination of two completely different detection logics requires only 58% Flip-Flops of the sum of the two. This is due to the fact that many parts of the detection logic can be shared, especially the bus connection and some configuration registers.

Table 3. Area footprint of MATANA for Prime + Probe.

Resources	Base SoC (Unprotected)	InstTimer		MemAccess		InstTimer + MemAccess	
		Module Alone	Total	Module Alone	Total	Module Alone	Total
Flip-Flops	22,167	2448	24,615	2172	24,339	2669	24,836
LUTs	31,668	1620	33,288	946	32,614	1354	33,022
BlockRAM	26	0	26	0	26	0	26

5.3. Use Case: Return-Oriented Programming Attack Detection

The MATANA framework is relevant to analyze and detect other kind of attacks insofar as they exhibit some specific microarchitectural behaviors. In this section, we present a use case in which we successfully built a detection logic for ROP attacks in the MATANA framework.

5.3.1. Technical Background

ROP [36] is an example of code reuse attacks. It manipulates return addresses to redirect the control flow to a series of small pieces of assembly code (called gadgets) chosen by the attacker from the target executable code itself or shared libraries. These gadgets usually execute elementary actions, such as performing a computation on a register, and end with a return instruction. This return instruction jumps to the caller function based on a return address previously provided by the caller function; however, as the attacker controls these return addresses, the return instruction actually jumps to a specific address chosen by the attacker to continue the execution. The gadgets are run just as if they form part of the victim program—with the same privilege, in the same memory zone. Unlike traditional buffer overflow attacks, code reuse attacks do not write the code to be executed into the memory, thus circumventing the protection of the type "Write xor Execute", which disables the executable stack. Other variants of ROP also exist, which do not use return addresses but function calls or indirect jumps to link gadgets.

ROP attacks can be carried out in RISC-V platforms [37]. Unlike in x86, the return address in RISC-V is not read from the stack, but from a register called `ra`. When calling a function, the caller puts the current address into `ra`; when the function returns, it executes the instruction `ret` (alias of `jalr zero, 0(ra)`), which jumps to the address pointed by `ra`. In case of nested function calls, the content of the `ra` register is pushed on the stack and restored when necessary. This allows an attacker who controls the stack to also control the return address of the gadget in the function. One sample gadget on RISC-V is shown in Listing 2.

Listing 2. Example of a gadget in RISC-V built with our ROP generator: this simple gadget adds 1 to the register a7 and can be used, for example, to prepare the value of a7 during a ROP chain execution.

```
gadget_a7plus1:
    addi a7, a7, 1    ; Elemental action
    ld ra, 0(sp)      ; Read the next gadget address from the stack to ra
    addi sp, sp, 8    ; Increment the stack pointer
    ret               ; Jump to the next gadget address stored in ra
```

5.3.2. Determination of a Suitable Attack Benchmark

As there has been no ROP attack benchmark published on RISC-V so far, we followed [37] and wrote a Python script to generate ROP of various lengths and types of instructions in C and assembly code. All generated attacks aim at spawning a shell, i.e., call `execve("/bin/sh", {"/bin/sh"}, {NULL}, {NULL})`. For that purpose, different registers must be set before calling the `ecall` instruction: `a7` is set to 221 (which identifies the `execve` syscall), and `a0`, `a1`, and `a2` are set according to the parameters of the syscall.

For the sake of simplicity, the ROP generator does not search and try to execute a ROP chain in a given binary but instead generates gadget-style code that is directly injected into the target executable. A `memcpy` is called to write the payload full of gadget addresses in the stack and to overwrite the original return address. Thus, we are able to generate ROP attacks with chains of gadgets of different lengths (number of returns called), with different numbers of gadgets and different methods to set the registers. The actual gadgets are spread in different locations of the memory to mimic the behavior of real attacks that use different gadgets in different memory locations.

5.3.3. Design of the Detection Algorithm

For this use case, the hardware Detection Module is synchronized with the processor, which means that the *DM cycle* is equal to the processor cycle. This configuration allows us to simplify some part of the detection algorithm while the methodology can be adapted to slower frequencies.

1. **Signal Selection.** As we were confident in the fact that ROP attacks can be detected mainly by analyzing the instructions flow and identifying jump instructions, we only selected signals related to this flow: 32-bit instruction signal and 1-bit instruction valid signal.
2. **Probe Connection.** As the hardware Detection Module is synchronized with the processor, no special Probe Connection is needed, which means that the probed signals are directly connected to the hardware Detection Module's inputs.
3. **Detection Logic Design.** A ROP attack is characterized by a certain number of consecutive short instruction sequences (so-called gadgets) that always ends with an indirect jump instruction (`jalr` instruction). To detect this behavior, the number of instructions between two `jalr` is traced at runtime. When a `jalr` occurs, we compare the current instruction count to a threshold value called *GadgetSizeThreshold* to determine if it is a short gadget (*GadgetSizeThreshold* is empirically determined). We keep track of the successive short gadgets executed in a dedicated register *GadgetCounter* and consider an attack when a threshold value *AttackThreshold* is reached (this value is also empirically determined). To take into account ROP chains that sometimes include longer gadgets, we do not reset to 0 *GadgetCounter* when a long gadget is detected but decrement the counter by 2 instead.
4. **Benchmarking** The test campaign was carried out in the same way as for the previous use case: (1) reset all detection logic and start monitoring; (2) execute the program under test; (3) stop monitoring and collect information (typically the *GadgetCounter* value) stored in MMIO registers. The benchmarks used are also the same as in the previous use case.

The determination of a suitable *GadgetSizeThreshold* is highly dependent on existing ROP attacks on a given ISA. To the best of our knowledge, there has been no exhaustive study on the maximum size of the small gadgets used in ROP for RISC-V but some papers show some proof of concept. Studies on x86 ISA [38,39] show that practical ROP attacks generally require a gadget chain longer than 15 with at most 5 instructions for the small gadgets (equivalent to 7 instructions in RISC-V due to the additional 2 instructions for the management of return address register). For this reason, we studied several scenarios with *GadgetSizeThreshold* ranging from 8 to 16. We evaluated the maximum *GadgetCounter* values for all non-attack-related benchmarks. Results are provided in Table 4. From this table, the determination of a suited *AttackThreshold* is quite straightforward and can be set as the maximum value of the measured *GadgetCounter* of all benchmarks plus one.

Table 4. Maximum value of *GadgetCounter* observed with different values of *GadgetSizeThreshold* configured in detection logic, during the non-attack-related benchmark's execution. Sets of benchmarks are noted with a *.

Program	GadgetSizeThreshold								
	8	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
CoreMark *	4	7	10	11	11	11	11	13	17
CoreMark-PRO *	4	7	10	11	11	11	11	13	54
Embench wikisort	4	758	788	800	801	801	801	801	801
Embench others *	4	7	10	11	11	11	11	13	17
Dhrystone	4	7	10	11	11	11	11	11	17
STREAM	4	7	7	11	11	11	11	11	17

For instance, for ROP attacks crafted with quite short gadgets (e.g., *GadgetSizeThreshold* set to 8) we can observe that all benchmarks exhibit a maximum *GadgetCounter* value less than 5. In that case, a *AttackThreshold* value of 5 is suited to detect ROP attacks with at least 5 small gadgets of at most 8 instructions. If we increase the threshold of the number of instructions for small gadgets, we observe that the minimal number of small gadgets required to avoid false positives on the non-attack-related benchmarks (i.e., *AttackThreshold*) also increases. Consequently, we can conclude that the accuracy of the detection depends on the typical value of the maximum number of instructions in short gadgets. For instance, for a *GadgetSizeThreshold* of 14, the detection covers ROP attacks of at least 12 chained short gadgets. As stated before, we can expect practical ROP attacks on RISC-V to require about 15 small gadgets of at least 7 instructions, which is typically what our detection can cover with an even security margin.

Let us note that the wikisort benchmark in Embench suit produces many false positives. By inspecting the code, we found that it uses several small and noninline C functions such as `TestCompare`, which could explain the long "short gadget chain" found by our detection pattern. Finally, let us underline that PID information from kernel is especially important to filter out the system activities and prevent the ROP attacks from escaping our detection due to the context switch interruptions.

5.3.4. Area Footprint

The area overhead considering MATANA with our ROP detection pattern is 1.90% of Flip-Flops and 1.95% of Look-Up Tables utilization compared with the base system. This value is much smaller than in the previous use case, mainly because the Detection Module is synchronized with the target component.

6. Discussion

MATANA is a framework suited for designing and implementing specific runtime detection logics (both hardware and software) for different classes of attacks. Even if we only described experiments for two specific attacks in this paper, we are highly confident

that some suited detection logics can be designed and integrated in MATANA for other classes of attacks. For instance, if we consider microarchitectural attacks such as Flush + Reload and Spectre, we can observe that they both exhibit short sequences of repeated instruction patterns and that they use hardware components such as the cache and the branch predictor in a particular way. As a consequence, there is a high probability that they are distinguishable from legitimate programs with patterns based on the instructions executed and hardware events even at low frequency.

MATANA can of course be used along with other defense mechanisms to improve detection efficiency. Existing attack detection techniques, such as those cited in Section 2, can be a source of insight of probed signals and detection logics. Other classical detection or mitigation mechanisms such as Control Flow Integrity and shadow stack can be implemented in MATANA's Detection Module, by using an isolated hardware memory zone to store the control flow graph or shadow return address and performing a check when encountering jump instructions. Finally, let us note that in addition to attack detection, MATANA can be used to analyze the information leakage of hardware components or to design more secure hardware components and more security-oriented HPCs.

Several limitations of MATANA can be identified. First, during the design phase, a deep understanding of target components and attacks is necessary, with possibly a large number of experiments required to identify interesting probed signals and design the appropriate detection logic. This limitation is not really specific to MATANA but is rather due to the complexity of attacks and defense mechanisms at the microarchitecture level. Second, if new classes of attacks are discovered, and if they only partially impact the implemented probed signals, the detection accuracy may be reduced compared with a detection system built with a more specific set of probes. Indeed, as explained in Section 4.1, for final product integration, we propose using a static set of limited probes and a reconfigurable Detection Module for the best balance between performance and detection efficiency. Nevertheless, in that case, we are still confident that the ability to build fine-tuned detection logics makes it possible to build suited detection logics, even if not perfect. Third, since the detection algorithms are both implemented in the hardware Detection Module and in a software kernel module, the kernel module must be trusted. Several solutions may be proposed for that purpose according to the architecture of the SoC considered. If the SoC embeds a Trusted Execution Environment (TEE)—which is quite realistic for embedded critical systems—which are the most likely to adopt protection mechanisms such as those proposed in this paper, the privileged software may be implemented in the TEE. Otherwise, if the SoC embeds an advanced OS, such as a Linux OS used in our experiments, some kernel well-known hardening measures can be implemented. In case of a tiny OS, in which such hardening measures cannot be implemented, the detection logics must only be integrated in the hardware Detection Module, which only authorizes simple detection algorithms.

7. Conclusions

In this paper, we proposed MATANA, a framework that allows the design and integration of a flexible runtime attack detector for different classes of attacks that exhibit specific footprints on the microarchitecture. The detection is based on the fine-grained observation and analysis of different microarchitectural signals, obtained by inserting probes in the target component. The flexibility of the detection is provided by the Detection Module composed of reconfigurable hardware and software, which allow for updating of detection logics at any moment if necessary. Different data transmission techniques have been proposed to preserve high information throughput and timing information, even with the presence of frequency discrepancies between reconfigurable hardware and the target component. Our iterative methodology allows the design of different detection logics according to experiment results in real environments.

An open-source system prototype has been implemented on FPGA, including a RISC-V processor Rocket and running Linux 4.15. Two use cases are presented, in which we describe how we were able to build suited detection logics for CSCAs and ROP attacks.

These two use cases demonstrate that MATANA can be used for attacks of a different nature, and that the built detector exhibits satisfying false positive and false negative detection rates with quite simple heuristics, even with frequency differences.

Our study shows that hardware security does not have to be limited to static solutions that cannot be modified after manufacturing or to solutions that only include a few programmable configuration bits. The security of systems over their lifetime can be enhanced by reconfigurable hardware equipped with attack detection logics based on analysis of microarchitectural signals. We believe this is particularly valuable today, when hardware lifetime is relatively long compared with the evolution speed of software attacks, and the fact that software attacks are able to target hardware vulnerabilities. Hardware component or system designers should be aware of this possibility and consider including this adaptable security into their design. Researchers can use MATANA's tools and methodology to design and evaluate new attack detection algorithms on hardware and software, understanding microarchitectural behavior, without being limited by information provided by existing hardware.

As for future work, we first plan to enrich our detection campaigns by submitting other use cases of attacks to MATANA—such as Spectre/Meltdown for example. We also think that it would be interesting to make MATANA more automatic and easy to use by the research community. For example, automatically selecting signals based on the hardware's source code, or generating patterns that analyze the attack's characteristic or the correlation between signals and secret based on some formal description of the attack, could provide useful and relevant enhancements. We also plan to investigate more complex detection logics to estimate their cost in term of area footprint and execution time. Indeed, even if the experiments we carried out so far tend to show that quite simple detection logics are sufficient to exhibit satisfying detection rates, we think it interesting to investigate the implementation of complex detection algorithms (such as Machine Learning algorithms) in order to evaluate to what extent it is possible to implement such algorithms in MATANA, both in hardware and software, to optimize the detection efficiency as well as the area footprint on the reconfigurable fabric.

Author Contributions: Conceptualization, Y.M., V.M. and V.N.; methodology, Y.M. and V.M.; software, Y.M.; validation, Y.M. and V.M.; formal analysis, Y.M. and V.M.; investigation, Y.M.; data curation, Y.M.; writing—original draft preparation, Y.M., V.M. and V.N.; writing—review and editing, Y.M., V.M. and V.N.; visualization, Y.M. and V.M.; supervision, V.M. and V.N. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: Thanks to Paul Florence for his support in writing the Linux kernel driver and the userland program. Thanks to Lucas Georget for his support in writing the ROP attacks generator.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CSCA	Cache Side-Channel Attack
CTC	Covert Timing Channel
DIFT	Dynamic Information Flow Tracking
FPGA	Field Programmable Gate Array
FPU	Floating-Point Unit
HPC	Hardware Performance Counter
ISA	Instruction Set Architecture

LRU	Least Recently Used
MMIO	Memory-Mapped I/O
OS	Operating System
PID	Process Identifier
ROP	Return-Oriented Programming
SoC	System-on-Chip
TEE	Trusted Execution Environment

References

1. Yarom, Y.; Falkner, K. FLUSH + RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, 20–22 August 2014; USENIX Association: Berkeley, CA, USA, 2014; pp. 719–732.
2. Osvik, D.A.; Shamir, A.; Tromer, E. Cache Attacks and Countermeasures: The Case of AES. In Proceedings of the Cryptographers' Track at the RSA Conference, San Jose, CA, USA, 13–17 February 2006; Springer: Berlin/Heidelberg, Germany, 2006; pp. 1–20.
3. Kocher, P.; Horn, J.; Fogh, A.; Genkin, D.; Gruss, D.; Haas, W.; Hamburg, M.; Lipp, M.; Mangard, S.; Prescher, T.; et al. Spectre Attacks: Exploiting Speculative Execution. In Proceedings of the 2019 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 20–22 May 2019; pp. 1–19.
4. Lipp, M.; Schwarz, M.; Gruss, D.; Prescher, T.; Haas, W.; Fogh, A.; Horn, J.; Mangard, S.; Kocher, P.; Genkin, D.; et al. Meltdown: Reading Kernel Memory from User Space. In Proceedings of the 27th USENIX Security Symposium, Baltimore, MD, USA, 14–16 August 2018; USENIX Association: Berkeley, CA, USA, 2018; pp. 973–990.
5. Ge, Q.; Yarom, Y.; Cock, D.; Heiser, G. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *J. Cryptogr. Eng.* **2018**, *8*, 1–27. [CrossRef]
6. Lyu, Y.; Mishra, P. A Survey of Side-Channel Attacks on Caches and Countermeasures. *J. Hardw. Syst. Secur.* **2018**, *2*, 33–50. [CrossRef]
7. Akram, A.; Mushtaq, M.; Bhatti, M.K.; Lapotre, V.; Gogniat, G. Meet the Sherlock Holmes' of Side Channel Leakage: A Survey of Cache SCA Detection Techniques. *IEEE Access* **2020**, *8*, 70836–70860. [CrossRef]
8. RISC-V International. RISC-V. 2021. Available online: https://riscv.org/ (accessed on 10 January 2022).
9. Asanović, K.; Avižienis, R.; Bachrach, J.; Beamer, S.; Biancolin, D.; Celio, C.; Cook, H.; Dabbelt, P.; Hauser, J.; Izraelevitz, A.; et al. *The Rocket Chip Generator*; Technical Report UCB/EECS-2016-17; EECS Department, University of California: Berkeley, CA, USA, 2016.
10. Li, C.; Gaudiot, J. Online Detection of Spectre Attacks Using Microarchitectural Traces from Performance Counters. In Proceedings of the 30th International Symposium on Computer Architecture and High Performance Computing, Lyon, France, 24–27 September 2018; pp. 25–28.
11. Demme, J.; Maycock, M.; Schmitz, J.; Tang, A.; Waksman, A.; Sethumadhavan, S.; Stolfo, S.J. On the Feasibility of Online Malware Detection with Performance Counters. In Proceedings of the 40th Annual International Symposium on Computer Architecture, Tel-Aviv, Israel, 23–27 June 2013; pp. 559–570.
12. Payer, M. HexPADS: A Platform to Detect "Stealth" Attacks. In Proceedings of the 8th International Symposium on Engineering Secure Software and Systems, London, UK, 6–8 April 2016; Springer: Berlin/Heidelberg, Germany, 2016; pp. 138–154.
13. Kulah, Y.; Dincer, B.; Yilmaz, C.; Savas, E. SpyDetector: An Approach for Detecting Side-Channel Attacks at Runtime. *Int. J. Inf. Secur.* **2019**, *18*, 393–422. [CrossRef]
14. Chiappetta, M.; Savas, E.; Yilmaz, C. Real Time Detection of Cache-Based Side-Channel Attacks Using Hardware Performance Counters. *Appl. Soft Comput.* **2016**, *49*, 1162–1174. [CrossRef]
15. Chen, J.; Venkataramani, G. CC-Hunter: Uncovering Covert Timing Channels on Shared Processor Hardware. In Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, UK, 13–17 December 2014; pp. 216–228.
16. Li, W.; Li, M.; Ma, Y.; Yang, Q. PMU-extended Hardware ROP Attack Detection. In Proceedings of the 12th IEEE International Conference on Anti-Counterfeiting, Security, and Identification, Xiamen, China, 9–11 November 2018; pp. 183–187.
17. ARM. ARM CoreSight Architecture Specification v3.0. 2017. Available online: https://developer.arm.com/documentation/ihi0029/e (accessed on 10 January 2022).
18. ARM. CoreSight Program Flow Trace Architecture Specification PFTv1.0 and PFTv1.1. 2011. Available online: https://developer.arm.com/documentation/ihi0035/b/ (accessed on 10 January 2022).
19. Wahab, M.A.; Cotret, P.; Allah, M.N.; Hiet, G.; Biswas, A.K.; Lapotre, V.; Gogniat, G. A Small and Adaptive Coprocessor for Information Flow Tracking in ARM SoCs. In Proceedings of the 2018 International Conference on ReConFigurable Computing and FPGAs, Cancun, Mexico, 3–5 December 2018; pp. 1–8.
20. Lee, Y.; Heo, I.; Hwang, D.; Kim, K.; Paek, Y. Towards a Practical Solution to Detect Code Reuse Attacks on ARM Mobile Devices. In Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy, Portland, OR, USA, 13 June 2015; pp. 3:1–3:8.
21. Dalton, M.; Kannan, H.; Kozyrakis, C. Raksha: A Flexible Information Flow Architecture for Software Security. In Proceedings of the 34th International Symposium on Computer Architecture, San Diego, CA, USA, 9–13 June 2007; pp. 482–493.

22. Deng, D.Y.; Lo, D.; Malysa, G.; Schneider, S.; Suh, G.E. Flexible and Efficient Instruction-Grained Run-Time Monitoring Using On-Chip Reconfigurable Fabric. In Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, 4–8 December 2010; pp. 137–148.
23. Deng, D.Y.; Suh, G.E. High-Performance Parallel Accelerator for Flexible and Efficient Run-Time Monitoring. In Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks, Boston, MA, USA, 25–28 June 2012; pp. 1–12.
24. Chen, S.; Falsafi, B.; Gibbons, P.B.; Kozuch, M.; Mowry, T.C.; Teodorescu, R.; Ailamaki, A.; Fix, L.; Ganger, G.R.; Lin, B.; et al. Log-Based Architectures for General-Purpose Monitoring of Deployed Code. In Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability, San Jose, CA, USA, 21 October 2006; pp. 63–65.
25. Delshadtehrani, L.; Canakci, S.; Zhou, B.; Eldridge, S.; Joshi, A.; Egele, M. PHMon: A Programmable Hardware Monitor and Its Security Use Cases. In Proceedings of the 29th USENIX Security Symposium, Baltimore, MD, USA, 12–14 August 2020; USENIX Association: Berkeley, CA, USA, 2020; pp. 807–824.
26. Mao, Y.; Migliore, V.; Nicomette, V. REHAD: Using Low-Frequency Reconfigurable Hardware for Cache Side-Channel Attacks Detection. In Proceedings of the IEEE European Symposium on Security and Privacy Workshops, Genoa, Italy, 7–11 September 2020; pp. 704–709.
27. Xilinx. UG029—ChipScope Pro Software and Cores. 2012. Available online: https://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/chipscope_pro_sw_cores_ug029.pdf (accessed on 10 January 2022).
28. Intel. Intel Quartus Prime Pro Edition User Guide: Debug Tools. 2020. Available online: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-qpp-debug.pdf (accessed on 10 January 2022).
29. Reinbrecht, C.; Hamdioui, S.; Taouil, M.; Niazmand, B.; Ghasempouri, T.; Raik, J.; Sepúlveda, J. LiD-CAT: A Lightweight Detector for Cache ATtacks. In Proceedings of the IEEE European Test Symposium, Tallinn, Estonia, 25–29 May 2020; pp. 1–6.
30. Amid, A.; Biancolin, D.; Gonzalez, A.; Grubb, D.; Karandikar, S.; Liew, H.; Magyar, A.; Mao, H.; Ou, A.J.; Pemberton, N.; et al. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* **2020**, *40*, 10–21. [CrossRef]
31. Bachrach, J.; Vo, H.; Richards, B.C.; Lee, Y.; Waterman, A.; Avizienis, R.; Wawrzynek, J.; Asanovic, K. Chisel: Constructing Hardware in a Scala Embedded Language. In Proceedings of the 49th Annual Design Automation Conference 2012, San Francisco, CA, USA, 3–7 June 2012; pp. 1216–1225.
32. SiFive. SiFive Freedom Unleashed SDK. 2021. Available online: https://github.com/sifive/freedom-u-sdk/tree/v1_0 (accessed on 10 January 2022).
33. Liu, F.; Yarom, Y.; Ge, Q.; Heiser, G.; Lee, R.B. Last-Level Cache Side-Channel Attacks are Practical. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, Washington, DC, USA, 17–21 May 2015; pp. 605–622.
34. Gruss, D.; Lipp, M.; Schwarz, M.; Genkin, D.; Juffinger, J.; O'Connell, S.; Schoechl, W.; Yarom, Y. Another Flip in the Wall of Rowhammer Defenses. In Proceedings of the 2018 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 20–24 May 2018; pp. 245–261.
35. Yarom, Y. Mastik: A Micro-Architectural Side-Channel Toolkit. 2016. Available online: https://cs.adelaide.edu.au/~yval/Mastik/ (accessed on 10 January 2022).
36. Shacham, H. The Geometry of Innocent Flesh on the Bone: Return-Into-Libc without Function Calls (on the x86). In Proceedings of the 2007 ACM Conference on Computer and Communications Security, Alexandria, VA, USA, 31 October–2 November 2007; pp. 552–561.
37. Jaloyan, G.; Markantonakis, K.; Akram, R.N.; Robin, D.; Mayes, K.; Naccache, D. Return-Oriented Programming on RISC-V. In Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, 5–9 October 2020; pp. 471–480.
38. Cheng, Y.; Zhou, Z.; Yu, M.; Ding, X.; Deng, R.H. ROPecker: A Generic and Practical Approach For Defending Against ROP Attacks. In Proceedings of the 21st Annual Network and Distributed System Security Symposium, San Diego, CA, USA, 18–21 February 2014; The Internet Society: Reston, VA, USA, 2014.
39. Chen, P.; Xiao, H.; Shen, X.; Yin, X.; Mao, B.; Xie, L. DROP: Detecting Return-Oriented Programming Malicious Code. In *International Conference on Information Systems Security*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2009; Volume 5905, pp. 163–177.