HarTBleed: Using Hardware Trojans for Data Leakage Exploits

Asmit De, Mohammad Nasim Imtiaz Khan, Karthikeyan Nagarajan, and Swaroop Ghosh

***Abstract*— Data and information leakage is an important security concern in current systems. Several data leakage prevention (DLP) techniques have been proposed in the literature to prevent external as well as internal data leakage. Most of these solutions try to trace data flow and perform privilege checks to ensure the security of the data at the software and system level. Architecture level leakage vulnerabilities such as Spectre and Meltdown can be mitigated by performance-expensive software patches or by modifying the architecture itself. However, these solutions assume that the underlying hardware platform is secure and free from tampering. In this article, we present HarTBleed, a class of system attacks involving hardware compromised with a Trojan embedded in the CPU. We show that attacks crafted specifically to make use of the Trojan can be used to obtain sensitive information from the address space of a process. We propose the use of a capacitor-based Trojan trigger that exploits the virtual addressing of L1 cache to activate a Trojan payload that resets a target translation lookaside buffer (TLB) entry to maliciously map to sensitive data in memory. Extensive circuit simulation indicates that the proposed Trojan trigger is not activated during test or normal operation even under a wide range of process/temperature conditions. Therefore, it remains undetected. A successful HarTBleed-based exploit is demonstrated using an attack code by modeling the Trojan effects in the GEM5 simulator.**

***Index Terms*— Data leakage, hardware Trojan, translation lookaside buffer (TLB), trigger.**

I. Introduction

TODAY'S computing systems have become increasingly complex with load-balanced and multiuser-shared infrastructure. Large cloud storage and computing services, such as Amazon Web Services and Microsoft Azure, allow a single computing resource to be shared across multiple users or services. This is made possible using isolation mechanisms such as containers and virtual machines. Such systems need to ensure that data contained inside a virtual machine or a container cannot be accessed from outside. Furthermore, the operating system kernel or the hypervisors in these systems need to ensure their integrity so that the internal resource allocation and memory layout information are not visible to the user-level processes.

Data leakage prevention (DLP) techniques usually offer protection of sensitive data by content monitoring and detection, tracking anomalies in data behavior [1], or apply cryptographic techniques such as encryption to secure the data [2]–[4]. However, these techniques only work at the application level. Data-flow analysis and dynamic taint tracking [5] are employed at the system level to ensure data protection of running processes. User-level processes are protected from code injection and code reuse by techniques such as control flow integrity (CFI) [6], data execution prevention (DEP) [7], and address space layout randomization (ASLR) [8]. Even with such protections in place, a bug was recently found in the OpenSSL cryptographic library that leaked data from memory using a buffer overread vulnerability, allowing attackers to eavesdrop on SSL/TLS secured communication channels [9]. User processes are separated from the OS kernel by employing protection rings in the CPU, which ensures that user space code cannot access data in kernel space. However, recent vulnerabilities have been found in the systems architecture, namely, Spectre [10] and Meltdown [11], which can take advantage of the speculative execution of out-of-order processors to gain unauthorized access to data from the CPU cache, and even leak kernel data. Existing OS level patches to handle Meltdown incur 0%–30% performance overhead whereas cleaning branch predictors and branch target buffers, Lfence, etc. [12] for Spectre are expected to impose more serious performance overheads. Architecture design changes are also planned to prevent such vulnerabilities.

The underlying assumption of the above threats and their mitigation techniques is that the hardware itself is free from malicious tampering. However, the recently surfaced news such as tampering of server motherboards by Chinese manufacturers that affected top U.S. companies such as Amazon and Apple [13] emphasize that this assumption might not be true due to the involvement of untrusted third parties in the semiconductor manufacturing supply chain. Another popular incident is the hardware fabricated with hidden-backdoor to disable radars in Syria [14] to facilitate an attack.

In light of the above threat, the U.S. Defense Advanced Research Projects Agency (DARPA) identified the trusted and untrusted steps of a supply chain [21] and initiated a Trust-in-IC (Integrated Circuit) program to develop tools and techniques to ensure that ICs are authentic and free of Trojans postmanufacturing. The Australian Department of Defense also raised awareness of the threat and proposed broad classes of Hardware Trojans and countermeasures [22]. Research has

Manuscript received September 16, 2019; revised November 22, 2019; accepted December 12, 2019. Date of publication January 14, 2020; date of current version March 20, 2020. This work was supported in part by SRC under Grant 2727.001; in part by NSF under Grant CNS1722557, Grant CCF-1718474, Grant DGE-1723687, and Grant DGE-1821766; and in part by the DARPA Young Faculty Award under Grant D15AP00089. *(Corresponding author: Asmit De.)*

The authors are with the School of Electrical Engineering and Computer Science, The Pennsylvania State University, University Park, PA 16802 USA (e-mail: asmit@psu.edu; muk392@psu.edu; kxn287@psu.edu; szg212@psu.edu).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2019.2961358

also been conducted to develop efficient DLP frameworks for the semiconductor industry [23]. In spite of the above efforts, the industry heavily relies on untrusted third parties for intellectual property (IP) and manufacturing to preserve the cost benefits.

A. Hardware Trojan

Hardware Trojan [24] is a malicious modification in a circuit that is introduced during the design and/or manufacturing process to force a chip to perform undesirable operations. Ideally, these modifications made to an IC should be detected during pre-Silicon verification and post-Silicon testing. In order to evade the structural and functional testing, an adversary designs the Trojan to activate only under certain rare conditions and to remain undetected during the test phase. Hardware Trojan has two main components, namely: 1) Trojan trigger and 2) Payload [25]. Trojan Trigger is designed in such a way that it gets activated under a certain unique condition that might not be possible to generate in the test/validation phase, and thereby, remains undetected. Once triggered, the Trojan can cause denial of service (DoS), fault injection, or information leakage [25], [26].

Although hardware Trojan is a well-explored topic in the VLSI design communities, most of the attack vectors are confined to low-level circuits and implications such as DoS and leakage of cryptographic keys. The potential of hardware Trojans in compromising system assets and launch system security attacks has received much less attention.

B. Proposed Attack Model

In this work, we assume an untrusted manufacturing house located outside the U.S. that can alter the chip GDS-II file to introduce the malicious Trojan trigger and payload. This assumption is widely accepted in the hardware security community because of large filler areas present in the chip and the adversary's access to the raw design. We propose the use of a capacitor-based hardware Trojan trigger [26] and novel payload circuits and perform detailed analysis to guarantee that the Trojan is: 1) triggered even under worst case process and temperature conditions with correct inputs and 2) able to bypass conventional postmanufacturing test. The Trojan is activated if a particular preselected address of L1 Cache is accessed for $\sim$1800 times. Note that the proposed Trojan trigger directly taps the wordline of the preselected address to leverage the existing decoder design framework and hence, does not incur any overhead for address decoding. Once activated, the trigger will reset the preselected translation lookaside buffer (TLB) entry to preselected bit pattern. A TLB entry is composed of a tag [virtual page number (VPN)] and mapping information [physical frame number (PFN) and other metadata bits]. The tag is implemented using a content addressable memory (CAM) for high-speed lookup operation, while the mapping information is implemented on a 6T SRAM array [27]. The proposed Trojan payload resets the SRAM portion of the TLB.

In order to simulate the effect of the Trojan, we design an attack with the hardware Trojan trigger embedded in the CPU cache and the TLB as a victim, as shown in Fig. 1. The goal of the Trojan payload is to manipulate the TLB mapping information to access data from a process's address space. The Trojan is triggered when a specific address of L1 is accessed multiple times. This changes the address mapping in a target TLB line to a specific physical page with sensitive information.

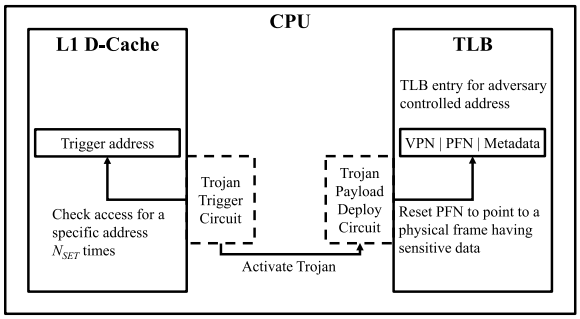

Fig. 1. Overview of HarTBleed Trojan.

In addition to our Trojan trigger proposed in [26], following contributions are made in this article. We

1) propose the SRAM Trojan payload to reset the TLB entry to a known value;
2) perform a detailed analysis of the Trojan trigger under process variations and temperature fluctuations;
3) design a system exploit using the Trojan;
4) demonstrate the HarTBleed exploit in action using GEM5 simulations;
5) explore possible attack surfaces using HarTBleed methodology; and
6) present countermeasures to detect the Trojan trigger and payload.

This article is organized as follows. Section II reviews the existing literature on hardware Trojans; Section III describes the proposed Trojan trigger and payload; Section IV describes the system architecture including the threat model under consideration; Section V presents the HarTBleed methodology; Section VI presents a discussion on the practicality, assumptions, and limitations of HarTBleed; and finally, Section VII draws the conclusion.

II. Background on Hardware Trojan

In this section, we present an overview of hardware Trojans explored in the existing literature.

A. Trojan Trigger and Payload

Many prior works have investigated possible hardware Trojans. A qualitative comparison of state-of-the-art Trojan designs is presented in Table I. In [15], a "content & timing" based Trojan trigger is designed and implemented in a Basys FPGA Board which gets activated only when the correct input pattern is entered at the correct time. The website demonstrates that the Trojan can evade the test phase even if a correct trigger pattern is provided since the timing constraints are not met. In [16], Trojans are proposed by leveraging the availability of multiple transistor threshold voltages in advanced technology nodes. The authors show that threshold voltage modification techniques (e.g., ion implantation) can be leveraged to introduce stuck-at-faults to a D flip-flop. They also show that

TABLE I
QUALITATIVE COMPARISON OF TROJAN DESIGNS

	[15]	[16]	[17]	[18]	[19]	[20]	HarTBleed
Trojan type	Content & timing	Multi-VT	MAPLE	Embedded memory	NVM	Analog capacitor	Capacitor
Trigger	Data pattern, thermal state,	VT manipulation, thermal state	Supply noise fluctuation	Data pattern	Memory access, delay and voltage	Divide-by-zero	L1 Cache access, data pattern
Payload	AES key leak	Fault injection, data latching	Fault injection	Fault injection	Fault injection	Privilege escalation	Data leakage with TLB faults
Attack scope	Crypto engines	Flip-Flops	Crypto engines	SRAM	Memory	Privilege register	TLB

manipulation can be done in a way that the Trojan activates only at a specific temperature and thereby evades testing at low temperature and at burn-in postfabrication. In [17], two approaches are proposed to change the voltage transfer characteristics of a target gate which activates as a Trojan when the supply voltage reduces. The proposed Trojan can extract secret keys by injecting transient faults into a lightweight cryptographic block. Hoque *et al.* [18] have proposed a Trojan for an embedded SRAM which evades industry-standard post-manufacturing memory tests (for example, March test). The Trojan gets activated by writing a specific pattern to one/few cells (that work as a trigger) that feed into the input of the Trojan transistors (payload). The Trojan payload transistors short the data node of a victim SRAM cell to ground (data corruption). Emerging NVM-based Trojans have also been proposed [19], where the Trojan is triggered by sensing delays and voltages for repeated data access to a particular address in a RRAM memory.

In [20], an analog Trojan trigger, A2, is presented which is controllable, stealthy, and small. This proposes a capacitor-based trigger that aims to flip specific bits of control logic after a number of instructions are issued resulting in escalation of the adversary's privilege. In [26], a capacitor-based Trojan trigger circuit is presented that is activated by writing a specific data pattern to a specific address for a number of times (∼260, denoted by N_{SET} in this article). Note that such triggers remain undetected during postmanufacturing test since an address is not hammered for many number of times to maintain quick test time. Furthermore, the Trojans incur very small area/power overhead which cannot be detected by optical inspection or side channel analysis during the test. Optical inspection also requires invasive reverse engineering which may not provide a good quality image for detection. Advanced techniques such as transmission electron microscopy (TEM) are also very expensive and may increase the test cost. Furthermore, to obfuscate the Trojan hardware, the adversary can also replicate the design in all the memory subarrays to make them identical.

B. Trojan Detection Techniques

Trojan detection techniques have been proposed using sophisticated failure analysis like light-induced voltage alternation (LIVA), charge-induced voltage alternation (CIVA), and other imaging techniques. However, these methods require significant time/effort (requires chip delayering) and are not highly effective for nanometer technologies [28]. Two other techniques are proposed in [8] namely, automatic test pattern generation (ATPG) and side channel analysis (SCA). ATPG does not work for logic Trojan where the malicious inserted logic is unknown [28]. Therefore, it cannot detect the proposed trigger. It might be possible to trigger the proposed Trojan by writing each address with all possible combinations for many number of times (1837 for this article). However, this increases the test time and time to market the chip significantly. Typically, each chip is tested for 2–3 s [29], which is not enough to catch such Trojans. Furthermore, the Trojans become easier to deploy effectively if the designer itself is the adversary. SCA techniques, such as power profiling and matching with a golden chip [30], [31], are also ineffective against the proposed memory Trojan since the trigger only consumes dynamic power when it is activated/deactivated. Furthermore, they require an existing golden chip which may not be possible.

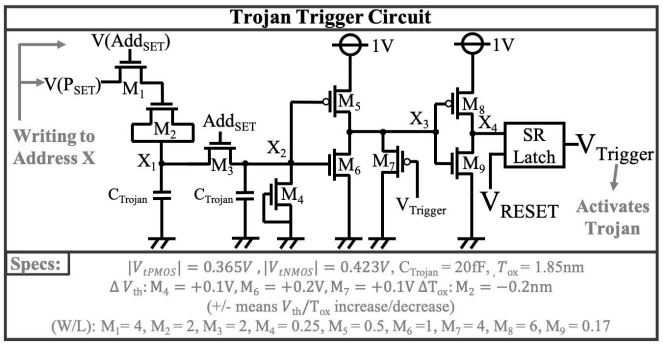

Fig. 2. Trojan trigger circuit (specifications provided at the bottom). $V(P_{\mathrm{SET}})$ and $V(\mathrm{Add}_{\mathrm{SET}})$ are two inputs.

III. HARDWARE TROJAN DESIGN FOR HARTBLEED

In this section, we present the Trojan trigger circuit and its process and temperature variation analysis.

A. Trojan Trigger

1) Design: The trigger circuit (Fig. 2) is designed to be activated if a particular memory address (chosen during design phase, let us call it ($\mathrm{Add}_{\mathrm{SET}}$) is accessed for at least N_{SET} times. The trigger has two inputs, namely $V(\mathrm{Add}_{\mathrm{SET}})$ and $V(P_{\mathrm{SET}})$. $V(\mathrm{Add}_{\mathrm{SET}})$ (= 1 V in this article) is the wordline

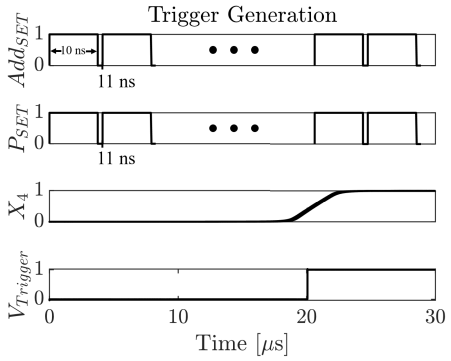

Fig. 3. Trojan trigger waveform.

enable signal of Add_{SET} and $V(P_{\text{SET}})$ is a constant voltage source of 1 V. For a more complex Trojan with a superior stealthiness, $V(P_{\text{SET}})$ can be programmable (discussed in Section VI-A).

Whenever Add_{SET} is accessed, $V(\text{Add}_{\text{SET}})$ is asserted and MOSFETs M_1 and M_3 are activated. M_2 has a thinner gate oxide compared to other MOSFETs and its source and drain are shorted. Therefore, M_2 works as a capacitor and charges C_{Trojan} from the P_{SET} source through Fowler Nordheim (FN) tunneling [32] if $V(\text{Add}_{\text{SET}})$ is asserted. M_4 is an OFF transistor which offsets gate leakage of M_5 and prevents unwanted charging-up of node X_2. M_7 keeps node X_3 as low as possible until node X_2 charges up sufficiently. The node X_4, that is charged up during the hammering process, is used as the SET input of a SR latch. The output of the SR latch (V_{Trigger}) transitions from $0 \rightarrow 1$ when X_4 charges up to 0.5 V. The signal V_{Trigger} is then used to activate the Trojan. Fig. 3 shows the Trojan trigger waveform.

The charge at node X_2 leaks away (due to capacitance leakage of C_{Trojan}) once the hammering is discontinued. However, V_{Trigger} will still be asserted due to the SR latch. In order to deactivate the Trojan, V_{RESET} needs to be asserted. V_{RESET} can be generated by accessing a different address (let us say $\text{Add}_{\text{RESET}}$) for at least N_{RESET} times and using a circuit similar to the trigger one. Note that a smaller C_{Trojan} ($\sim$1fF) can be used in the RESET circuit to minimize the area overhead which leads to $N_{\text{RESET}} = 92$. However, the AND'ed output of $V(\text{Add}_{\text{RESET}})$ and $V(P_{\text{RESET}})$ can also serve as V_{RESET} which further reduces the area overhead.

2) Simulation Results: Node X_2 charges up to 125 mV (steady state) from all the leakage considering $V(P_{\text{SET}}) = 1$ V (worst case charging due to leakage). This value is not enough to trigger the circuit. For the rest of the simulation, we have considered that $V(\text{Add}_{\text{SET}})$ is a pulse source with ON/OFF time of 10 ns/1 ns. We consider the circuit to be triggered when V_{Trigger} reaches up to 0.5 V, where initial $C_{\text{Trojan}} = 20$ fF.

Fig. 4(a) shows the design space exploration of trigger circuit considering two variables, the (W/L) ratios of MOSFETs M_1 and M_2. For a lower (W/L) ratio for both the MOSFETs, N_{SET} increases. We have chosen (W/L) of M_1 and M_2 as 4 and 2, respectively, for a sufficiently higher N_{SET}.

Next, we considered that the adversary accesses the preselected address for $T_{\text{on}} = 10$ ns and then stays idle for

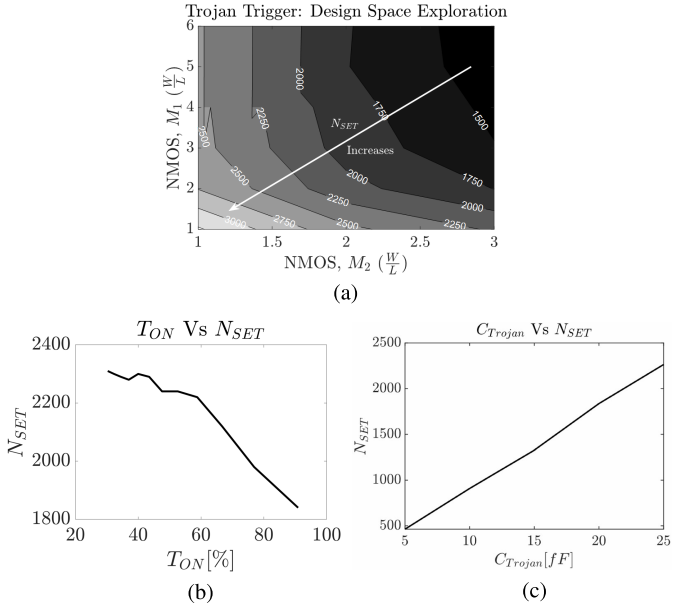

Fig. 4. (a) Design space exploration of Trojan trigger (Fig. 2) with respect to (W/L) ratio of MOSFET M_1 and M_2, (b) required number of access (N_{SET}) increases if the adversary cannot access continuously, and (c) a scaled down C_{Trojan} leads to less number of N_{SET} to activate the trigger.

$T_{\text{off}} = 1/3/.../21/23$ ns and repeats this cycle. We found that the C_{Trojan} does not leak in the OFF cycle significantly and the circuit can still be triggered but with a higher N_{SET} [Fig. 4(b)]. We observe that the circuit will trigger even with a low T_{ON} of 30%. This means that it becomes even harder to prevent Trojan activation using system level techniques such as limiting repeated access to one particular address.

Note that the attack gets auto reset without the SR latch since the node X_2 discharges (due to charge leakage of C_{Trojan}) and eventually node X_4 goes down once the adversary stops the hammering after the trigger activates. Results indicate that the attack (charge at node X_4) lasts for 163.73 μs if Add_{SET} is accessed for 18 μs. However, by adding the SR latch, the attack can last indefinitely until Add_{SET} access is discontinued and V_{RESET} is asserted.

A small C_{Trojan} will require a low N_{SET} to get the trigger activated. For example, $N_{\text{SET}} = 464$ for $C_{\text{Trojan}} = 5$fF [Fig. 4(c)]. This is still significantly high enough to evade the test phase. The value of C_{Trojan} is chosen as 20 fF since it offers a high N_{SET} ($= 1837$) under nominal conditions and successfully triggers under all process corners and temperatures (further details in Section III-B), and minimum N_{SET} in the worst case ($= 68$) is still high to evade testing phase.

B. Impact of Process and Temperature Variations

Process variations can lead to worst case scenarios where the required N_{SET} might be too low and lead to the circuit inadvertently triggering during the test phase. We have performed a 500-point Monte Carlo analysis at temperatures -10 °C, 25 °C, and 90 °C with the process variation modeled as 3 σ of 30 mV of each transistor's initial V_{th} for nominal design ($N_P N_N$). The lowest N_{SET} ($= 1318$) achieved at 90 °C is still

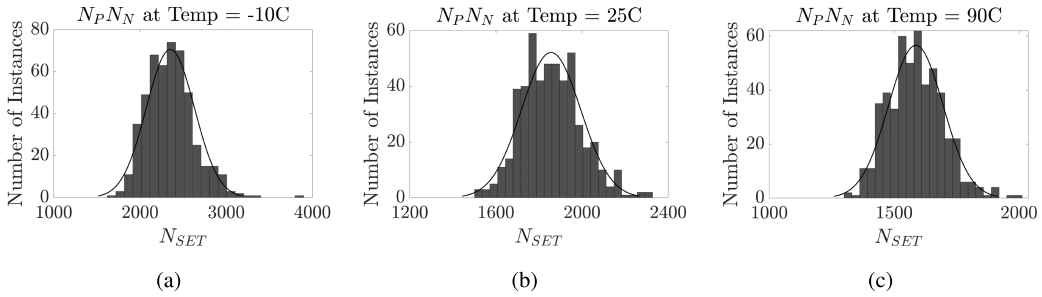

Fig. 5. Process variation analysis of the trigger: distribution of N_{SET} at (a) -10 °C, (b) 25 °C, and (c) 90 °C.

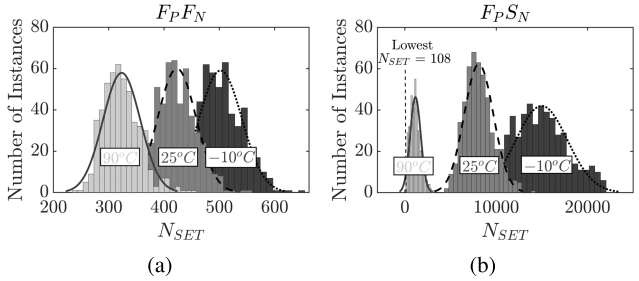

Fig. 6. Process variation analysis of trigger at corners. (a) F_PF_N. (b) F_PS_N.

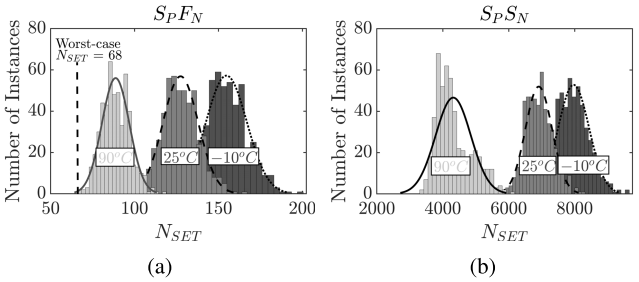

Fig. 7. Process variation analysis of trigger at corners. (a) S_PF_N. (b) S_PS_N.

high enough to evade the test phase. The distribution of N_{SET} for each temperature is shown in Fig. 5.

We have also performed process variation analysis at each process corner that includes all combinations of fast (F) and slow (S) p- and n-type transistors. The threshold voltage V_{th} of the F-transistors is reduced by 0.2 V and that of S-transistors is increased by 0.2 V. Figs. 6 and 7 show the distribution of N_{SET} at all process corners under all temperatures. We note that even in the worst case-corner of S_PF_N at 90 °C, the minimum recorded $N_{\text{SET}} = 68$ which is high enough to evade the test. We also observe the highest $N_{\text{SET}} = 22028$ at the F_PS_N corner at -10 °C. It is possible that a memory controller might be employed to prevent continuous accesses to Add_{SET}. However, we have shown that the trigger can operate even with breaks in accessing the target memory address [Fig. 4(b)].

C. Area and Power Analysis

Table II summarizes the key performance metrics of the Trojan trigger. The absolute area and static power of the proposed trigger are 42.9 μm^2 and 0.589 μW, respectively, which are 5.34×10^{-5}% and 6.24×10^{-5}% of a typical memory chip area and static power [33], respectively.

TABLE II
FEATURES OF THE TROJAN TRIGGER

Parameter	Trojan Trigger
Dynamic Power (μW)	3.781
Static Power (μW)	0.589
Energy/hammer (nJ)	2.059
Area (μm^2)	42.97
Target N_{SET}	1837
Worst-Case N_{SET}	1502

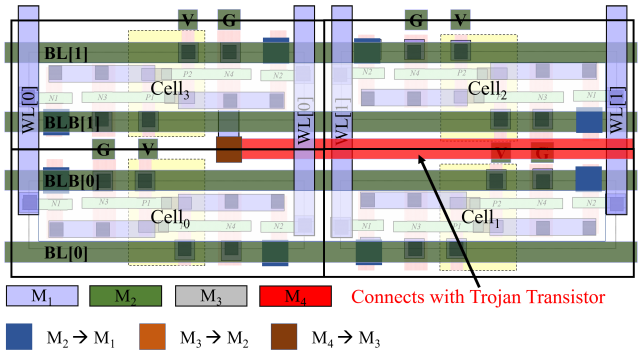

Fig. 8. Layout showing 4 SRAM bitells and metal tracks allocated for bitlines (horizontal) and wordlines (vertical). One M_4 track can be stolen to connect the SRAM data node to Trojan payload transistor (located in the column area).

Therefore, the overhead due to the Trojan trigger is negligible to be detected via optical inspection or side channel analysis.

D. Resetting Single and Multiple TLB Entries

The Trojan payload resets TLB entries after the trigger is activated. We have chosen the TLB as the victim as it provides a greater attack surface to leak data from multiple sources such as kernel, process space, and shared memory (further explained in Section VI-B). Furthermore, the payload circuit is placed in the SRAM portion of the TLB itself, as it is easy to conceal the transistors in the SRAM layout and peripheral circuits (as shown in Figs. 8 and 9).

A single TLB entry can be reset to preselected data pattern (i.e., "0" or "1") as shown in Fig. 9(a) and (b). If the adversary needs to reset multiple TLB entries (say, 10), he needs to select 10 L1 cache addresses. Let us call them AddX_{SET}, where $X = 1, 2, \ldots, 10$. For each of the target TLB entries the adversary designs one AND gate with V_{Trigger} and $V(\text{AddX}_{\text{SET}})$ as inputs [Fig. 10(a)]. When the adversary accesses address AddX_{SET}

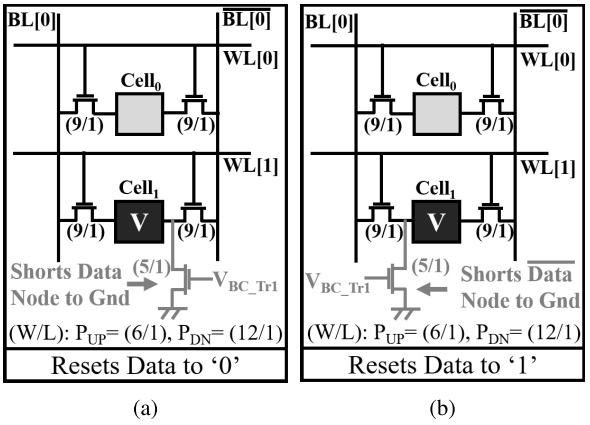

Fig. 9. TLB entry reset to (a) "0" and (b) "1." The sizes of SRAM pull-up, pull-down and access transistors are shown. The back-to-back inverters in SRAM are omitted for clarity.

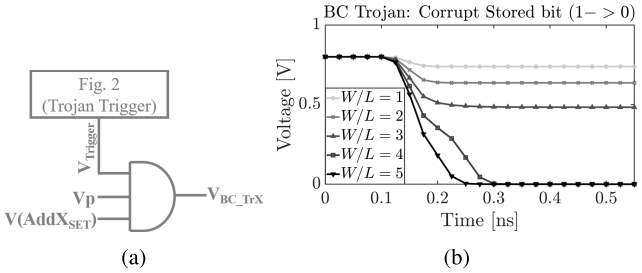

Fig. 10. (a) Logic circuit to generate V_{BC_TrX}. (b) Data node voltage discharges as the (W/L) of Trojan transistor increases. It discharges to 0V for a minimum (W/L) of 4. This means that the stored data flipped, i.e., $1 \rightarrow 0$ fault occurred.

after the Trojan is activated, $V_{\mathrm{BC\_TrX}}$ will be asserted. This will reset the stored bits in the corresponding TLB entry to "0"/"1" as designed during the Trojan insertion. Fig. 10(b) shows the simulation result of flipping the stored bit to "0" with respect to (W/L) of Trojan transistor. For a successful flip operation, minimum (W/L) is 4. We have used (W/L) = 5 for faster switching. Fig. 8 shows the layout of 4 SRAM cells and metal tracks allocated for bitlines (horizontal) and wordlines (vertical). One M_4 track per global column of SRAM array connects the target SRAM data node to the Trojan payload transistor which is colocated with sense-amp and other peripherals in the column area. To reset a n-bit wide PFN in the TLB, n Trojan payload transistors are required.

E. Evading Test

During conventional memory testing such as March test, each address is written with different data patterns (e.g., block-0/1, stripe, and checkerboard) [34] and then read to verify memory functionality and correctness (i.e., free of faults such as stuck-at faults and coupling faults). The test pattern consists of a finite sequence of March elements. Each element consists of increasing or decreasing address order of read/write operations covering all memory cells. In a 32-bit wide memory, a striping pattern requires 32 accesses per address, whereas a checkerboard (alternating) pattern requires 16 accesses per address. Therefore, each address may be accessed for a maximum of 16–32 times. This method of testing ensures linear test-time complexity. The Trojan proposed

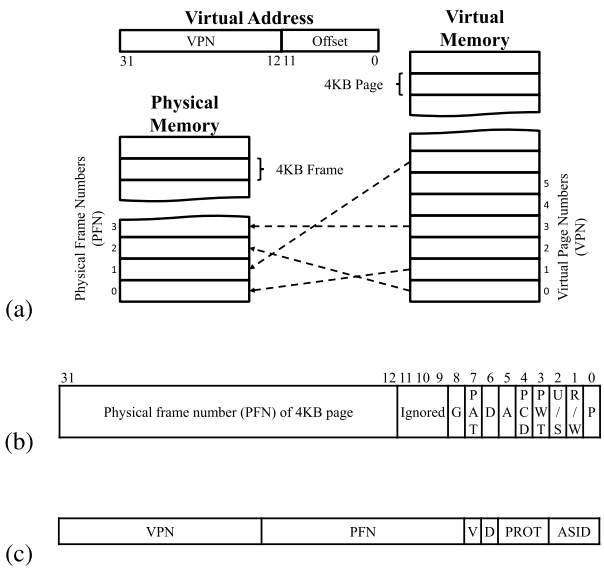

Fig. 11. (a) Paging in a 32-bit address space with 4 KB pages. (b) Page Table Entry (PTE) fields for Intel X86 32-bit paging. (c) Example TLB entry configuration showing VPN, PFN, and the associated metadata bits.

in this article requires 1837 accesses which is significantly higher than 32 and, therefore, evades the test. Considering the overall test time, the proposed trigger requires approximately $1837 \times 64 \times 1024 = 120\,389\,632$ accesses in the worst case per 64-Kb byte-addressable memory. Even if high temperature and V_{dd} is used during the burn-in test, the trigger does not get activated. An advanced trigger with a unique data pattern makes detection more difficult (discussed in Section VI-B).

F. Bypassing Error Detecting Codes

TLBs typically employ parity-based error-detecting codes which can detect the fault injection by the Trojan. The parity bits can also be reset to match the new data to avoid detection. If error-correcting code (ECC) is employed, the Trojan payload could reset the ECC appropriately to bypass detection at the cost of few extra transistors.

IV. Systems Architecture

In this section, we describe the architecture for HarTBleed deployment and the threat model under consideration.

A. Overview of the Systems Architecture

We consider a standard X86 microprocessor architecture with a single core. The memory system is configured with L1 and L2 caches, where the L1 cache is further separated into L1 instruction cache and L1 data cache. The L1 cache is virtually indexed and physically tagged to improve performance. The processor is connected to a DDR system memory. The system runs a standard Linux kernel with paging enabled, with a fixed page size of 4 KB. The memory management unit (MMU) in the microprocessor also has a fully associative TLB to speed up address translation.

The physical memory in a system is very limited in size, and much less than the range of addresses the CPU can reference.

All applications running on a specific CPU architecture is designed for the addressable memory space that the specific CPU architecture can reference. For example, a 32-bit CPU can address 2^{32} memory locations, which is a 4-GB address space. This is known as the virtual address space. However, the installed physical memory (and the physical address space) can be different for different system configurations. Hence the virtual address space is divided into smaller chunks known as virtual pages [Fig. 11(a)]. Typically, the virtual pages are 4 KB each in a 32-bit address space. Each process has its own view of the entire virtual address space. When an application process runs, the required virtual pages are mapped to the physical memory as physical frames [Fig. 11(a)]. The physical memory can have frames belonging to multiple processes simultaneously. The corresponding mapping information (virtual page number $\rightarrow$ physical frame number) is stored in a per-process page table in the OS kernel. The page table entries (PTEs) also store some associated metadata bits signifying protection status, caching information, etc. as shown in [Fig. 11(b)]. When the CPU accesses a memory address, the corresponding page mapping is pulled from the page table in the kernel and cached onto the TLB in the CPU.

The TLB is implemented as a small, fully associative SRAM memory. Each entry in the TLB contains the virtual to physical memory mapping information of a page of the current process in context. TLB entries are configured as shown in Fig. 11(c). In a 32-bit address space with 4 KB pages, the virtual page number (VPN) is the MSB 20 bits of the virtual address. The physical frame number (PFN) bits signify the frame number on the physical memory where that page is mapped. Aside from VPN and PFN, the TLB may also have protection bits specifying read/write/execute permissions, a valid (V) bit indicating if the frame is actually present in physical memory, and other bits such as global (G), cached (C), dirty (D), etc. When a process goes out of context, the TLB is flushed or invalidated. However, to improve performance, some TLBs may also have an address space identifier (ASID) field that helps in caching and lookup of pages from multiple processes without flushing the TLB during context switches.

B. Virtual Memory Layout

The virtual address space of a process is divided into several memory segments as shown in Fig. 12. In a 4-GB virtual address space, the higher order 1 GB of addresses are used by the kernel and is known as the kernel space. The remaining 3 GB are given to the user processes and is known as the user space. The topmost segment in the user space is the stack, which is used to store the local variables and function parameters of the program. As more data are pushed to the stack, it continues to expand downward. Below the stack is the memory mapping segment (mmap). This segment is used to map the contents of files directly into memory. Any application can request access to a file, and the kernel handles the request by performing the memory mapping. This segment is also used to hold the linked libraries, which may be shared across processes. The heap lies below the memory mapping segment and is used to serve memory allocation requests at runtime. The heap grows upward with allocation

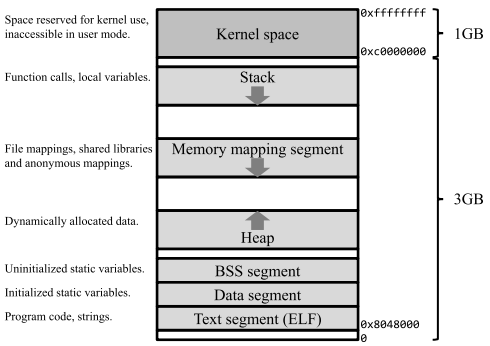

Fig. 12. Process anatomy in the virtual memory.

of data. It is important to note that data allocated in the heap may outlive the function that requested the allocation and thus is a potential source of data leakage. In unmanaged languages such as C, the onus is on the application developer to ensure that the allocated heap regions are freed when they are no longer in use, while in managed code such as C# and Java, this task is automatically handled by a garbage collector. The BSS and data segments below the heap are used for allocating uninitialized and initialized static or global variables, respectively. The lowermost addresses in the address space are used as the text segment, which maps the part of the program binary that contains string literals and executable code. These different segments are maintained as special data structures inside the memory descriptor of the program held inside the kernel space.

C. Threat Model

We describe the threat model under consideration as follows: We assume a standard multiuser multiprocess system. The microprocessor hardware used in the system is compromised with a hardware Trojan (introduced during the manufacturing process) that was undetectable during post-Silicon testing. We assume that the adversary has the knowledge of the Trojan hardware and the specific scenario in which the Trojan can be triggered and activated. This is very likely if the adversary is foundry-sponsored. After the deployment of the chip in the market, the adversary can launch a malicious program to trigger the Trojan for the desired payloads. The adversary has standard user privilege, i.e., no administrator or superuser privileges. The adversary can interact with the existing applications running on the system and can also compile and run his own malicious program with user privileges. It is assumed that the OS kernel is generally secure and cannot be tampered by a normal nonadversarial user. The adversary need not have physical access to the system and can access the system over a network. However, the adversary is knowledgeable about the hardware details of the system in use.

We assume that the hardware Trojan has deep access to the system hardware at the physical layer. The Trojan trigger can tap into and monitor the CPU address and data buses and the payload can reset the TLB entries and their associated metadata to the adversary-known values. The Trojan can only be triggered after a specific event or a series of events have taken place. This trigger event may not be an event which

occurs during normal operation but can be forced to happen by a knowledgeable adversary. The adversary can also deactivate the Trojan after carrying out the attack to prevent further detection or to prevent system faults. It should be noted that the Trojan hardware is dormant and it cannot proactively engage and cause data-leaks or failures. It only serves as a hidden system backdoor, which a knowledgeable adversary can leverage to launch his exploits.

V. Designing HarTBleed Exploits

In this section, we describe and demonstrate the design methodology for HarTBleed exploits.

A. Attack Design

We describe a simple proof-of-concept attack scenario demonstrating the use of the hardware Trojan. In this attack, we focus on extracting data from a process's heap. Let us consider a single victim process which accesses a certain memory location T that is initialized (written) with a specific bit pattern, and then repeatedly accessed. The number of accesses to the location is controlled by a user input N. Let us also assume that the process stores some "secret" data D, which are kept in the process's heap at location S. This "secret" data are not available for the user to view in its raw form. This can happen, for example, in a program which performs encryption/decryption in software by using some cryptographic keys. The keys will not be available to the user, but it may still be loaded into the process's memory. The goal of the adversary posing as the user is to obtain the secret cryptographic key, using some existing input–output operation in the program. The process also reads back and prints data from location P to the user. The goal of this attack is to interact with the victim program by controlling N and force it to print the secret D using the code which normally prints from location P.

It should be noted that this is a very controlled attack where we make certain assumptions about the program structure, which may not be possible in real attacks. However, this attack example shows how a hardware Trojan can be used to perform unauthorized data access. More sophisticated attacks can stem from this simple example. We make the following assumptions in this attack: 1) memory locations T, S, and P reside in different pages, i.e., accesses to each of these addresses creates different page table entries (and TLB entries); 2) addresses S and P have the same offsets into their pages; 3) all these pages are mapped to the physical memory frames and hence have valid and present entries in the TLB; 4) the adversary has knowledge about the cache line that maps to trigger address T, the TLB entry for P, and the Trojan payload used to modify the TLB entry to maliciously map to S.

The malicious hardware Trojan is designed as shown in Fig. 13. The trigger for the Trojan is placed in a L1 d-cache line (in this case, the cache line which maps to T), that monitors its access for N_{SET} times. This action activates the Trojan trigger which generates a signal for the Trojan payload. The Trojan payload performs a malicious mapping in the TLB. It first accesses the TLB entry that stores the mapping for

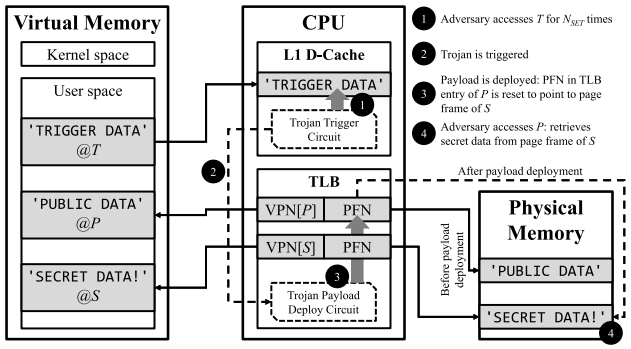

Fig. 13. Trojan design in the system architecture. The attack steps are annotated.

the page of P. The initial mapping is VPN[P] $\rightarrow$ PFN[P]. It then resets the physical frame number for P to that of S. Thus, the effective mapping becomes VPN[P] $\rightarrow$ PFN[S].

We launch the attack as follows. We first provide N as a large value. This makes the process access the location T for N times. This action creates N repeated accesses to the same cache line that is mapped to T, which serves as the trigger for the Trojan. The activated Trojan performs malicious mapping as detailed above. Now, when the process reads from virtual address P, the TLB translates this address to the physical address that stores the secret data D, and the process unknowingly prints out D.

This attack assumes that the victim program coincidentally happened to work in favor of the hardware Trojan, which may not be the case in reality. However, the adversary can use the same attack methodology to write his own malicious program with the knowledge of the address (T) which the Trojan uses as a trigger, the number of accesses required to trigger the Trojan (N), and the location of the data of the victim process in physical memory which he is trying to leak. The secret data may be in the kernel (e.g., page tables) or in other processes that have a presence in the physical memory. We discuss these attacks in Section VI-B.

B. Attack Demonstration

We demonstrate the attack with the help of a simple C program shown in Listing 1. We assume that the program itself is the victim and contains some sensitive "secret" data that the adversary is trying to leak. In the code, `addr_t` is used as the trigger address location (T). This is first initialized (written) with the data "`TRIGGER DATA`." The location (S) for the sensitive data (D) is allocated in the heap as `addr_s`. The data to be leaked (D) is represented as "`SECRET DATA!`" and is copied to `addr_s`. A public data location (P) is also allocated on the heap as `addr_p`, and the data '`PUBLIC DATA`' is copied to it. The sizes of the allocated memory pointed to by `addr_s` and `addr_p` are large enough so that they are in different pages but at the same offset in the pages, and thus have different TLB entries.

We initially loop over and read the Trojan trigger address `addr_t` 2000 times, which is greater than the minimum accesses required by the Trojan trigger as set by N_{SET}. This Trojan trigger monitors the L1 data cache line mapped

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main()
{
    char *addr_t, *addr_s, *addr_p;
    char data;
    int i, len;

    /* N: Number of accesses to trigger address,
     ↪  minimum 1837 */
    int n = 2000;


    /* T: Address of trigger; value initialized */
    addr_t = (char*)malloc(sizeof(char) * 4080);
    strcpy(addr_t, "TRIGGER DATA");

    /* S: Address of secret data */
    addr_s = (char*)malloc(sizeof(char) * 4080);
    strcpy(addr_s, "SECRET DATA!");

    /* P: Address of public data */
    addr_p = (char*)malloc(sizeof(char) * 4096);
    strcpy(addr_p, "PUBLIC DATA");

    /* Before Trojan activation, P points to public
     ↪  data */
    printf("## Before Trojan is activated ##\n");
    printf("addr_t = %p\tvalue = %s\n", addr_t,
     ↪  addr_t);
    printf("addr_s = %p\tvalue = %s\n", addr_s,
     ↪  addr_s);
    printf("addr_p = %p\tvalue = %s\n", addr_p,
     ↪  addr_p);

    /* Trigger the Trojan by N accesses to T */
    for (i = 0; i < n; i++)
    {
        data = addr_t[0];
    }

    /* After Trojan activation, P points to secret
     ↪  data */
    printf("## After Trojan is activated ##\n");
    printf("addr_t = %p\tvalue = %s\n", addr_t,
     ↪  addr_t);
    printf("addr_s = %p\tvalue = %s\n", addr_s,
     ↪  addr_s);
    printf("addr_p = %p\tvalue = %s\n", addr_p,
     ↪  addr_p);
}
```

Listing 1. Example attack code using the Hardware Trojan.

to addr_t. After N_{SET} (= 1837) accesses, the Trojan is triggered, and a signal is sent to the TLB to perform the malicious mapping. Once triggered, the signal stays asserted for several microseconds to facilitate the attack. In the TLB, the Trojan payload is placed at the entry corresponding to addr_p by resetting the PFN field in the entry to the PFN for addr_s. The frame for addr_s stores the secret in physical memory. After the TLB receives the signal to deploy the payload, it performs this malicious mapping. Now when we read from addr_p, it is able to read the secret. We can see the output of the program in Fig. 14. Before the Trojan is triggered, reading addr_p prints "PUBLIC DATA." After the Trojan is triggered, addr_p can access and readback "SECRET DATA!."

C. Evaluation

We model the effects of the hardware Trojan in the GEM5 architectural simulator [35]. The system is designed

```
warn: ignoring syscall mprotect(...)
## Before Trojan is activated ##
addr_t = 0x602010       value = TRIGGER DATA
addr_s = 0x603010       value = SECRET DATA!
addr_p = 0x604010       value = PUBLIC DATA
## After Trojan is activated ##
addr_t = 0x602010       value = TRIGGER DATA
addr_s = 0x603010       value = SECRET DATA!
addr_p = 0x604010       value = SECRET DATA!
Exiting @ tick 180743000 because exiting with
```

Fig. 14. GEM5 console output for the attack code in Listing 1. *P* reads "SECRET DATA" after Trojan activation.

with an AtomicSimpleCPU model running a single core at 1 GHz, connected to a DDR3 physical memory of size 512 MB. Two levels of cache memory are configured, L1 and L2. L1 I-Cache is 32 KB and L1 D-Cache is 64 KB, while L2 cache is 2 MB. The MMU in the CPU is configured with a 64 entry TLB. The GEM5 simulator is compiled and built targeting X86 architecture.

To simulate the Trojan trigger, we added code to the readMem() function in the CPU code to monitor accesses to trigger address addr_t (0x602010). Since the L1 cache is virtually indexed, we can just monitor the cache line mapped to the trigger address. For the Trojan payload insertion, we modified the TLB class code to handle Trojan activation and deactivation states. We added additional class methods to deploy the Trojan payload when activated. In the payload deployment method, a lookup is performed to access the handle for the TLB trie entry corresponding to the "public" data address addr_p (0x604010) with VPN 0x604. Using the trie handle the PFN field (paddr) in the entry is reset to 0x44d, which is the PFN targeted to leak the "secret" data from. The "secret" data are in addr_s (0x604010), which is mapped to the physical address 0x44d010. The payload deployment method is called from the CPU code once the trigger condition is satisfied.

We compiled the attack demonstration code in Listing 1 with GNU *C* compiler for X86 architecture. The attack is evaluated by running the compiled binary in the GEM5 simulator. As seen in Fig. 14, we were successfully able to retrieve the secret data from the process's address space.

VI. Discussions

A. Advanced Hardware Trojan

The Trojan hardware described in this article only exploits the decoded address for trigger. To lower the probability of accidental triggering further, a simple logic circuit can be implemented to generate $V(P_{SET})$ input of Fig. 2 which outputs logic 1 (1 V) only if a specific data pattern (let us say P_{SET}) is sent to the data bus. This will guarantee that the trigger capacitor gets discharged if the L1 data do not match the trigger data. For example, let us consider that the data bus width is 8-bits. Assume that we take four specific data bits to design the trigger logic, e.g., data[0], data[3], data[4], and data[6]. The logic circuit will output "1" if these bits are asserted except data [4] which should be deasserted (Fig. 15). In practice, data bits with low activation probabilities should be

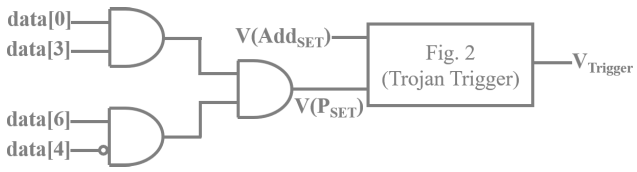

Fig. 15. Example of the logic circuit to generate $V(P_{\mathrm{SET}})$ from a specific data pattern which serves as an input of the proposed Trojan Trigger (Fig. 2).

used to design the trigger logic to lower the overall probability of assertion unless intended. Note that, even if the $\mathrm{Add}_{\mathrm{SET}}$ is asserted N_{SET} times during normal/test conditions, the Trojan will not be activated since a specific data pattern is required to assert $V(P_{\mathrm{SET}})$. Note that in this case, the adversary has to write P_{SET} data pattern to $\mathrm{Add}_{\mathrm{SET}}$ for the first time and then keep reading it to charge the trigger capacitor incrementally. To detect such triggers, the test methodology becomes very complex, since all possible bit combinations have to be tried for repeated times for each location of memory. This results in an exponential test-time complexity.

B. Attack Opportunities

Our simple simulated HarTBleed attack can be extending to other possible exploits crafted using the hardware Trojan.

1) Reading Data From Other Processes: In this attack, the goal of the adversary is to extract data from processes that he does not have access to. For example, in a multiuser system, a user (victim) may be running his personal financial application. The adversary, as a second user on the system does not have access to the victim user's financial application, but he wants to gain access to that application process when it is used by the victim user. To design the exploit, the adversary can write his own attack program similar to Listing 1 with the knowledge of the Trojan trigger and payload specifications. The adversary's attack program and the finance application process pages simultaneously reside in the physical memory. If the Trojan payload PFN maps to the pages of the finance application, then the adversary can trigger the Trojan using his attack program and then force the mapping of one of his own virtual pages to the physical page of the finance application. In this way, he will be able to read out data from the address space of the finance application process. This is shown in Fig. 16, where S may be on a page located in the stack or heap of the victim process. The trigger addresses T and the adversary controlled address P are in the adversary's malicious process.

2) Reading Data From Shared Memory Locations: In this attack, the goal of the adversary is to gain access to memory locations that are shared between processes. Assuming ASLR is off, it is possible to know the location of the shared memory. For example, dynamic linked libraries and files read from disk are mapped to the mmap region of the address space. The pages from the mmap region are shared between multiple processes. At the design time, the Trojan payload can be designed to map to the pages of the mmap region in physical memory. A victim process may request access to a sensitive database file. The request is served by the kernel by mapping

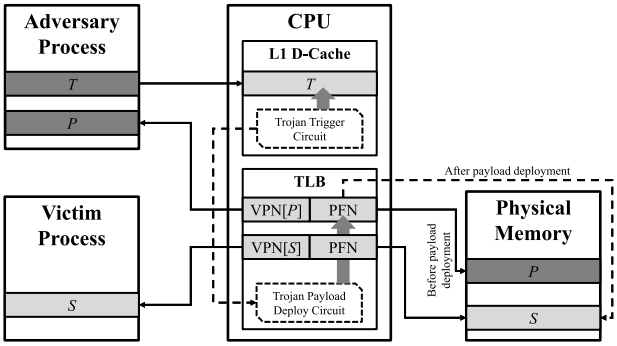

Fig. 16. Using HarTBleed to leak data from other processes. T and P ae in adversary's process, while S is in the victim process. Depending on the memory segment (stack, heap, mmap, kernel) where the page of S is located, the TLB entry of P can be made to point to the frame of S in physical memory to gain access to data from the page of S.

the contents of the file to the mmap region. The adversary in this case can write his attack program to trigger the Trojan and map one of his own pages to the mmap page containing the contents of the database file in physical memory, thereby gaining access to the sensitive data. As shown in Fig. 16, S may be residing in the mmap segment. Adversary can use the Trojan to map his P to point S in the TLB.

3) Reading Data From System Kernel: In this attack, the adversary tries to gain access to sensitive kernel data such as page tables from kernel memory. The kernel space in the virtual memory is mapped to a fixed location in the physical memory. For example, in a 32-bit address space, the higher order 1 GB used as the kernel space is mapped exactly to the higher order 1 GB in the physical memory. Thus, it is possible to determine the location of kernel data in physical memory. The Trojan payload can be designed to point to the pages corresponding to kernel space, and the adversary can write his attack code and launch the exploit as before. In Fig. 16, S may be an address in the kernel space. Although in the figure, S is shown to be in the victim process, it should be noted that the kernel space is shared across all processes since they map to the same region in the physical memory. However, there is a high possibility that this attack may be thwarted since a user mode process is trying to gain access to data which is only accessible in kernel mode. To circumvent this issue, the Trojan can be made to change or bypass the CPU status register that sets the protection rings or execution modes.

4) Resetting Multiple TLB Entries: The adversary can be provided with greater control of page frames to leak from the memory by using a hardware Trojan capable of conditionally resetting multiple TLB entries (refer to Section III-D for hardware details). The Trojan is designed with multiple trigger addresses (T_1, T_2, etc.), and corresponding target victim addresses (S_1, S_2, etc.). Then, depending on the page frame in the physical memory where the sensitive data are mapped (for S_1, S_2, etc.), the adversary can craft his attack by choosing one of his controlled address (P_1, P_2, etc.) for deploying the Trojan payload and triggering the Trojan using the corresponding trigger address. The payload deployment circuit can then choose the required TLB entry based on the trigger address and map to the victim frame. If the adversary is unsure of the

location of the sensitive data, he can hit-and-try all the triggers to reset multiple TLB entries at the same time.

C. Possible Issues or Limitations

HarTBleed has few limitations that may present challenges to an adversary in launching the exploits.

1) Context Switching: We have assumed that the Trojan payload in the TLB remains active while the adversary is carrying out the attack. However, there may be a context switch after the payload is deployed and before the adversary is able to access the sensitive data. During the context switch, if the TLB is flushed, the deployed payload may be removed and later reverted to the original mapping from the corresponding PTE when the adversary's process comes into context. In this case, the adversary may not be able to read the secret data. The adversary may have to run his attack multiple times in order to be successful. This limitation is somewhat alleviated when the TLB uses the ASID fields to cache PTEs of multiple processes at the same time, since the TLB may not be flushed or invalidated as frequently.

2) Detection/Faults During Attack: Gaining access to kernel data may also prove difficult, as mentioned before. The CPU runs processes in different protection rings or modes of operation. For example, in X86 architecture, kernel runs in ring 0 (highest privilege), while user processes run in ring 3 (least privilege). These protection rings are enforced in hardware using registers, which specify the current mode. This is to ensure that code executing in user mode cannot do something outside its purview, such as accessing data from kernel pages. Such actions result in a trappable exception being thrown, which terminates the process. These protection rings can be bypassed if a Trojan is designed specifically to do that transparently in the hardware.

3) Caching Effects: We have assumed a virtually indexed, physically tagged (VIPT) L1 data cache, since that is most commonly used in modern cache architectures [36], [37]. Here, the virtual page number of the address is used to index the cache and lookup the data for faster access. Simultaneously, the virtual address is also sent to the TLB for address translation, so that the page can be retrieved from the lower levels of memory hierarchy in case the data are not already present in L1. This is an advantage that works in favor of the Trojan trigger, since it is easier to infer the cache line mapped to the trigger address. However, the L1 cache may already have a valid mapping for the adversary-controlled (public data) address. In such a scenario, even after deploying the Trojan payload to reset the PFN of public data address to map to that of the victim (secret data) address, that mapping may not be used to retrieve the secret data from physical memory, if the adversary controlled address already has a valid mapping in L1, containing the original data in that address. In this case, the adversary may not be able to read the secret data. However, L1 caches are typically very small, which will eventually replace the cache line mapped to the adversary-controlled address with a different mapping to accommodate data from other memory locations. Once that cache line is remapped, accessing the adversary-controlled address will force the use of the maliciously mapped TLB entry and retrieve the secret data from the lower levels of memory hierarchy (such as the L2 or L3 caches which are physically indexed, or the physical memory). However, if the L1 data cache is not VIPT, it becomes difficult for the adversary to gain access to his controlled addresses. In such cases, the adversary has to infer the virtual-to-physical translation using other means [38], [39].

4) Address Space Randomization: In order to effectively obtain the secret data from a process, the adversary needs to know the location of the data in the address space. However, ASLR is a commonly deployed technique that randomizes the locations of the different memory segments in the address space. Typical ASLR deployments randomize the memory-mapped segment. This can create a challenge for the adversary, since it will be difficult to know the location of the secret data, if the goal is to obtain the data from the mmap segment. OS kernels may also enable KASLR [40], which randomizes the pages in the kernel space. ASLR may be circumvented if there is an existing disclosure vulnerability in the process. Sophisticated attacks [41], [42] have also been shown to thwart ALSR in modern systems.

D. Detecting/Preventing Trojan Attacks

1) Address Scrambling: Since the adversary exploits a predefined memory address to trigger the Trojan, we can scramble the logical to physical address mapping [fixed or generated from a physically unclonable function (PUF)]. This adds a layer of complexity on the attacker to hit the predefined physical address. Randomizing memory-to-cache mapping dynamically has also been shown to be effective in preventing side-channels in the cache [43].

2) Small Validated ECC for TLB: A carefully validated and optically inspected ECC (free of Trojan) can be used to store the ECC for each memory word. If the Trojan performs fault injection/DoS, the ECC will detect it.

3) Analysis of Memory Images During Testing Phase: Memory Trojans are visually tedious to identify due to replication of a large number of memory instances. Machine learning can be applied to analyze the memory bank images to identify anomalies. This approach can worsen the test/validation time.

4) Temperature/Voltage Modulation During Post-Silicon Test: Higher operating voltages accelerate the trigger by lowering N_{SET}. Therefore, the Trojan could be triggered quickly and be detected. Similarly, higher temperatures lower the N_{SET} and aid in detection. At 1.5 V and $T = 90$ °C, N_{SET} reduces from 1837 to 187 for the nominal design. Furthermore, for the worst case process corner which is $\mathrm{S_PF_N}$, N_{SET} becomes 23 for $V_{\mathrm{dd}} = 1.5$ V and $T = 90$ °C. Few points to note here are: 1) we have used a 22-nm PTM model for the transistors used in this work where the nominal voltage is 0.95 V [44]. Therefore, using 1.5 V might cause oxide breakdown and other device issues; 2) considering process variation, lowest N_{SET} can be 23 and the tester can detect few chips out of a bulk with potential Trojans if all the addresses of all chips are hammered at least 23 times; 3) tester still needs to figure out the unique data pattern for effective hammering (for the advanced Trojan) which can be difficult. For example, there are 2^{512} combinations to try with for a 512-bit cache line.

VII. Conclusion

In this article, we presented HarTBleed, an attack methodology that uses hardware Trojans to gain access to sensitive data from a process's memory. We used a capacitor-based hardware Trojan that can monitor accesses to a preselected wordline in SRAM-based L1 cache and proposed payloads to reset TLB entries by fault injection. We analyzed the HarTBleed hardware for robustness to ensure post-Silicon test evasion. Finally, we also designed an HarTBleed system exploit using an attack code taking advantage of the Trojan and demonstrated successful data leakage in the GEM5 simulator.

References

[1] J. Schütte, D. Titze, and J. M. D. Fuentes, "AppCaulk: Data leak prevention by injecting targeted taint tracking into Android apps," in *Proc. IEEE 13th Int. Conf. Trust, Secur. Privacy Comput. Commun.*, Sep. 2014, pp. 370–379.

[2] X. Zhang, F. Liu, T. Chen, and H. Li, "Research and application of the transparent data encryption in intranet data leakage prevention," in *Proc. Int. Conf. Comput. Intell. Secur.*, vol. 2, Dec. 2009, pp. 376–379.

[3] C. Cao *et al.*, "CryptMe: Data leakage prevention for unmodified programs on arm devices," in *Research Attacks, Intrusions, Defenses*, M. Bailey, T. Holz, M. Stamatogiannakis, and S. Ioannidis, Eds., Cham, Switzerland: Springer, 2018, pp. 380–400.

[4] Z. Ma, "CPSec DLP: Kernel-level content protection security system of data leakage prevention," *Chin. J. Electron.*, vol. 26, no. 4, pp. 827–836, 2017.

[5] Y. Wen, J. Zhao, and H. Chen, "Towards thwarting data leakage with memory page access interception," in *Proc. IEEE 12th Int. Conf. Dependable, Autonomic Secure Comput.*, Aug. 2014, pp. 26–31.

[6] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity: Principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, pp. 1–40, Oct. 2009.

[7] (2018). *Data Execution Prevention*. Accessed: (Feb. 26, 2019). [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/memory/dataexecution-prevention

[8] (2003). *Pax address space layout randomization (ASLR)*. Accessed: (Feb. 26, 2019). [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt.

[9] Z. Durumeric *et al.*, "The matter of heartbleed," in *Proc. Conf. Internet Meas. Conf.*, New York, NY, USA, 2014, pp. 475–488.

[10] P. Kocher *et al.*, "Spectre attacks: Exploiting speculative execution," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 1–19.

[11] M. Lipp *et al.*, "Meltdown: Reading kernel memory from user space," in *Proc. 27th Secur. Symp. (USENIX Security)*, 2018.

[12] (2018). *Software Techniques for Managing Speculation on AMD Processors*. Accessed: Feb. 26, 2019. [Online]. Available: https://developer.amd.com/wpcontent/resources/managing-speculation-on-amd-processors.pdf

[13] (2018). *The Big Hack: How China Used a Tiny Chip to Infiltrate U.S. Companies*. Accessed: Oct. 28, 2018, [Online]. Available: https://bloom.bg/2owldii

[14] (2008). *The Hunt for the Kill Switch*. Accessed: Oct. 28, 2018. [Online]. Available: https://spectrum.ieee.org/semiconductors/design/the-hunt-for-thekill-switch

[15] (2018). *Hardware Trojan Designs on BASYS FPGA Board*. Accessed: Oct. 28, 2018. [Online]. Available: http://isis.poly.edu/vikram/vt.pdf

[16] V. C. Patil, A. Vijayakumar, and S. Kundu, "Manufacturer turned attacker: Dangers of stealthy trojans via threshold voltage manipulation," in *Proc. IEEE North Atlantic Test Workshop (NATW)*, May 2017, pp. 1–6.

[17] R. Kumar, P. Jovanovic, W. Burleson, and I. Polian, "Parametric trojans for fault-injection attacks on cryptographic hardware," in *Proc. Workshop Fault Diagnosis Tolerance Cryptogr.*, Sep. 2014, pp. 18–28.

[18] T. Hoque, X. Wang, A. Basak, R. Karam, and S. Bhunia, "Hardware trojan attacks in embedded memory," in *Proc. IEEE 36th VLSI Test Symp. (VTS)*, Apr. 2018, pp. 1–6.

[19] K. Nagarajan, M. N. I. Khan, and S. Ghosh, "ENTT: A family of emerging NVM-based trojan triggers," in *Proc. IEEE Int. Symp. Hardw.-Oriented Secur. Trust (HOST)*, May 2019, pp. 51–60.

[20] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester, "A2: Analog malicious hardware," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 18–37.

[21] (2018). *Trusted Integrated Circuits (TRUST)*. Accessed: Oct. 28, 2018. [Online]. Available: https://www.darpa.mil/program/trusted-integrated-circuits

[22] B. H. M. Beaumont and T. Newby, "Hardware trojans-prevention, detection, countermeasures (a literature review)," Command Control Commun. Intell. DIV, Defence Sci. Technol. Organisation Edinburgh, Canberra, ACT, Australia, Tech. Rep. DSTO-TN-1012, 2011.

[23] S. Zhu, E. Guo, M. Lu, and A. Yue, "An efficient data leakage prevention framework for semiconductor industry," in *Proc. IEEE Int. Conf. Ind. Eng. Eng. Manage. (IEEM)*, Dec. 2016, pp. 1866–1869.

[24] S. Bhunia and M. Tehranipoor, *The Hardware Trojan War*. Cham, Switzerland: Springer, 2018.

[25] 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.

[26] M. N. I. Khan, K. Nagarajan, and S. Ghosh, "Hardware trojans in emerging non-volatile memories," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, Mar. 2019, pp. 396–401.

[27] S. Manne *et al.*, "Low power TLB design for high performance microprocessors," Univ. Colorado Boulder, Boulder, CO, USA, Tech. Rep. CU-CS-834-97, 1997.

[28] X. Wang, M. Tehranipoor, and J. Plusquellic, "Detecting malicious inclusions in secure hardware: Challenges and solutions," in *Proc. IEEE Int. Workshop Hardw.-Oriented Secur. Trust*, Jun. 2008, pp. 15–19.

[29] (2017). *Addressing Test Time Challenges*. Accessed: Oct. 28, 2018. [Online]. Available: https://semiengineering.com/addressing-test-time-challenges/

[30] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar, "Trojan detection using IC fingerprinting," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2007, pp. 296–310.

[31] R. Rad, J. Plusquellic, and M. Tehranipoor, "Sensitivity analysis to hardware trojans using power supply transient signals," in *Proc. IEEE Int. Workshop Hardw.-Oriented Secur. Trust*, Jun. 2008, pp. 3–7.

[32] N. M. Ravindra and J. Zhao, "Fowler-Nordheim tunneling in thin SiO_2 films," *Smart Mater. Struct.*, vol. 1, pp. 197–201, Sep. 1992.

[33] M.-T. Chang, P. Rosenfeld, S.-L. Lu, and B. Jacob, "Technology comparison for large last-level caches (L^3Cs): Low-leakage SRAM, low write-energy STT-RAM, and refresh-optimized eDRAM," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2013, pp. 143–154.

[34] M. Bushnell and V. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*, vol. 17. Cham, Switzerland: Springer, 2004.

[35] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

[36] M. Cekleov and M. Dubois, "Virtual-address caches. Part 1: Problems and solutions in uniprocessors," *IEEE Micro*, vol. 17, no. 5, pp. 64–71, Sep./Oct. 1997.

[37] (2019). *Arm Cortex-A77 Core Technical Reference Manual*. Accessed: Nov. 22, 2019. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0201d/i21752.html

[38] Y. Kim *et al.*, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit. (ISCA)*, Jun. 2014, pp. 361–372.

[39] B. Hopkins, J. Shield, and C. North, "Redirecting DRAM memory pages: Examining the threat of system memory hardware trojans," in *Proc. IEEE Int. Symp. Hardw.-Oriented Secur. Trust (HOST)*, May 2016, pp. 197–202.

[40] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is dead: Long live KASLR," in *Engineering Secure Software and Systems*, E. Bodden, M. Payer, and E. Athanasopoulos, Eds., Cham, Switzerland: Springer, 2017, pp. 161–176.

[41] A. Barresi, K. Razavi, M. Payer, and T. R. Gross, "CAIN: Silently breaking ASLR in the cloud," in *Proc. 9th USENIX Workshop Offensive Technol. (WOOT)*, Washington, DC, USA: USENIX Association, 2015.

[42] Y. Jang, S. Lee, and T. Kim, "Breaking kernel address space layout randomization with Intel TSX," in *Proc. 2016 ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*. New York, NY, USA, 2016, pp. 380–392.

[43] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, no. 5, pp. 8–16, Sep./Oct. 2016.

[44] (2018). *22nm PTM Technology File*. [Online]. Accessed: Feb. 10, 2019 Available: http://ptm.asu.edu/modelcard/lp/22nmlp.pm