GlueZilla: Efficient and Scalable Software to Hardware Binding using Rowhammer

Ruben Mechelinck[1], Daniel Dorfmeister[2], Bernhard Fischer[2], Stijn Volckaert[1], and Stefan Brunthaler[3]

[1] DistriNet, KU Leuven, Belgium
{ruben.mechelinck,stijn.volckaert}@kuleuven.be
[2] Software Competence Center Hagenberg, Austria
{daniel.dorfmeister,bernhard.fischer}@scch.at
[3] μCSRL, CODE Research Institute, University of the Bundeswehr Munich, Germany
brunthaler@unibw.de

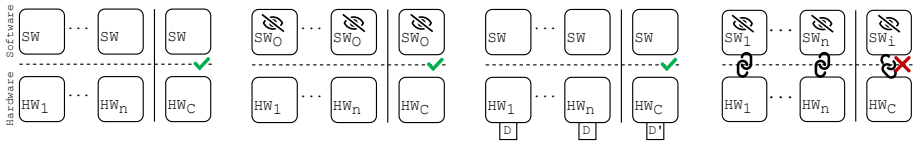
Abstract. Industrial-scale reverse engineering affects the majority of companies in the mechanical and plant engineering sector and imposes significant economic damages. Although reverse engineering mitigations exist, economic damage has not been impacted, indicating that they have failed to address the problem. A closer investigation shows that industrial-scale reverse engineering typically only expends efforts on replicating hardware, since software can often be copied verbatim—no reverse engineering effort required.

We present GlueZilla, a system that binds software to hardware through user-space rowhammer PUFs. GlueZilla transforms programs such that they only exhibit their intended behavior on the single machine they are bound to at compile time. When run on any other machine, the programs will exhibit a different functionality. GlueZilla relies on unclonable machine features and thereby forces counterfeiters to not clone just the hardware but also the software. Cloning both hard- and software drives up reverse engineering costs, thereby also decreasing the economic viability of industrial-scale reverse engineering.

GlueZilla works on commodity hardware and does not rely on expensive hardware components. Our evaluation shows that GlueZilla is effective and incurs 16% run-time performance overhead in a practical case.

1 Introduction

Estimated at 6.4 billion euros in Germany alone, industrial-scale reverse engineering poses a significant risk that endangers economic prosperity [34]. Mechanical engineering companies traditionally invest substantial amounts into research and development, resulting in high-tech products with correspondingly high premiums. To offset these premiums and leapfrog lacking prior research and development costs, industrial-scale reverse engineering has become a billion-dollar business. Based on industry findings, industrial-scale reverse engineering is financially sound and viable. This financial viability, however, rests crucially on the costs involved in reverse engineering versus actual development costs. If stealing intellectual property through reverse engineering



(a) No protection: An attacker can simply copy the software to a cloned device (HW_C).

(b) Obfuscation: An attacker cannot easily reverse-engineer the software but can copy it to a cloned device.

(c) Dongle: An attacker can clone or steal a dongle to run the software on a cloned device.

(d) GlueZilla: An attacker can neither run the software on a cloned device nor easily reverse-engineer it.

Fig. 1: Overview of different protection modes of intellectual property.

costs less than actual development, then reverse engineering pays off; otherwise actual development is preferable.

The costs of reverse engineering can roughly be divided into two parts: (i) costs to reverse-engineer hardware machinery, and (ii) costs to reverse-engineer software components. At present, most reverse engineering efforts are spent on hardware components. A fundamental problem favoring industrial reverse engineering practices is that a machine's operational software runs on both the original and the reverse-engineered machine.

Thus, we propose an approach, called GlueZilla, which protects against reverse engineering of software. In contrast to other protection techniques—such as dongles or obfuscation [23], whose protective power remains low [3,25]—, GlueZilla also protects against unauthorized copying of software, as shown in Figure 1. Software obfuscation thwarts the reverse engineering process but poses no obstacle to software cloning. Dongles, on the other hand, hinder software running across machines, but an attacker can clone them or remove the checking mechanism, which enables copies of the associated software instance to run on any machine.

Forcing industrial-scale reverse engineering companies to reverse-engineer both hard- *and* software components increases overall costs, which is the key goal of GlueZilla. To this end, GlueZilla glues hardware and software together in a novel, cost-efficient, and effective way. The resulting programs behave differently on different machines, having an intentional and an unintentional execution mode. Both modes are fully functional but perform different actions. For each software instance, there is only one associated machine instance which lets the program run its intentional execution mode. If the software runs on any other machine, including exact clones of the associated machine, the program remains in unintentional execution mode. GlueZilla achieves this goal by way of combining user-space-only, rowhammer-based [17] physically unclonable functions (PUFs) [13] with techniques borrowed from software diversity [21]. To the best of our knowledge this combination has not yet been proposed. Through its unique design, binaries compiled with GlueZilla need not carry any anti-theft protection techniques, which more often than not help adversaries during reverse engineering. GlueZilla works on commodity hardware without the need for expensive hardware components.

Summing up, this paper makes the following contributions:

- We present GlueZilla, a system that glues software to hardware through user-space rowhammer PUFs (see Section 6). The resulting glued software poses substantial obstacles to industrial-scale reverse engineering, thus driving up costs.
- We break with the prevalent hardware-oblivious paradigm of prior software defenses by introducing *hardware-dependent* software diversification (see Section 7).
- We discuss how our approach secures programs against reverse engineering and report findings of our experimental evaluation of GlueZilla (see Sections 8 and 9).

2 Background

In this section, we describe the fundamentals of our approach, i.e., rowhammer. Rowhammer is a disturbance error in DRAM modules that allows unprivileged malicious actors to flip bits in physical memory, including in page frames allocated to privileged processes [17]. Each DRAM cell stores a single bit of data, whose value is determined by the charge of the capacitor. Cells naturally lose their charge over time, so the memory controller periodically recharges all rows to preserve their values. Because of implicit electromagnetic couplings in the DRAM chip, one can increase the charge leakage of some memory cells in the chip, i.e., the victim cells, by accessing a certain pattern of neighboring DRAM rows, i.e., the aggressor rows, at a very high frequency, called hammering. In the extreme case, the leftover charge in the victim cell falls below a threshold value and results in a flip of the logical value originally stored in the victim cell, i.e., a bit flip.

Rowhammer affects most commodity DDR3 and DDR4 DRAM chips [16,24,14], including high-end ECC RAM [7], and can be reliably exploited. According to Kim *et al.*, more than 70% of rowhammer-susceptible cells flipped in ten out of ten iterations on DDR3 [17]. Schaller *et al.* show that the number of bit flips increases with temperature, however, the noise level stays constant [29]. Furthermore, the set of flippable bits depends on variances in the manufacturing process of the DRAM chip. This makes the set of flippable bits and the rows that contain these bits a unique and unclonable identifier for the DRAM chip. For this reason, Schaller *et al.* proposed to use rowhammer as an intrinsic PUF [29]. Recently, Jattke *et al.* showed rowhammer-induced bit flips on one out of ten tested DDR5 modules [15]. They conclude, however, that more work is required to reliably circumvent modern rowhammer mitigations, on-die ECC, and higher refresh rates available in DDR5 modules.

After its initial discovery in 2014, several security researchers demonstrated rowhammer-based exploits, for example to gain kernel privileges [30], extract sensitive information from a victim VM using a malicious co-hosted VM [5], gain unauthorized access to co-hosted VMs [27], or to escape from browser sandboxes [8,12,5]. Kwong *et al.* even showed that rowhammer can be used as a read primitive to leak sensitive information of other processes [20].

Most successful rowhammer exploits take three steps. During *memory templating*, the attacker constructs a list of vulnerable cells by probing the DRAM [35]. Next, the attacker loads a virtual memory page of the victim process into a physical page frame with a vulnerable cell at the desired offset. Attackers commonly resort to *memory*

massaging techniques to bring the operating system's page frame allocator into a predictable state, and subsequently trigger a page allocation in the victim process. Finally, the attacker starts *hammering* the correct aggressor rows and causes the expected bits to flip in the page of the victim process's page.

3 Rowhammer Properties

In this section, we lay out a number of properties of the rowhammer effect, which we will later use as building blocks for stealthy, machine-aware computations. In the past, rowhammer was primarily used for targeted exploitation, because it allows stealthy cross-process memory modifications. Apart from this property, the rowhammer effect has many other interesting properties, including:

1. **Unclonable and unique bit flip pattern** Whether a DRAM cell is susceptible to rowhammer or not depends on many uncontrollable variations during the manufacturing process of the DRAM chip. The distribution and sensitivity of rowhammer-susceptible memory cells is, therefore, unclonable and practically unique for large memory regions.
2. **Bit flips in the full memory range** Rowhammer-susceptible cells and their corresponding aggressor rows are distributed throughout the whole physical memory. This allows programmers to use rowhammer to perform intra- and inter-process alterations of code and data. Inter-process bit flips, in particular, are well documented in the literature on rowhammer attacks.
3. **No static knowledge of victim cell locations** For many rowhammer patterns, one can deduce the victim row based on knowledge of the location of the aggressor rows. For example, in the single-sided, double-sided, one-location, many-sided, and half-double hammer patterns, the victim rows are located next to the aggressor row(s) [17,30,11,18,9,28]. However, the precise location of rowhammer-susceptible cells is still unknown because it can be any variation of the 65,536 bits in the row (assuming a DRAM row is 8 KiB long). In more complex hammer patterns, e.g., non-uniform hammering [14], one cannot even deduce the precise location of the victim row(s).
4. **Bit flip success depends on execution speed** If the aggressor accesses happen too slowly, e.g., because the CPU executes different instructions (of any process) in between, the victim cells will not lose enough charge in between row refreshes, resulting in no bit flips.
5. **Bit flip information isolated from OS and CPU** Bit flips in memory happen directly in DRAM without memory write instructions. They are the result of successive memory reads to different locations in physical memory. The OS and CPU are, therefore, unaware of any modifications in memory due to bit flips. One important consequence is that the OS will not trigger any faults, for example, when flipping bits in pages mapped as read-only or pages that are mapped in another process's address space.

We introduce the idea of using these rowhammer properties for purposes other than exploitation. Schaller *et al.* already explored the application of property 1 to build

a rowhammer PUF [29]. Properties 2 and 5 allow programs to make modifications to any process, including their own, without the OS or the CPU noticing. These properties, furthermore, enable a new form of self-modifying code, even in non-writable code pages. Due to property 3, the static image of the binary does not contain any direct information on which modifications are performed at run time. These properties limit the amount of information one can extract from the static image about the run-time behavior. Properties 4 and 5 frustrate several dynamic analysis techniques.

4 Case Study: GlueZilla

We now present GlueZilla[4], a new approach to prevent software theft by binding a software instance to a single associated machine instance. Hackers who engage in industrial-scale reverse engineering specialize in creating replicas of expensive pieces of machinery, including their software. Currently, these hackers spend most of their efforts on reverse engineering and copying the machine's hardware components. Once they have copied these components, they can run the original machine's software on the replica machine as-is. GlueZilla can make reverse engineering much more difficult and costly by forcing hackers to reverse-engineer the hardware and subsequently modify the original software before it can run on a copied machine. One of the unique properties that sets GlueZilla apart from most traditional hardware-software binding mechanisms in use today (e.g., those that use dongles or trusted platform modules (TPMs)) is the stealthiness of the interaction between the GlueZilla-protected software and the hardware it is bound to. Whereas protected user-space software interacts with dongles and TPMs over well-defined and easily identifiable interfaces, GlueZilla-protected software uses standard memory access instructions that can hide in plain sight.

The protection system we present in this section is just one implementation of our hardware-software binding approach. This implementation does not offer bulletproof protection, as we will show in Sections 8 and 10. GlueZilla is vulnerable to attackers who can create full-memory snapshots directly on a machine that runs the associated software instance.

5 Threat Model

For this case study, we make the following assumptions about the host system, the attacker, and the protected applications.

We assume the host system uses commodity rowhammer-susceptible DRAM modules. Related work shows that this includes most of the DDR3 and DDR4 DRAM modules in use today [16,24,14], including those that use ECC [7]. Given the fact that no general rowhammer mitigation exists as of yet [7,9], we consider this a realistic assumption.

We assume the protected application is a user-space program, deployed on a machine that is part of industrial machinery, such as an assembly line. We consider an

[4] `https://github.com/COMET-DEPS`

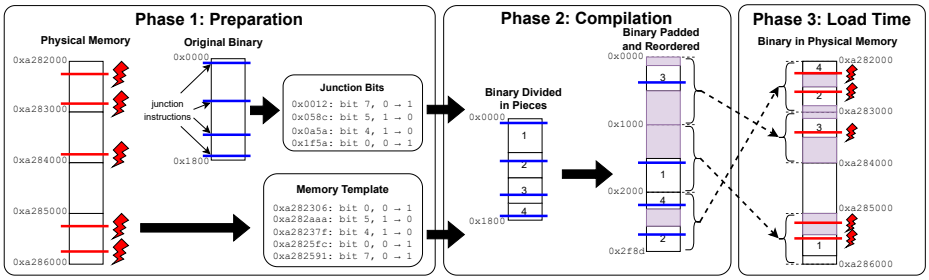


Fig. 2: GlueZilla generates a memory template on the target device and compiles the protected program such that all developer-specified junction instructions can be loaded into rowhammer-susceptible DRAM cells at run time.

attacker performing industrial espionage, i.e., aiming to obtain a copy of the binary image of the protected software, for example, by stealing it from the storage of a genuine machine or by intercepting software updates. The attacker is, furthermore, able to acquire or create machine clones that meet the same specifications as the original machine. They will, however, show different results compared to the original machine when querying physically unclonable components, such as the DRAM modules. The attacker's goal is to deploy fully functional copies of the protected software on clones of the original machine. The attacker does not have access to the original source code.

6 GlueZilla: Overview and Design

Overview GlueZilla programs contain two functional operation modes: an *intentional execution mode* and an *unintentional execution mode*. The intentional execution performs the original, intended purpose of the program. The unintentional execution performs actions differing from the intended purpose without obvious signs that something is wrong with the program, e.g., program crashes. These two operation modes only differ at unsuspicious-looking program points. For example, a branch condition in the intentional execution could be inverted in the unintentional execution, or a call in the intentional execution to one function could be changed to a call to another function in the unintentional execution. If chosen specifically so as to not arouse suspicion, it becomes difficult for an attacker to distinguish unintentional from intentional behavior. Reliable distinction of both behaviors requires expert understanding of the software's intentional behavior. Adversaries typically lack this knowledge since gaining this knowledge is the primary reason for reverse engineering the program in the first place.

Once our GlueZilla compiler has built the protected GlueZilla program, it executes the unintentional behavior by default. The program uses rowhammer-induced bit flips to switch to its intentional execution mode at run time. GlueZilla leverages several of the rowhammer properties we laid out in Section 3 to implement the run-time switch from the unintentional to intentional execution mode. Property 2 allows GlueZilla to use rowhammer-induced bit flips to morph the unintentional code into the intentional code. For the transition to intentional code to be successful, the pro-

tected program requires specific bits to flip. Due to property 1, the desired bit flip pattern is unclonable and only present at the associated machine for which we compiled the software instance. This ensures the intentional execution mode is only reconstructed on this physical machine. The static binary image, furthermore, does not contain any information about the intentional code, as explained by property 3. Finally, property 5 states that the CPU does not explicitly perform the required memory modifications. There is, thus, no need for any unstealthy branching instruction inside the protected program itself that conditionally reconstructs the intentional code. As explained in property 5, these memory modifications also work in non-writable code pages.

Design Conceptually, GlueZilla operates in several phases shown in Figure 2 that are required to create binaries glued to a specific hardware instance, as described above:

1. *Pre-Compilation Phase*:
   (a) Memory Templating Stage: In this stage, we gather information about the specific location of bit flips in the physical memory of the target device.
   (b) Junction Instruction Selection Stage: A junction instruction is a pair that consists of (i) the original, intentional instruction, and (ii) its unintentional instruction counterpart. During this stage, the compiler selects suitable pairs, either automatically through random selection or manually.
2. *Compilation Phase*: In the compilation phase, the compiler fuses the information of the memory templating with the selected junction instructions. Since both instructions and possibly even functions need to be put into specific locations, the compiler needs to perform advanced code scheduling techniques.
3. *Load Time Phase*: Before we can execute a program, we need to perform the required memory massaging at load time. This phase ensures that all required physical page frames from user space are allocated. Then, we load all of the code pages into the expected page frames.
4. *Run-Time Phase*: Rowhammering is used on junction instructions, such that the program's control flow is diverted from unintentional to intentional execution mode on the target device.

7 The Implementation of GlueZilla

In this section, we describe the previously mentioned phases in greater detail.

7.1 GlueZilla Memory Templating

Our approach requires a priori information about the behavior of rowhammer on the target device. We gather this information by measuring the rowhammer-susceptibility of the device's physical memory using a number of hammer patterns that are known to induce bit flips [17,30,11,18,9,28,14]. For all vulnerable cells we discover, we collect the parameters required to reproduce the bit flip later on, including (i) the physical address and (ii) the bit index of the vulnerable byte, (iii) the corresponding aggressor row addresses, (iv) the hammer pattern, and (v) the flip direction. We repeat this scan several times to assess the repeatability of each bit flip. Our memory templater and additional analysis tools consist of approximately 3,500 lines of C++ code.

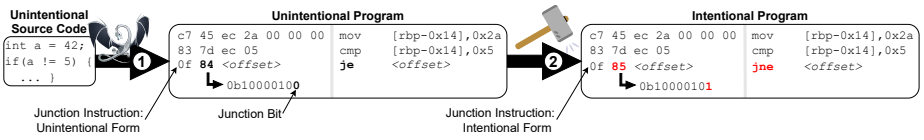


Fig. 3: At run time, GlueZilla uses rowhammer to transform junction instructions from their unintentional into their intentional form.

7.2 GlueZilla Junction Instruction Selection

Junction instructions define the points where the intentional and unintentional execution of the program differ. The effort to reverse-engineer the protected program increases with greater difference between both execution forms and, therefore, the number of junction instructions. Selecting these instructions is hard to automate because we want to ensure that these differences are not obvious to reverse engineers. We want to guarantee that the program is not actively harmful, even when it executes its unintentional code.

An additional challenge lies in the sparse distribution of rowhammer-susceptible cells across the DRAM. In practice, this distribution limits us to a single bit flip per junction instruction. We refer to this bit as the junction bit. Fortunately, if chosen well, flipping one bit in a few junction instructions often suffices to make the intentional and unintentional executions of a program differ substantially. This is especially true on architectures with dense instruction sets (e.g., x86). On such architectures, there are many valid machine instruction sequences that differ in only one bit from several distinct sequences with different semantics. The individual instructions in the new interpretation of the sequence can have a different encoding length as long as the total sequence is still valid, i.e., the bit flip did not introduce invalid instructions in the code sequence. This allows us to create a different control flow (e.g., by changing loop predicates, jump or call targets, etc.) and data flow (e.g., by changing arguments, constants, etc.) between the intentional and unintentional executions.

We classify junction instructions w.r.t. their position-related constraints into the following two classes:

1. Unmovable junction instructions (U-JI) only require that their junction bit be aligned with a suitable rowhammer-susceptible DRAM cell. Figure 3 illustrates an unintentional `je` instruction, which becomes an intentional `jne` instruction after the bit flip (2). This instruction should be placed into a DRAM cell in which we can flip bit 0 of the instruction.
2. Movable junction instructions (M-JI) create inter-instruction placement constraints. Figure 4 illustrates an unintentional call to `foo`, which becomes an intentional call to `bar` after the bit flip (2). If we choose bit index 8 of the second byte of the branch target operand as our junction bit, the target changes from 0x1**0**00 to 0x1**1**00 during the run-time transition from the unintentional to the intentional execution mode. This puts a relative placement constraint on the two call targets (`foo` and `bar`) as they must be exactly 0x100 bytes apart.

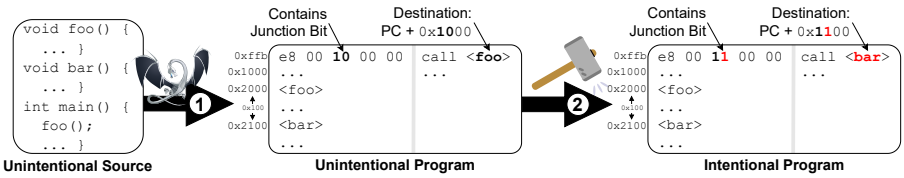


Fig. 4: Junction bits inside control transfer destinations add constraints to the relative position of the destinations.

To select junction instructions, we explored an automatic and a manual operation mode. The automatic operation mode selects a provided number of unmovable and movable junction instructions randomly and uniformly across the binary. This mode, however, provides no guarantee about the behavior of the software on cloned machines. It may, in the worst case, cause harm to the machine, or its environment. Instead of ignoring such a security-critical scenario, we also support a manual operation mode.

In manual operation mode, we rely on a developer to modify their source code to unintentional form and keep track of the changed instructions, i.e., the junction instructions. To facilitate this task, the developer can construct the unintentional behavior by adding subtle alterations to the existing control and data flow graph of the intentional behavior. To increase the number of available alterations, a developer can perform semantic-preserving transformations, such as data structure or function parameter permutation.

7.3 Compiling GlueZilla Binaries

The transition from intentional to unintentional execution mode only works if we can position the program's code in physical memory such that every junction bit is stored in a rowhammer-susceptible DRAM cell. This poses several challenges:

Challenge 1: Bit Alignment Machine instructions are aligned to byte boundaries in physical memory. Thus, if the X-th bit within a junction instruction is a junction bit, then it is only possible to store this bit in the Y-th cell of a DRAM row if X and Y are congruent modulo 8 (i.e., $X \equiv Y \pmod 8$).

Challenge 2: Page Alignment For practical reasons, and although not strictly required, code pages within a program binary should be aligned to page boundaries when loaded into virtual and physical memory. This means that if a junction instruction is at page offset X within a code page of a program binary, said instruction can only be stored at byte offset Y in a page frame if X and Y are congruent modulo the page size.

GlueZilla: Compiler + Loader We developed a compiler toolchain and a binary loader to tackle these challenges. To overcome the first challenge, our compiler consults the memory template of the target device to find candidate cells for each junction

bit that complies with the required bit alignment. We use this list of candidates to constrain our layouting passes. These layouting passes assign each code page to a physical page frame with a suitable candidate DRAM cell, where that page's contents must be loaded. We emit information about these assignments alongside the program binary in the form of a *loader map*. Afterwards, they apply dead `nop` insertion and instruction/basic block reordering [21] to move each junction instruction to the page offset of the candidate cell, which solves the page alignment problem. We implemented all logic related to the custom binary layout in the Machine Code (MC) sub-project of LLVM 11 in almost 1,200 lines of C++ code.

7.4 Loading GlueZilla Binaries into Memory

GlueZilla only works if it can guarantee that every junction instruction is stored in its intended physical memory cell. However, user-space programs have no direct control over where their code pages get loaded into physical memory by the operating system. We tackle this challenge using *memory massaging* from user space, similar to prior rowhammer attacks [30,35,27,11,20,28]. In detail, we massage the operating system's page frame allocator so we can load every code page that appears in our loader map into its intended physical frame.

The massaging techniques presented in related work are built for attack scenarios where the attacker controls the massaging process, and tries to load pages of the victim process in a specific physical page frame [30,35,27,11,20,28]. These techniques typically leave a time window between the freeing of rowhammer-susceptible page frames by the massaging process and the reallocation of said frames by the victim program. In this time window, the operating system could allocate the desired frame to other programs running in the background. Most attacks tolerate this probabilistic behavior, as they typically require only very few rowhammer-susceptible page frames and one can simply relaunch the attack without anyone noticing.

In contrast, programs protected by GlueZilla might require rowhammer-susceptible cells in many different page frames and, therefore, probabilistic massaging is inadequate. To address this challenge, we design a *deterministic* massaging technique by performing the massaging step from within the same process whose page frames we want to hammer. This decision closes the time window, because we do not have to deallocate frames only to trigger their reallocation in a different process. Our technique allocates pages until we find all physical page frames in the loader map and their corresponding aggressors.

We implemented our binary loader and supporting tools in approximately 1,200 lines of Rust code.

7.5 Rowhammering GlueZilla Binaries at Run-Time

After loading all code pages into their associated page frames, we can execute the program and use rowhammer to transition between execution modes. We flip the junction bit by rapidly hammering the junction instruction's corresponding aggressor rows with the chosen hammer pattern.

In most cases, this operation works only in one direction: it can transform a junction instruction from its original unintentional form into its intentional form but not vice versa. Thus, after executing a junction instruction once, it would remain in its intentional form until the program stops. However, it is also possible to reverse the transformation after executing the junction instruction. For example, we could reset the contents of page frames containing junction instructions by either explicitly reloading the code from disk or by forcing the operating system's pager to evict the frame and, subsequently, page it back in. In this case, we could restore the junction instructions' unintentional forms, either periodically or immediately after executing them.

Other degrees of freedom lie in the timing of the transition and the placement of the hammering instructions. We could invoke the transformation at program startup, just before executing the junction instruction, or anywhere in between. Similarly, we can invoke the transformation from within the GlueZilla-protected process or from an external process. By separating the transformation code in time and space from the junction instruction it transforms, we can increase stealthiness and resilience against reverse engineering. However, too much separation, especially in time, can facilitate memory snapshotting attacks, as we explain in Section 10.

While rowhammering, the protected program uses approximately all available memory bandwidth. To enable a successful transition, no other program running on the system should consume a significant portion of the memory bandwidth while the protected program is hammering. Multiple GlueZilla protected programs can run on the same system simultaneously as long as they adhere to this constraint, i.e., they should synchronize their rowhammer-based transitions.

8 Security Evaluation

An attacker can try to reverse-engineer the protected binary to extract secrets or to remove the machine dependence. We describe two attackers that have access to increasingly powerful analysis techniques: (i) static analysis and (ii) dynamic analysis on a cloned machine.

Static Analysis Due to rowhammer properties 2 and 3, the binary is only an image of the unintentional program and does not contain any direct information about the transformations performed by GlueZilla to transition to the intentional execution mode.

The attacker can perform a differential attack on multiple binary instances that are compiled for different machines. However, the unintentional behavior and the set of junction bits are exactly the same for all instances. Only the order of the code fragments and the amount of padding between them differs. These reorderings do not provide any information disclosing junction instructions.

By inspecting the loader map, the attacker can learn which page frame is assigned to each code page. However, due to property 1, the attacker knows neither the location nor the number of rowhammer-susceptible cells these page frames contain. Additionally, due to rowhammer property 3, even if the attacker learns the location of the victim row based on the aggressor rows, they cannot reliably infer which bit flips are induced by them. Assuming a row is 8 KiB long (65,536 bits), the total number

of possible variations is given by: $\sum_{p=1}^{M}(P_p^N)$ with M being the attacker-assumed maximum number of bit flips in a row and N being the number of bits giving valid instructions when flipped. Let us assume the compiler emits code fragments with 35% code and 65% `nop` instructions[5] and that 50% of the code bits result in valid instructions when flipped. On a single channel system, this gives 11,469, 131,537,961, or 1,508,345,821,725 possible bit flip variations if the attacker assumes a maximum of one, two or three flips, respectively. For every considered bit flip, the attacker has to reason about its effect on the program semantics, which is difficult to automate. This difficulty renders a brute-force attack impractical due to prohibitive time requirements. When using complex hammer patterns, the victim row is not adjacent to the aggressor row and bit flips can happen anywhere in the program, providing additional protection.

We conclude that a static adversary cannot obtain sufficient information regarding the intentional behavior using static analysis alone.

Dynamic Analysis on Cloned Hardware In this scenario, the attacker has a clone of the software instance and a clone of the machine associated with the software instance. The protected program is always loaded at the same physical addresses, dictated by the loader map. Property 1 states that the number and location of rowhammer-susceptible DRAM cells are different between any two systems. Therefore, on a cloned machine, the junction bits will not reside in the required rowhammer-susceptible cells, leading to either no bit flips or unexpected bit flips. The attacker, therefore, has no incentive to perform any type of dynamic analysis on this program execution since they cannot learn anything about the intentional behavior.

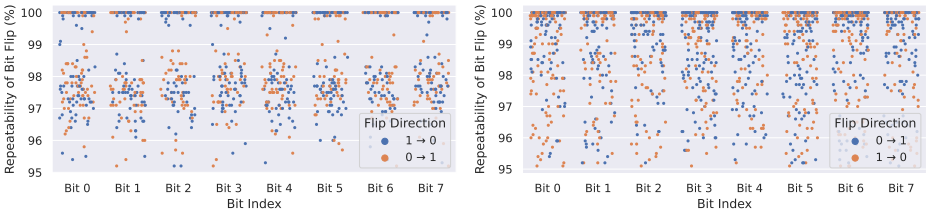
9 Empirical Evaluation

We performed our experiments on both DDR3 and DDR4 systems. For DDR3 memory, we used the double-sided hammer pattern to induce bit flips [30]. Due to the presence of rowhammer mitigations on DDR4, e.g., Target Row Refresh (TRR), we required more advanced hammer patterns, such as half-double sided hammering [18], many-sided hammering [9,28], or non-uniform hammer patterns [14]. First, we evaluated the repeatability of the rowhammer-susceptible DRAM cells and showed that these can be used as a foundation for GlueZilla. Second, we evaluated the functionality of GlueZilla by protecting `coreutils` programs and performance on the SPEC CPU 2017 benchmark suite.

Bit Flip Repeatability As we require individual bits to flip reliably, we must ensure a high repeatability. In several experiments conducted on multiple DRAM/machine combinations, a share of rowhammer bit flips is highly reliable, i.e., flips in all tests, under specific operating conditions.

For example, Figure 5a shows the results of two rowhammer experiments, executed using the same 54 MiB of consecutive DDR3 memory of vendor B. Per experiment, we executed 1,000 iterations of a double-sided rowhammer test, with the only difference

[5] Average size overhead for 100 junction instructions in SPEC CPU 2017 (see Section 9)



(a) Double-sided pattern on DDR3 DRAM (b) Non-uniform pattern on DDR4 DRAM

Fig. 5: Bit flip repeatability with 1,000 iterations per flip direction. Each dot represents a bit flip at a specific location in DRAM. They are grouped by their bit index within the respective directly addressable byte.

being the initialization of the victim and aggressor rows so that we can find bit flips in both directions ($0 \rightarrow 1$ and $1 \rightarrow 0$). While some bit flips occur in a very small number of tests, which can be considered as noise, several bits flipped in all iterations and thus are suitable for use in GlueZilla. Figure 5a shows that there are several reliable bit flips for each bit index in both flip directions.

We executed the same experiment on DDR4 memory of vendor A, although for just 800 KiB of consecutive memory and using a Blacksmith-generated non-uniform hammer pattern. Since DDR4 is more susceptible to bit flips than DDR3, the smaller memory area was sufficient to induce a comparable number of bit flips. As evidenced by the results shown in Figure 5b, the results align with the ones obtained for DDR3. In consequence, we find that both DRAM types offer sufficiently reliable bit flips.

Among other factors, temperature and aging can influence the repeatability of rowhammer bit flips [17,29,24,33,4]. We do not consider these factors in this paper and want to refer to future work for possible error correction mechanisms.

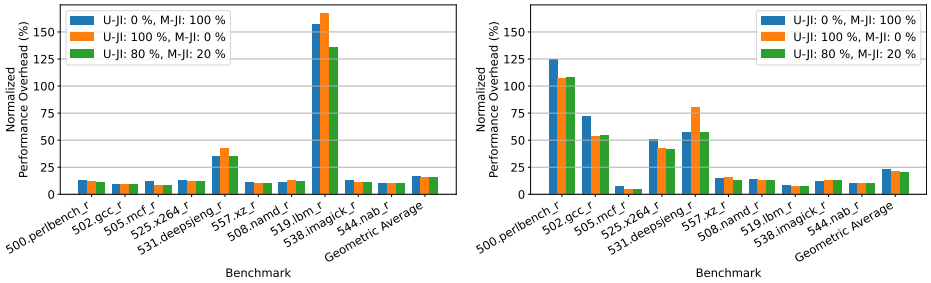
Functionality Evaluation To evaluate the practicality of GlueZilla, we applied GlueZilla to the `coreutils` programs `ls` and `cp`. We constructed an unintentional behavior for both programs and used the manual operation mode described in Section 7.2 to select the junction instructions.

For `cp`, we made the unintentional variant of the program delete rather than copy the source file. We implemented the unintentional behavior by modifying:

- Two function signatures to add an extra pointer indirection for some arguments.
- The struct layout of two structs to put common fields at the same offset.
- A function call that serves as movable junction instruction with the remove function as unintentional destination and the copy function as intentional destination.

For `ls`, we built the unintentional behavior by adding disruptive operations to the intentional behavior so it does not print the current directory correctly. We implemented this behavior by adding:

- A conditional function call that increments the pointer to the element to print. The condition always evaluates to true, so the unintentional form effectively skips an en-



(a) 100 junction instructions.

(b) 1% junction instructions.

Fig. 6: Performance impact of GlueZilla on SPEC CPU 2017.

try in each iteration. This condition functions as an unmovable junction instruction in which we flip an operand, so it evaluates to false in the intentional execution.
- A time dependence that jumps over the print code between 3 p.m. and 6 p.m. This jump is a movable junction instruction with the print code as intentional destination and the code thereafter as unintentional destination.
- An increment of two bytes to the char pointer pointing to the current entry's name. Each entry will thus miss the two starting letters in its name. The increment instruction is an unmovable junction instruction in which we flip the constant from two to zero.

We performed this experiment on both a DDR3 and a DDR4 system. For each system, we took two different DRAM modules that contain rowhammer-susceptible cells of which one is associated with the program by the GlueZilla compiler. At run time, we verified that the intentional execution is only unlocked on the associated module while on the other module the program performs its unintentional execution.

Performance Evaluation Our current prototype performs all massaging and hammering at program launch, so these steps only add overhead when the protected program starts. The start-up overhead heavily depends on the current state of the memory allocator. For the average case, when most of the required page frames are free, this takes about 2.7 seconds for a program requiring about 1,000 page frames.

At run time, the custom binary layout adds performance overhead because of reduced cache utility due to reduced code locality. We evaluated this overhead on all C benchmarks of the SPEC CPU 2017 benchmark suite (we omit C++ benchmarks as our prototype does not currently support C++ exceptions). Our machine has an Intel Core i7-2600 CPU and runs Fedora 39 with Linux 6.6. We used four DRAM modules with 12 GiB of DDR3 memory in total, operating at 1333 MT/s. To reduce interference with other processes, we disabled SMT, turbo boost and isolated one core to which we pinned the protected program and set the scaling governor to performance.

For each benchmark, we used the automatic operation mode, which selects a provided number of unmovable and movable junction instructions randomly and uniformly across the binary. This mode allows us to measure the effect of the custom

layout on large quantities of junction instructions. Furthermore, we defined the unintentional form equal to the intentional form so we do not require the rowhammer step in our layout-related overhead measurements.

Figure 6a shows the run-time performance overhead of binaries containing 100 uniformly distributed junction instructions, each requiring one junction bit. We divide these in unmovable and movable junction instructions in three configurations, represented by the three bars per benchmark. For each configuration, we ran three experiments per benchmark program, each with a different set of junction bits and report the geometric average. Figure 6b shows the overhead when 1% of all machine instructions in the binary are junction instructions. This setting is the maximum amount of junction instructions we could select with a memory template containing about 200,000 rowhammer-susceptible DRAM cells.

Our current compiler prototype does not yet take into account any optimization regarding code placement and alignment, cache prefetching and utilization, branch probabilities, etc. It may even deteriorate the optimizations performed by earlier compiler passes, resulting in an almost worst-case overhead. This explains why both figures show a relatively high impact for the modified binary layout. However, in the realistic scenario of 100 junction instructions, the run-time performance overhead is limited to 16%. The geometric average of the size overhead of all benchmarks over the three configurations is 185% for 100 junction instructions and 843% when 1% of all instructions are junction instructions.

We investigate these results using the hardware performance counters available in our benchmark machine. In our experiments using 100 junction instructions, all measured performance loss is caused by a variety of effects related to suboptimal code layout, including instruction cache misses, iTLB misses, and CPU frontend stalls. For 519.lbm_r, however, we found that almost all overhead is caused by branch resteers in the CPU frontend. 519.lbm_r is the smallest benchmark program and only includes around 1,800 assembly instructions that fit on three 4 KiB pages in the baseline binary. Adding 100 junction instructions to these small binaries results in many small code fragments, often containing only one or two assembly instructions separated by large amounts of padding. To fit the spare distribution of rowhammer-susceptible cells in physical memory, the padding can be multiple KiB long. As suggested by Godbolt, this greatly underuses the available entries in the branch target buffer (BTB) leading to many branch mispredictions [10]. We found that the protected program had approximately 12,000 times more branch resteers compared to the baseline version. During a branch resteer, the branch address calculator updates an entry in the BTB that caused a branch misprediction or misidentification and restarts instruction fetching. These additional resteers account for 10% of all reference clock cycles.

In our experiments where 1% of all instructions are junction instructions, we found that more benchmarks are affected, albeit not as much as 519.lbm_r with 100 junction instructions, by branch resteers, most notably 500.perlbench_r, 502.gcc_r, 525.x264_r, and 531.deepsjeng_r. 519.lbm_r in these experiments only has approximately 18 junction instructions (1% of its 1,800 assembly instructions) and, therefore, has less run-time overhead compared to our evaluation using 100 junction instructions.

10 Discussion

Implementation Limitations Our prototype presented in Section 7 only focuses on the functional requirements and, therefore, has some limitations related to security and performance.

We choose to construct the required binary layout during the compilation phase, because all necessary information is available at this stage. This approach, however, results in large binaries of which a large portion is `nop` padding. Alternatively, we could put each page in a separate ELF segment and directly load it in the assigned page frame. This approach only has intra-page padding but requires more information to transfer between the compiler and the linker.

The padding also identifies the edges of each code fragment which either contains a single junction instruction or is the destination of a movable junction instruction. This information slightly narrows down the adversary's search space for junction instructions, which makes it easier to reconstruct the intentional program. For example, the attacker can identify a potential intentional destination when it is preceded by a large amount of padding in the unintentional binary. Subsequently, they scan the unintentional binary and match all target addresses of direct jump instructions with the address of the potential intentional destination. If both addresses differ only in a single bit, the adversary can with great certainty assume the jump is a movable junction instruction with the identified code fragment as its intentional target.

We can eliminate this attack by limiting the attacker's capability to identify potential intentional targets based on the left-over intra-page padding. One approach is to substitute the padding with dead code or with used code that does not have any positional constraints. In both cases, it should not be obvious where the substituted padding ends and the intentional target starts by making the substituted padding as benign looking as possible, for example:

1. Whenever the padding ends with a return instruction, there should be a function symbol associated with the succeeding code, i.e., the intentional target.
2. If the padding ends with an unconditional jump, the padding should also include (dead) jumps to the succeeding code.
3. When the padding consists of dead code, we can include apparent data dependencies between the dead padding code and the intentional target's code.
4. If the padding consists of used code, we can include data dependencies delimited by an opaque predicate between the padding and the intentional target.

Our prototype requires access to the Linux pagemap, for which we launch the program with `sudo` and later switch to an unprivileged user. Because this is bad security practice, we could leverage a kernel module that exposes a memory massaging interface limited to the memory of the current process. Such an interface would not give new capabilities to the attacker because it provides the same functionality as massaging from user space with the information in the loader map.

To reduce run-time overhead, we could define the binary layout problem as an optimization problem to increase code locality. The optimization variables include: the size and order of the code fragments, the location of the junction instruction within the fragment, and the selection of the rowhammer-susceptible cell w.r.t. its page frame offset.

Because we merely want to demonstrate the general idea of using rowhammer for stealthy computations, we rely on the developer to manually analyze the code base and select enough junction instructions (see Section 7.2). This could be an enormous effort depending on the desired level of protection, i.e., the distance between the intentional and unintentional form. Automating this process requires multiple analyses, including a syntactical analysis of the source code to explore neighboring valid candidate programs, and a semantical analysis to evaluate the conspicuousness of each candidate and guarantee the safety for machine and environment of the unintentional form. Automation gets easier if we relax the conspicuousness requirement and allow, for example, program crashes or infinite loops without dangerous side effects. In this case, we could select condition statements as junction instructions.

Reliability In Section 9, we showed the existence of rowhammer-susceptible cells that flip reliably in all our tests. However, we cannot guarantee 100% reliability because the rowhammer effect is still a hardware disturbance error and numerous factors might undermine the reliability, e.g., hardware aging and interference with other running software. The intentional execution is not fully reconstructed in case some junction bit does not flip as expected.

To maximize reliability, we assume the software and hardware vendor do not include any constructs that actively counteract our rowhammer-based program transitions. These include advanced error correction codes for memory transactions, which could correct the bit flips in junction instructions, or anti-tamper protections, which could detect changes in the executing program code.

As many industrial systems are considered critical, our approach might need additional measures to increase reliability. For example, we could leverage multiple distinct junction instructions to enable each functionality change, rather than just one. Whether the changed functionality is enabled at run time then depends on whether the majority of its junction instructions flipped.

In the absence of suitable error correction mechanisms, our method is still applicable to non-critical applications. An alternative approach is to make the intentional and unintentional execution functionally equivalent but make the unintentional form run less optimally. We could achieve this, e.g., by using less optimal algorithms, using more system resources, and requiring more user input. This way, the program will always execute its critical behavior, but it will only execute efficiently on the associated machine.

For systems demanding absolute reliability and with no room for occasional suboptimal code execution, the software vendor may find a different, potentially weaker, protection approach more suitable.

Potential Attacks and Further Mitigations So far, we have assumed that the reverse engineer cannot run analyses directly on a coupled software and hardware pair. This assumption is in line with our threat model and is not unrealistic given that industrial machinery is often locked down using software-based access control mechanisms and physical sealing. However, given that no protection mechanism is perfect, we now discuss which types of attacks the adversary could attempt given full and unrestricted access to an associated device.

One threat that immediately comes to mind is Dynamic Binary Translation (DBT). An attacker could learn about the protected program's behavior by running it in a tool such as Valgrind to dynamically add instrumentation code or to monitor interesting features such as the execution trace, data dependencies, etc. DBT tools preserve neither the input program's binary layout nor the intended mapping between virtual pages and physical memory frames. Therefore, DBT tools would be unable to observe the intended bit flips and thus give the attacker very little useful information about the intentional behavior.

Layout-preserving Dynamic Binary Analysis (DBA) tools or debuggers could resort to hardware features such as performance counters, processor tracing, or hardware breakpoints to analyze an instance of the protected program. These tools can, theoretically, observe transitions between the unintentional and intentional execution modes. However, due to property 5 of rowhammer, detecting these transitions is difficult because there are no obvious, dedicated instructions that explicitly modify the code. At best, the adversary could detect the rowhammering code based on its signature, i.e., a sequence of memory read instructions and a cache line flush of the same address and observe the program's memory afterwards. Prior work, however, already explored numerous other stealthy rowhammer primitives, for example, using non-temporal and implicit loads, uncached memory regions, and cache eviction sets [26,12,1,2,35,37,38]. These techniques remove the visual clues that prompt the adversary to inspect the code pages for bit flips. The attacker can, at best, check for bit flips at regular time intervals. If this interval is big, the attacker risks missing many bit flips, while a small interval is very time-consuming and will frustrate the attacker.

Another problem with DBA tools and debuggers is their performance impact. Due to property 4 of rowhammer, analysis tools cannot meaningfully interfere with the memory controller while the protected program executes its transition code. Even minimal interference could slow down the protected program to a point where the transition fails.

One particularly dangerous threat that *would* be successful against our current prototype is memory snapshotting. Given that our implementation only supports one-time transitioning from the unintentional to intentional execution mode immediately after the program starts, the attacker could simply create a full-memory snapshot when the program reaches its original entry point. This snapshot would show the junction instructions in their intentional form. We could defend against this attack by making the program transition back and forth between its unintentional and intentional execution mode (see Section 7.5).

Attackers could perform their own templating step using the full set of rowhammer parameters (e.g., the aggressors, the hammer pattern, and access counts and speeds) they extracted from the binary. Using the loader map, they could identify the junction instruction that gets loaded into a rowhammer-susceptible cell. We can mitigate this attack by inserting many bogus hammer sequences that either flip unimportant bits or no bits at all. With a sufficiently high number of genuine and bogus hammer sequences, this attack becomes a time-consuming and costly endeavor.

11 Related Work

To the best of our knowledge, there is no directly related work. Thus, we explore the two closest concepts, physically unclonable functions and software diversity.

Physically Unclonable Functions Physically Unclonable Functions (PUFs) are hardware primitives that provide component-specific responses to user-provided challenges [32,13]. PUFs derive their responses from physical hardware characteristics that are extremely difficult to clone and that are expected to be unique because they stem from the many random variations in the hardware's production process. Over the past two decades, researchers have proposed several uses for PUFs, including identification and authentication [32], cryptographic key storage [31,22], and hardware-software binding [19,36]. Using PUF-based encryption of data or code for obfuscation purposes, however, is less stealthy than GlueZilla because all queries to the PUF and the encryption operations are visible and prone to dynamic analysis.

Schaller *et al.* first proposed a rowhammer PUF in 2017 [29]. They validated the PUF's uniqueness and robustness and showed that its entropy is high enough to derive cryptographic keys from. Their design uses a fixed memory area and requires a kernel module for PUF interaction. GlueZilla's design, on the other hand, can operate at any location in physical memory, has no fixed memory size requirement, and does not need a kernel model to safeguard the reserved memory area.

Software Diversity A side effect of our approach is that it results in a form of software diversity [6,21]. Modern software diversity addresses the problem of the software monoculture by generating a unique version of a program for each user. GlueZilla, however, requires specific locations for the junction instruction in memory and, therefore, generates a unique binary layout for each associated machine. Furthermore, the prevalent paradigm in software diversity is to be *hardware agnostic* [21]. GlueZilla is, to the best of our knowledge, the first system breaking with this paradigm.

12 Conclusions

We present GlueZilla, a system that binds software to hardware to prevent industrial-scale reverse engineering. To this end, we combine new compile-time transformations on the software side with DRAM PUFs on the hardware side.

Through GlueZilla, a program now has two different modes of operation: an intentional and an unintentional mode. The intentional mode requires the correct bit flips at the correct locations. The unintentional mode executes the program in its statically encoded representation.

We present the relevant design decisions and implementation details for a fully-fledged GlueZilla prototype. Our evaluation shows that the performance impact of GlueZilla depends primarily on the required number of instructions to modify by bit flips. Although prohibitive performance impact may arise in some configurations, we find that these occur only in compute-intensive programs of the chosen benchmark

suite. Software in our target area of industrial computing systems, however, is typically not compute-intensive. As a result, we expect negligible to moderate performance impacts in the target environment.

Our security evaluation shows that the software bound to hardware by GlueZilla poses significant obstacles to reverse engineering. Reverse engineering based on static analysis is frustrated by software diversity, amplified by not knowing which candidate instructions will be modified by rowhammering. Reverse engineering based on dynamic analysis is frustrated by both not having a hardware environment with identical bit flips and not knowing *when* all of the program's candidate instructions have been modified.

Acknowledgements

The research was supported by the Austrian ministries BMK and BMAW and the State of Upper Austria in the frame of the COMET Module DEPS (FFG 888338) and the SCCH COMET competence center INTEGRATE (FFG 892418).

References

1. Aga, M.T., Aweke, Z.B., Austin, T.: When good protections go bad: Exploiting anti-DoS measures to accelerate rowhammer attacks. In: HOST (2017)
2. Aweke, Z.B., Yitbarek, S.F., Qiao, R., Das, R., Hicks, M., et al.: ANVIL: Software-based protection against next-generation rowhammer attacks. In: ASPLOS (2016)
3. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (im)possibility of obfuscating programs. In: CRYPTO (2001)
4. Bepary, M.K., Talukder, B.M.S.B., Rahman, M.T.: DRAM retention behavior with accelerated aging in commercial chips. Applied Sciences **12**(9) (1 2022)
5. Bosman, E., Razavi, K., Bos, H., Giuffrida, C.: Dedup est machina: Memory deduplication as an advanced exploitation vector. In: S&P 2016 (2016)
6. Cohen, F.B.: Operating system protection through program evolution. Comput. Secur. **12**(6) (10 1993)
7. Cojocar, L., Razavi, K., Giuffrida, C., Bos, H.: Exploiting correcting codes: On the effectiveness of ECC memory against Rowhammer attacks. In: S&P (2019)
8. Frigo, P., Giuffrida, C., Bos, H., Razavi, K.: Grand pwning unit: Accelerating microarchitectural attacks with the GPU. In: S&P (2018)
9. Frigo, P., Vannacc, E., Hassan, H., der Veen, V.v., Mutlu, O., Giuffrida, C., Bos, H., Razavi, K.: TRRespass: Exploiting the many sides of target row refresh. In: S&P (2020)
10. Godbolt, M.: The BTB in contemporary Intel chips — Matt Godbolt's blog (2016), `https://xania.org/201602/bpu-part-three`, [Online; accessed 18 Apr. 2024]
11. 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: S&P (2018)
12. Gruss, D., Maurice, C., Mangard, S.: Rowhammer.js: A remote software-induced fault attack in JavaScript. In: DIMVA (2016)
13. Herder, C., Yu, M.D., Koushanfar, F., Devadas, S.: Physical unclonable functions and applications: A tutorial. Proceedings of the IEEE **102**(8) (2014)
14. Jattke, P., van der Veen, V., Frigo, P., Gunter, S., Razavi, K.: Blacksmith: Scalable rowhammering in the frequency domain. In: S&P (2022)
15. Jattke, P., Wipfli, M., Solt, F., Marazzi, M., Bölcskei, M., Razavi, K.: ZenHammer: Rowhammer attacks on AMD Zen-based platforms. In: USENIX Security (2024)

16. Kim, J.S., Patel, M., Yağlıkçı, A.G., Hassan, H., et al.: Revisiting RowHammer: An experimental analysis of modern DRAM devices and mitigation techniques. In: ISCA (2020)
17. Kim, Y., et al.: Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. ACM SIGARCH Computer Architecture News **42**(3) (2014)
18. Kogler, A., Juffinger, J., Qazi, S., Kim, Y., Lipp, M., Boichat, N., Shiu, E., Nissler, M., Gruss, D.: Half-double: Hammering from the next row over. In: USENIX Security (2022)
19. Kohnhäuser, F., Schaller, A., Katzenbeisser, S.: PUF-based software protection for low-end embedded devices. In: TRUST (2015)
20. Kwong, A., Genkin, D., Gruss, D., Yarom, Y.: RAMBleed: Reading bits in memory without accessing them. In: S&P (2020)
21. Larsen, P., Homescu, A., Brunthaler, S., Franz, M.: SoK: Automated software diversity. In: S&P (2014)
22. Lim, D., Lee, J.W., Gassend, B., Suh, G.E., Van Dijk, M., Devadas, S.: Extracting secret keys from integrated circuits. VLSI **13**(10) (2005)
23. Nagra, J., Collberg, C.: Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Pearson Education (2009)
24. Orosa, L., et al.: A deeper look into RowHammer's sensitivities: Experimental analysis of real DRAM chips and implications on future attacks and defenses. In: MICRO (2021)
25. Piazzalunga, U., Salvaneschi, P., Balducci, F., Jacomuzzi, P., Moroncelli, C.: Security strength measurement for dongle-protected software. Security & Privacy **5**(6) (2007)
26. Qiao, R., Seaborn, M.: A new approach for rowhammer attacks. In: HOST (2016)
27. Razavi, K., Gras, B., Bosman, E., Preneel, B., Giuffrida, C., Bos, H.: Flip feng shui: Hammering a needle in the software stack. In: USENIX Security (2016)
28. de Ridder, F., Frigo, P., Vannacci, E., Bos, H., Giuffrida, C., Razavi, K.: SMASH: Synchronized many-sided Rowhammer attacks from JavaScript. In: USENIX Security (2021)
29. Schaller, A., Xiong, W., Anagnostopoulos, N.A., et al.: Intrinsic rowhammer PUFs: Leveraging the Rowhammer effect for improved security. In: HOST (2017)
30. Seaborn, M., Dullien, T.: Exploiting the DRAM rowhammer bug to gain kernel privileges (2015), https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, [Online; accessed 30 Apr. 2024]
31. Škorić, B., Tuyls, P., Ophey, W.: Robust key extraction from physical uncloneable functions. In: ACNS (2005)
32. Suh, G.E., Devadas, S.: Physical unclonable functions for device authentication and secret key generation. In: DAC (2007)
33. Tehranipoor, F., Karimian, N., Yan, W., Chandy, J.A.: DRAM-based intrinsic physically unclonable functions for system-level security and authentication. VLSI **25**(3) (2017)
34. VDMA: VDMA study product piracy 2022 (2022), https://www.vdma.org/documents/34570/51629660/VDMA+Study+Product+Piracy+2022_final.pdf
35. van der Veen, V., Fratantonio, Y., Lindorfer, M., Gruss, D., Maurice, C., et al.: Drammer: Deterministic Rowhammer attacks on mobile platforms. In: CCS (2016)
36. Xiong, W., Schaller, A., Katzenbeisser, S., Szefer, J.: Software protection using dynamic PUFs. IEEE Transactions on Information Forensics and Security **15** (2019)
37. Zhang, Z., Cheng, Y., Liu, D., Nepal, S., Wang, Z., Yarom, Y.: PThammer: Cross-user-kernel-boundary rowhammer through implicit accesses. In: MICRO (2020)
38. Zhang, Z., et al.: Implicit Hammer: Cross-privilege-boundary rowhammer through implicit accesses. IEEE Transactions on Dependable and Secure Computing (2022)