Statistical Analysis of the Impact of Bit-Flips in Security Critical Code

Tobias Worm Bøgedal[1], René Rydhof Hansen[1][0000−0002−5688−6432], Kim Guldstrand Larsen[1][0000−0002−5953−3384], Axel Legay[2][0000−0003−2287−8925], and Danny Bøgsted Poulsen[1][0000−0001−9623−0748]

[1] Dept. of Computer Science, Aalborg University, Denmark
{tobiaswb,rrh,kgl,dannybpoulsen}@cs.aau.dk
[2] Université Catholique de Louvain, Belgium
axel.legay@uclouvain.be

Abstract. Fault injection is a sophisticated attack in which an attacker may sidestep security of an application by inducing *bit-flips* in the underlying platform. These attacks are typically performed by tampering with the system hardware, but recent RowHammer attacks have shown that bit-flips can be induced predictably and on a large scale through software alone [12]. It is practically impossible for a developer to evaluate and assess if and how much an application is vulnerable to RowHammer attacks. In this paper, we leverage statistical model checking (SMC) to help with these challenges by modelling and analysing potential effects of bit-flips as well as measure the efficacy of proposed mitigation. We illustrate our approach on SUDO, one of several security critical applications recently targeted in the RowHammer-based Mayhem attacks [1].

Keywords: Bit-flips · Fault attacks · Software verification · Model Checking · Formal Methods

1 Introduction

It has long been known that sophisticated attackers can circumvent security in a system by deliberately inducing faults during execution of critical code, e.g., in the access control checks or the cryptographic primitives [5,18]. Such attacks are called *fault injection* attacks or *bit-flip* attacks since the effect of such a fault is typically to "flip" bits in memory or a hardware register.

Generally, fault injection attacks were thought to be primarily a problem for systems where an attacker could gain physical access to tamper with the hardware [4,8]. However, this view changed with the discovery of "RowHammer" attacks [12], showing that large clusters of bit-flips could be induced in a highly predictable way *through software alone*. Recently researchers succeeded in using RowHammer to attack the stack variables and register values of an application. This was showcased by attacking security critical systems software like SUDO and SSH among others, in an attack dubbed "Mayhem" [1].

Targeted fault injection attacks are not only difficult and costly to defend against, it is also very difficult for developers to actually assess if or how vulnerable an application is to such attacks and how effective potential mitigation is. In this paper, we leverage statistical model checking (SMC) to help with these challenges. Using the applications targeted in the Mayhem attack [1], SUDO and SSH, as illustrative examples, we first show how the security critical parts of the code can be modelled; we next use SMC to find possible bit-flip attacks in the code, replicating and validating the findings of the Mayhem attacks, but also resulting in the discovery of a novel bit-flip attack; finally we show ho SMC can be used to evaluate the efficacy of proposed defensive measures, such as those implemented in SUDO [13] to mitigate the effects of the Mayhem attack. In particular, we use SMC to determine if the proposed mitigation offers statistically significant improvement against bit-flip attacks.

We consider the following to be the main contributions of the paper:

- A formal approach to modelling security critical application code;
- The use of symbolic model checking to find bit-flip attacks and/or verify existing bit-flip attacks;
- The use of statistical model checking to determine if the efficacy of proposed mitigation is statistically significant;
- Validation of our approach on the security critical applications SUDO and OpenSSH, recently found vulnerable to RowHammer attacks (the latter only briefly described in this paper). We verify the reported vulnerabilities and report on novel bit-flip attacks against the `strCmp()` function used, e.g., in SUDO.

2 Modelling Code, Bit-flips, and Attackers

Formally, we model a program (written in RISC-V assembly code) as a so-called *control-flow automaton*, in which edges correspond to instructions and locations correspond to the program points immediately before and after an instruction. We have chosen to model programs at the assembly code layer, since bit-flips and their effects are more naturally represented at this layer. The effect of executing instructions, i.e., the semantics of instructions, is modelled as a set of operations $\texttt{Ops}(\mathcal{R})$, parameterised over the set of available registers $\mathcal{R}$, and used to annotate corresponding edges in the control-flow automaton. Edges may also be annotated with boolean guards, $\texttt{G}(\mathcal{R})$, similarly parameterised over the set of available registers. Formally we define

Definition 1. *A control-flow automaton over a set of registers $\mathcal{R}$ is a tuple $\mathcal{C} = \langle \mathcal{L}, \hat{\ell}, \mathcal{E} \rangle$ where*

- *$\mathcal{L}$ is a finite set of control locations,*
- *$\hat{\ell} \in \mathcal{L}$ is the initial location, and*
- *$\mathcal{E} \subseteq (\mathcal{L} \times \texttt{G}(\mathcal{R}) \times \mathcal{L}) \cup (\mathcal{L} \times \texttt{Ops}(\mathcal{R}) \times \mathcal{L})$ is a set of edges, with*
  - *$\texttt{Ops}(\mathcal{R})$ is a set of operations over the registers, and*

```
    xor x1,x1,x1      ; x1 = 0
    slti x2,x1,10     ; x2 = (x1 < 10)
loop:
    beq x2,x0,l8      ; if(x2 == 0) goto l8
    addi x1,x1,1      ; x1 = x1 + 1
    slti x2,x1,10     ; x2 = (x1 < 10)
    j loop            ; goto loop
l8:
    add x3,x0,11      ; x3 = 11
    bne x1,x3,good    ; if (x1 != x3) goto good
error:
    nop               ; error if(x1 == 11)
good:
    nop
```

Listing 1.1. An Example Program

- $\mathtt{G}(\mathcal{R})$ *is a set of boolean expressions over the registers.*

As a shorthand we write $l \xrightarrow{o} l'$ *whenever* $(l, o, l') \in \mathcal{E}$ *and* $o \in \mathtt{Ops}(\mathcal{R})$ *and* $l \underset{g}{\rightarrow} l'$ *whenever* $(l, g, l') \in \mathcal{E}$ *and* $g \in \mathtt{G}(\mathcal{R})$*.*

Registers are assigned values from a domain of bitvectors $\mathcal{B}^n$ of width n, via a mapping $v : \mathcal{R} \rightarrow \mathcal{B}^n$ representing the *current state* of execution in our model. We let V^n be the set of all such mappings. For any operation $o \in \mathtt{Ops}(\mathcal{R})$ we assume there exists a function $\mathcal{M}_o : V^n \rightarrow V^n$ implementing the semantic meaning of that operation, i.e., defines how the state changes during execution in our model. Likewise, we assume it is possible for any element $b \in \mathtt{G}(\mathcal{R})$ and any mapping $v \in V^n s$ to determine if b is satified by v (written $b \vDash v$) or not (written $b \nvDash v$).

Example 1. As an example of a control-flow automaton (CFA), consider the RISC-V assembly code in Listing 1.1. The code is for illustrative purposes only, but is loosely based on a PIN code checker that verifies that a checking loop has been iterated the expected number of times. The CFA representation of this program can be seen in Figure 1. It uses four registers: `x01`, `x1`, `x2`, and `x3`. The initial location is indicated by the black node (numbered 1). Guards over the registers are marked with an orange color. The red node (numbered 12) in the CFA marks the location corresponding to the error label of the program, indicating that the program has reached an error state.

The states of a control-flow automaton $\langle \mathcal{L}, \hat{\ell}, \mathcal{E} \rangle$ over registers $\mathcal{R}$ with domain $\mathcal{B}^n$ are elements $\langle l, v \rangle \in \mathcal{L} \times V^n$ where l is the current control location and v is the current value of the registers. The initial state is the element $\langle \hat{\ell}, v_0 \rangle$ where v_0 assigns a default value to all registers. In many cases, we let this default value be 0, but it can in principle be any value in $\mathcal{B}^n$. In normal operations the CFA may transit from the state $\langle l, v \rangle$ to another state $\langle l', v' \rangle$ (denoted $\langle l, v \rangle \rightarrow \langle l', v' \rangle$)

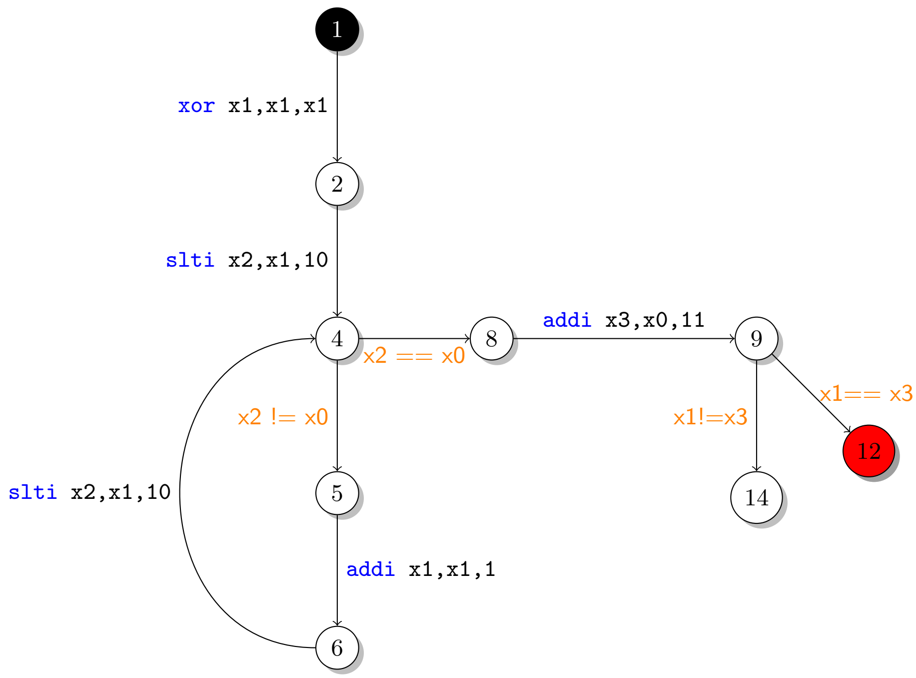

Fig. 1. CFA Representation of code in Listing 1.1

if *(1)* there exists an edge $l \xrightarrow{o} l'$ and $v' = \mathcal{M}_o(v)$ and *(2)* there exists an edge $l \underset{g}{\rightarrow} l'$,$v' = v$ and $v \models g$.

Our definition of CFAs allows the definition of non-deterministic behaviours. However, as we model *closed* programs with no external interaction we find *non-deterministic* executions are unwanted behaviour and will assume CFAs are deterministic meaning that for any state s, if $s \rightarrow s_1$ and $s \rightarrow s_2$ then $s_1 = s_2$.

An execution of a CFA $\langle \mathcal{L}, \hat{\ell}, \mathcal{E} \rangle$ is a (possibly infinite) sequence of states $s_0, s_1, s_2, \ldots$ such that s_0 is the initial state and for all i we have $s_i \rightarrow s_{i+1}$.

Example 2. Consider again the CFA in Figure 1. A (partial) execution of this CFA with standard RISC-V semantics is the sequence

$$\langle \text{❶}, \begin{bmatrix} \mathtt{x0} \mapsto 0 \\ \mathtt{x1} \mapsto 0 \\ \mathtt{x2} \mapsto 0 \\ \mathtt{x3} \mapsto 0 \end{bmatrix} \rangle \langle \text{②}, \begin{bmatrix} \mathtt{x0} \mapsto 0 \\ \mathtt{x1} \mapsto 0 \\ \mathtt{x2} \mapsto 0 \\ \mathtt{x3} \mapsto 0 \end{bmatrix} \rangle \langle \text{④}, \begin{bmatrix} \mathtt{x0} \mapsto 0 \\ \mathtt{x1} \mapsto 0 \\ \mathtt{x2} \mapsto 1 \\ \mathtt{x3} \mapsto 0 \end{bmatrix} \rangle \langle \text{⑤}, \begin{bmatrix} \mathtt{x0} \mapsto 0 \\ \mathtt{x1} \mapsto 0 \\ \mathtt{x2} \mapsto 1 \\ \mathtt{x3} \mapsto 0 \end{bmatrix} \rangle \ldots$$

2.1 Attacker Modelling

Since we wish to analyse the behaviour of programs while under attack, we need an explicit model of the potential actions an attacker may perform. In our work the only actions an attacker can perform are 1) $\mathtt{flip}(r, i)$ to flip the $i^{\mathtt{th}}$ bit of register r, and 2) `skip` indicating the attacker does nothing. For a set of registers $\mathcal{R}$ we let $\mathtt{Act}(\mathcal{R}) = \{\mathtt{flip}(r,i) | r \in \mathcal{R}\} \cup \{\mathtt{skip}\}$. If the attacker performs multiple flips in rapid succession then they can, essentially, overwrite a register with whatever value they wish. A CFA in state $\langle \ell, v \rangle$ will under a $\mathtt{flip}(r,i)$ action forcefully transit to state $\langle \ell, v' \rangle$ where $v[r \mapsto (v(r) \bigotimes s(2^i))]\rangle$ and $\bigotimes$ is the bitwise exclusive or operation. We denote this transition by $\langle \ell, v \rangle \overset{\mathtt{flip}(r,i)}{\dashrightarrow} \langle \ell, v' \rangle$. If we have $A \subseteq \mathtt{Act}(\mathcal{R})$ then we allow writing $\langle \ell, v \rangle \overset{A}{\dashrightarrow} s$ to find state s which is the result of applying each action in A successively.We can now model an attacker against a CFA $\mathcal{C} = \langle \mathcal{L}, \hat{\ell}, \mathcal{E} \rangle$ over registers $\mathcal{R}$ as a transition system $\langle \mathcal{S}^{\mathcal{A}}, \hat{s}^{\mathcal{A}}, \mathcal{P}, \dashrightarrow \rangle$ where

- $\mathcal{S}^{\mathcal{A}}$ is a set of states,
- $\hat{s}^{\mathcal{A}} \in \mathcal{S}^{\mathcal{A}}$ is the initial attacker state,
- $\mathcal{P}$ is a set of propositions giving the attacker (limited) information about the state of the CFA, and
- $\dashrightarrow \subseteq \mathcal{S}^{\mathcal{A}} \times 2^{\mathcal{P}} \times 2^{\mathtt{Act}(\mathcal{R})} \times \mathcal{S}^{\mathcal{A}}$ is a set of transitions labelled with actions the attacker performs and guarded by a set of propositions that must be true while transitting. As a shorthand we allow writing $s_0 \overset{p,a}{\dashrightarrow} s_1$ whenever $(s_0, p, a, s_1) \in \dashrightarrow$.

We assume the existence of a function $\mathcal{K} : \mathcal{S} \to 2^{\mathcal{P}}$ mapping CFA states $\mathcal{S}$ to observable propositons. Having all these parts in place we can now define transition rules for how an attacker and the CFA interact:

$$\frac{s \to s_1}{\mathcal{K} \vdash (s, s^{\mathcal{A}}) \to (s_1, s^{\mathcal{A}})} \qquad \frac{s \overset{a}{\dashrightarrow} s_1 \qquad s^{\mathcal{A}} \overset{p,o}{\dashrightarrow} s^{\mathcal{A}}{}_1 \qquad p \subseteq \mathcal{K}(s)}{\mathcal{K} \vdash (s, s^{\mathcal{A}}) \overset{a}{\dashrightarrow} (s_1, s^{\mathcal{A}}{}_1)}$$

Example 3. Consider again the CFA in Figure 1 this time running in parallel with an attacker that is allowed to flip one bit in any register but only once: $\langle \{s^{\mathcal{A}}{}_1, s^{\mathcal{A}}{}_2\}, \emptyset, \dashrightarrow \rangle$ with $\dashrightarrow = \{(s^{\mathcal{A}}{}_1, \emptyset, r s^{\mathcal{A}}{}_2) | r \in \mathtt{Act}(\mathcal{R})\}$. Given this fairly restricted attacker we might be wondering whether it is possible to get the program to malfunction and end in line 12. To answer this we simply explore the joint state space in search of a state where the CFA is in location $\textcircled{12}$. An exploration of this kind will reveal the following execution

$$(\langle ❶, \begin{bmatrix} \mathtt{x0} \mapsto 0 \\ \mathtt{x1} \mapsto 0 \\ \mathtt{x2} \mapsto 0 \\ \mathtt{x3} \mapsto 0 \end{bmatrix} \rangle, s^{\mathcal{A}}{}_1)(\langle \textcircled{2}, \begin{bmatrix} \mathtt{x0} \mapsto 0 \\ \mathtt{x1} \mapsto 0 \\ \mathtt{x2} \mapsto 0 \\ \mathtt{x3} \mapsto 0 \end{bmatrix} \rangle, s^{\mathcal{A}}{}_1)(\langle \textcircled{4}, \begin{bmatrix} \mathtt{x0} \mapsto 0 \\ \mathtt{x1} \mapsto 0 \\ \mathtt{x2} \mapsto 1 \\ \mathtt{x3} \mapsto 0 \end{bmatrix} \rangle, s^{\mathcal{A}}{}_1)$$

$$(\langle \textcircled{5}, \begin{bmatrix} \mathtt{x0} \mapsto 0 \\ \mathtt{x1} \mapsto 0 \\ \mathtt{x2} \mapsto 1 \\ \mathtt{x3} \mapsto 0 \end{bmatrix} \rangle, s^{\mathcal{A}}{}_1)(\langle \textcircled{6}, \begin{bmatrix} \mathtt{x0} \mapsto 0 \\ \mathtt{x1} \mapsto 1 \\ \mathtt{x2} \mapsto 1 \\ \mathtt{x3} \mapsto 0 \end{bmatrix} \rangle, s^{\mathcal{A}}{}_1)(\langle \textcircled{4}, \begin{bmatrix} \mathtt{x0} \mapsto 0 \\ \mathtt{x1} \mapsto 1 \\ \mathtt{x2} \mapsto 1 \\ \mathtt{x3} \mapsto 0 \end{bmatrix} \rangle, s^{\mathcal{A}}{}_1) \ldots$$

$$(\langle \textcircled{4}, \begin{bmatrix} \mathtt{x0} \mapsto 0 \\ \mathtt{x1} \mapsto 10 \\ \mathtt{x2} \mapsto 0 \\ \mathtt{x3} \mapsto 0 \end{bmatrix} \rangle, s^{\mathcal{A}}{}_1) \lightning (\langle \textcircled{4}, \begin{bmatrix} \mathtt{x0} \mapsto 0 \\ \mathtt{x1} \mapsto 10 \\ \mathtt{x2} \mapsto 1 \\ \mathtt{x3} \mapsto 0 \end{bmatrix} \rangle, s^{\mathcal{A}}{}_2)(\langle \textcircled{5}, \begin{bmatrix} \mathtt{x0} \mapsto 0 \\ \mathtt{x1} \mapsto 10 \\ \mathtt{x2} \mapsto 1 \\ \mathtt{x3} \mapsto 0 \end{bmatrix} \rangle, s^{\mathcal{A}}{}_2)$$

$$(\langle \textcircled{6}, \begin{bmatrix} \mathtt{x0} \mapsto 0 \\ \mathtt{x1} \mapsto 11 \\ \mathtt{x2} \mapsto 1 \\ \mathtt{x3} \mapsto 0 \end{bmatrix} \rangle, s^{\mathcal{A}}{}_2)(\langle \textcircled{4}, \begin{bmatrix} \mathtt{x0} \mapsto 0 \\ \mathtt{x1} \mapsto 11 \\ \mathtt{x2} \mapsto 0 \\ \mathtt{x3} \mapsto 0 \end{bmatrix} \rangle, s^{\mathcal{A}}{}_2)(\langle \textcircled{8}, \begin{bmatrix} \mathtt{x0} \mapsto 0 \\ \mathtt{x1} \mapsto 11 \\ \mathtt{x2} \mapsto 0 \\ \mathtt{x3} \mapsto 0 \end{bmatrix} \rangle, s^{\mathcal{A}}{}_2)$$

$$(\langle \textcircled{9}, \begin{bmatrix} \mathtt{x0} \mapsto 0 \\ \mathtt{x1} \mapsto 11 \\ \mathtt{x2} \mapsto 0 \\ \mathtt{x3} \mapsto 11 \end{bmatrix} \rangle, s^{\mathcal{A}}{}_2)(\langle \textcircled{12}, \begin{bmatrix} \mathtt{x0} \mapsto 0 \\ \mathtt{x1} \mapsto 11 \\ \mathtt{x2} \mapsto 0 \\ \mathtt{x3} \mapsto 11 \end{bmatrix} \rangle, s^{\mathcal{A}}{}_2)$$

The execution shows the attacker was successful in forcing the CFA into an error state by "flipping" the result of the test in line 6, falsely indicating that the content of register `x2` is still less than 10. In the execution we have indicated the place that the flip occured by a $\lightning$ symbol.

For the remaining parts of this paper we insist attackers are *action-deterministic* meaning that for any state $s^{\mathcal{A}}$, if $s^{\mathcal{A}} \overset{a}{\dashrightarrow} {s^{\mathcal{A}}}_1$ and $s^{\mathcal{A}} \overset{a}{\dashrightarrow} {s^{\mathcal{A}}}_2$ then ${s^{\mathcal{A}}}_1 = {s^{\mathcal{A}}}_2$. This assumption is mainly needed to make the definition of a probabilistic semantics easier in the following section. In case we need to model an attacker with $s^{\mathcal{A}} \overset{a}{\dashrightarrow} {s^{\mathcal{A}}}_1$ and $s^{\mathcal{A}} \overset{a}{\dashrightarrow} {s^{\mathcal{A}}}_2$ and ${s^{\mathcal{A}}}_1 \neq {s^{\mathcal{A}}}_2$ then we can accomodate this by just creating an extra action a' with semantics as a and let $s^{\mathcal{A}} \overset{a'}{\dashrightarrow} {s^{\mathcal{A}}}_2$.

2.2 Probabilistic Attacker

In real-life bit-flip attacks, the attacker typically cannot control the specific register or bit that flips particularly well. Insted they use techniques that has a certain probability of flipping bit n of register r. We can model such techniques by adding to an attacker $(\mathcal{S}^{\mathcal{A}}, \hat{s}^{\mathcal{A}}, \mathcal{P}, \dashrightarrow)$ a function $\pi : \mathcal{S}^{\mathcal{A}} \times 2^{\mathcal{P}} \to \mathtt{Act}(\mathcal{R}) \to [0, 1]$ that for each state-observation pair assigns probabilities for the potential attacker actions. For this function to be well-behaved it must: 1. only assign probabilities to actions that are actually possible; and 2. $\pi(s, p)$ must in fact be a probability mass function. More formally, for any $s^{\mathcal{A}} \in \mathcal{S}^{\mathcal{A}}$ and any $p \in 2^{\mathcal{P}}$:

- $\pi(s^{\mathcal{A}}, p)(a) \neq 0$ implies $s \overset{p,a}{\dashrightarrow} s_1$ for some s_1,
- $\left(\sum_{a \in \mathtt{Act}(\mathcal{R})} \pi(s^{\mathcal{A}}, p)(a)\right) = 1$

We can now describe the probability that an attacker performs a specific finite sequence of actions $\sigma = a_1, a_2, a_3, \ldots, a_n$ from state $(s, s^{\mathcal{A}})$ recursively as:

$$\mathbb{P}_{(s,s^{\mathcal{A}})}(\sigma) = \pi(s^{\mathcal{A}}, \mathcal{K}(s))(a_1) \cdot \mathbb{P}_{(s',s^{\mathcal{A}\prime})}(a_2, a_3 \ldots, a_n),$$

where $(s, s^{\mathcal{A}}) \overset{a}{\dashrightarrow}\!\!\to (s', s^{\mathcal{A}\prime})$.

Consider now that we are given two different implementations, $\mathcal{C}_1$ and $\mathcal{C}_2$ of the same feature and we want to answer the question

Is $\mathcal{C}_1$ more secure than $\mathcal{C}_2$?

Given the stochastic nature of the attacker, we believe the most natural way to answer this question, is to determine if the probability of a successful attack on $\mathcal{C}_1$ is less than the probability of a successful attack on $\mathcal{C}_2$. This question could be answered by use of probablistic model checking or with statistical reasoning e.g., by performing a t-test (see Section 5 for details).

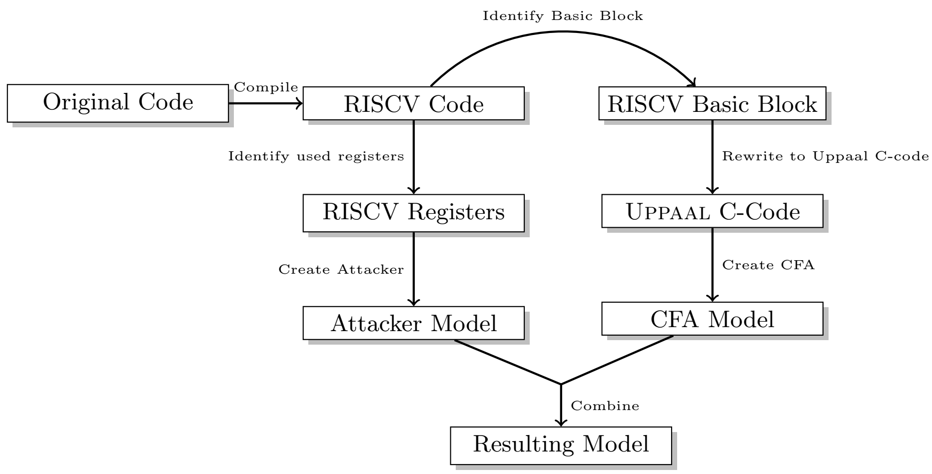

Fig. 2. The process used for creating the models.

3 Modelling SUDO and Attackers

In the remainder of this paper, we detail our approach by analysing the critical parts of the SUDO application using the Uppaal model checker[3] for modelling and analysis.

The SUDO command is an integral part of UNIX-like operating systems. It allows authorised users to execute commands as another user and is typically used to execute a few commands with superuser (or administrator) privileges. This is both more convenient and more secure than having to log in as the superuser and then execute the commands. However, this also makes SUDO a security critical component, since a malicious user able to break or circumvent the SUDO authorisation can perform any privileged actions on the system and thus fully compromise it. The Mayhem attack showed how SUDO authorisation could be compromised through targeted bit-flips induced by RowHammer attacks [1].

As mentioned in the introduction, we model code at the assembly code layer where the effects of bit-flips are more naturally represented. Consequently, we must first compile the C source of SUDO to RISC-V assembly code. Since we are only interested in the security critical part, i.e., user authorisation, we first extract that into a separate stand-alone C file. As a part of this, and to avoid modelling function calls and returns, we also inline library functions, in this case only the `strCmp()` function is included. The resulting, simplified C source can be seen in Listing 1.2, reduced essentially to the source of the `sudo_passwd_verify()` function and its dependencies. The simplified C code is then compiled to assem-

[3] `https://uppaal.org/`

```c
#define AUTH_SUCCESS    0x52a2925     /* 0101001010100010100100100101 */
#define AUTH_FAILURE    0xad5d6da     /* 1010110101011101011011011010 */

typedef struct sudo_auth {
    unsigned int flags;   /* various flags, see below */
    int status;           /* status from verify routine */
    const char *name;     /* name of the method as a string */
    void *data;           /* method-specific data pointer */
} sudo_auth;

int sudo_passwd_verify(const char *pass, sudo_auth *auth)
{
    char *pw_passwd = auth->data;
    int ret;

    int strCmpRes = 4095;
    while (*pass == *pw_passwd++)
        if (*pass++ == '\0'){
            strCmpRes = 0;
            break;
        }
    if (strCmpRes != 0)
        strCmpRes = (*(const unsigned char *)pass -
                        *(const unsigned char *)(pw_passwd - 1));

    if (strCmpRes == 0)
        ret = AUTH_SUCCESS;
    else
        ret = AUTH_FAILURE;

    return ret;
}
```

Listing 1.2. Modified `sudo_passwd_verify()`

bly code, in this case using the Compiler Explorer[4] for convenience, and divided into the basic blocks comprising the control flow of `sudo_passwd_verify()`. The RISC-V code can be seen in Appendix A. The machine registers used are identified to be used in the attacker specification. The entire process for creating the model can be seen in Figure 2. As of writing it is a manual process, but big parts of it could be automated. We will not go into further detail with the conversion from RISC-V assembly here, merely note that the same process has been used on OpenSSH, another target of the Mayhem attack, briefly described in Section 6.

[4] https://godbolt.org/

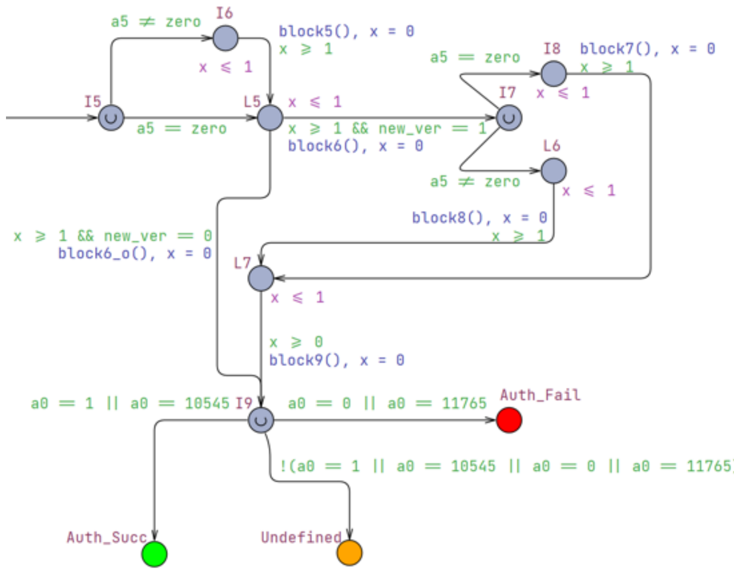

Fig. 3. Part of the model for the `sudo_passwd_verify()` function

3.1 Modelling `sudo_passwd_verify()` in Uppaal

From the assembly code, obtained as described above, we follow the process described in Section 2 and model the control-flow automaton corresponding to the code of the `sudo_passwd_verify()` function as a timed automaton in Uppaal. In this model basic blocks are assigned to nodes and transitions correspond to executing a basic block implemented in the C like modelling language available in Uppaal. An excerpt of the model is shown in Figure 3.[5]

Using timed automata to model the code (and the system) opens a wide range of options for specifying when bit-flip attacks can occur, e.g., in terms of how long time or how many CPU cycles have been executed. In this work, we have chosen a timing model that allows an attacker to perform a bit-flip between the execution of basic blocks. We implement this approach using a local clock `x` to explicitly track and restrict time spent in basic blocks through invariants $x \leq 1$ and guards $x \geq 1$.

One notable exception to the use of basic blocks is the block named ***block3()*** which is broken into two parts, named ***block3_a()*** and ***block3_b()*** respectively, with a committed location between them to still have the two parts performed atomically. This is done because we need to check if array indexing during the execution of ***block3()*** will be out of bounds as this will lead to Uppaal throwing an error and halting verification. Therefore, if the value of register `a5` is larger than the variable named `size` (which is the length of the passwords) after

[5] The full model is available from `https://github.com/dannybpoulsen/at_refutation_models`

executing `block3_a()` then the model goes into the deadlocked location labeled `MemSegFault`, representing a memory fault.

Modelling code and bit-flips at the basic block level simplifies both modelling and analysis but also means some granularity is lost when looking for potential attacks. Our approach can, however, easily be adapted to modelling individual instructions, or even the underlying micro-code. Furthermore, experiments show that interesting and relevant results can already be obtained with the current coarse model, cf. Table 1.

To simplify modelling of branching in the code, through branching instructions such as `bne`, `beq`, and so on, we model these instructions, i.e., the final instruction of a basic block, as a separate *urgent* location, e.g., location `I7` in Figure 3. The reason for using urgent locations is that the branching instructions are still part of the basic block executed during the ingoing transitions. We therefore want these instructions to execute without the delay otherwise imposed on basic blocks, but not atomically since bit-flips during these comparisons and the subsequent branching could be of interest.

An urgent location is also used after the execution of all basic blocks to check the return value of `sudo_passwd_verify()`: The model enters one of three *deadlock* states depending on whether the authentication is successful, resulting in the `Auth_Succ` location (marked green), or whether the authentication has failed resulting in the `Auth_Fail` location (marked red), or finally whether a bit-flip has caused the return value to be unrecognisable, resulting in the `Undefined` location (marked orange). The latter is also considered a failed authentication.

We model the memory relevant to the `sudo_passwd_verify()` function as a local struct. This is done because the attacker does not need access to the memory used by the function since we only look for bit-flips in registers. This of course means that the registers are globally available in the model so that the attacker can access them.

In preparation for the intended use of statistical model checking, we encode two versions of the `sudo_passwd_verify()` code in the same model: the original version and the patched version that aims to protect against RowHammer attacks [13]. The encoding works by using two *committed* locations, non-deterministically choosing between the two versions. This is done by having the transitions from the location `L5` be decided by a global variable called `new_ver`, determining which version to use. This works because the two versions of the function only differ after reaching the location labeled `L5` in the model. If we want to control whether we use the old or new version explicitly, we can make the second location be the initial location and manually choose a value for `new_ver`. This can be seen in Figure 3 where the two guards on the transitions from `L5` depend on the value of `new_ver`.

A final thing to note, is that the SUDO password is hardcoded. An alternative would be to let the model generate a password at random at the start of a trace. However, even with a small set of values to choose from the state space of the entire model (`sudo_passwd_verify()` and the attacker) grows to the point where symbolic model checking becomes infeasible. Furthermore, it is often preferable

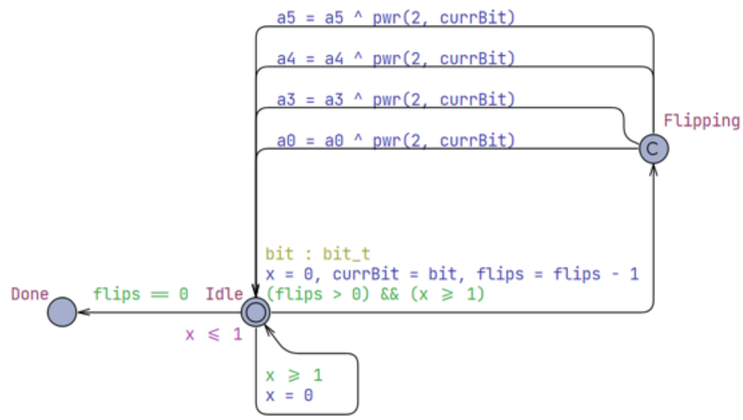

Fig. 4. Model of the attacker for SUDO

to be able to control the password explicitly to ensure that corner cases are taken into account, e.g., particularly weak or strong passwords.

3.2 The Attacker Model

The model for the attacker can be seen in Figure 4. The modelling is inspired in part by [9]. The attacker starts in the `Idle` location and can choose to perform a bit-flip a number of times equal to the global variable `MAX_FLIPS` per trace. This is implemented by setting the local variable `flips = MAX_FLIPS` at the instantiation of the model and when `flips == 0` the attacker goes into the deadlocked location labelled `Done`.

As mentioned in the previous section, the attacker can only perform bit-flips between the execution of basic blocks, as tracked by the clock `x`, i.e., when $x \geq 1$.

A bit-flip is performed by the attacker by first non-deterministically choosing a value for the variable called `bit` and setting `currBit = bit`. The value of `bit` can be between zero and eight. Afterwords the attacker moves to the committed location labelled `Flipping`. From here it chooses one of four transitions each corresponding to a different register. For the model we have chosen these registers by looking at the RISC-V code for `sudo_passwd_verify()` and seeing which registers are used throughout. Once the attacker chooses one of the transitions the value of the corresponding register is bitwise XOR'ed with $2^{\texttt{currBit}}$ resulting in a simulated bit-flip. Afterwards the attacker returns to the `Idle` location.

The attacker model is run in conjunction with the model of the code for `sudo_passwd_verify()`.

4 Verification

To evaluate the model under different scenarios we set up variables for the stored password and the password input by the user. We also limit the number of times the attacker can bit-flip during a single trace to have more realistic scenarios.

First we look at whether or not the added countermeasures in the new version of `sudo_passwd_verify()` protects against the bit-flip attacks found in the Mayhem attack [1]. This is done by using a configuration where the arrays `user_pass` and `stored_pass` are set to different values and the attacker can bit-flip once per trace. We then run the following query on the model:

```
A<> (SUDO.Auth_Fail || SUDO.MemSegFault || SUDO.Undefined)
```

This query should hold if the function is *not* susceptible to bit-flip attacks bypassing authentication given *non-matching* passwords. We run this query on both the new and old version of the system by changing the `new_ver` variable. As expected the query does *not* hold for the old version of the system. This is indeed due to the fact that an attacker can flip the value of the `matched` variable after `strCmp()` has been executed (cf. Listing 1.2). Furthermore, the query only holds for some passwords in the new version. If the passwords have a matching first character, the query still does not hold (cf. Section 4.1). This means the changes to the code do increase protection against bit-flip attacks but do not eliminate them.

While working on verifying the efficacy of the countermeasures placed in the new version we also found other possible bit-flip attacks around `strCmp()`. These were found by changing the passwords and the number of allowed bit-flips in the model. The attacks can be seen by going through the trace for the counter-examples that UPPAAL gives when a query does not hold.

4.1 Attack A: Comparison Shortcut

In the first bit-flip attack, the attacker only has to guess the first character of a user's password to bypass SUDO authentication. The way this attack works is that, if the attacker correctly guesses the first character of the password, then they can bit-flip one of the registers (`a4` or `a5`) used to compare the values for equality during `strCmp()` (cf. Listing 1.2). If the first character of the input matches the password, then the values of `a4` and `a5`, in that iteration of the loop, are equal. This should lead to comparing the characters at the next index of each string. However, if the attacker flips the value of `a4` or `a5` to any other value the comparison yields an inequality (in the model, the location in which the the bit-flip takes place is `I2`). This triggers the else branch of `strCmp()` which normally subtracts the two different character values from each other. However, since the values are actually equal this subtraction yields zero, which makes `strCmp()` return that the strings are equal.

4.2 Attack B: Index Skip

We also found further attacks when the attacker is allowed to perform bit-flips more than once during a trace. These attacks are mainly elaborate versions of the `strCmp()` attack described in the previous section (Comparison Shortcut). As an example, if the correct password is `1234` and the attacker tries to input the password `2234`, then it is possible to bit-flip the register used for pointers to the array indexes early in the trace so that the first comparison done by `strCmp()` is on the second character of the correct password and either the first or second character of the input. From this point the attacker can perform the Comparison Shortcut attack.

4.3 Attack C: Result Flip

The last attack we found in the model is when `strCmp()` checks two values for equality when they are different, but close to each other. When this happens, `strCmp()` subtracts the values and returns the result. If the result is sufficiently small, then it is feasible to perform a bit-flip and change the result to zero when the model is in the location labeled `I7`. This causes `ret` to be set to the value `AUTH_SUCCESS` and bypasses authentication.

5 Statistical Analysis

Given that the new version of SUDO tries to address the bit-flip attack described in [1] and that we can still identify possible attacks, an interesting question is how much more secure is the new version compared to the old one. In our view the "more secure" means attacks are less likely to happen on the improved version than it is on the original version (given a specific stochastic attacker). To reason about probabilities we use Uppaal SMC [2], which is a statistical model checking engine in Uppaal. Uppaal SMC performs simulations on a given model instead of symbolic model checking. This leads to the probabilities of properties holding rather than hard guarantees.

We use Uppaal SMC to estimate the probability that our attacker succesfully attacks the system (i.e., reaches `Auth_Succ`) on different configurations of our model. A configuration is in this regard a different initialisation of `new_ver`, `MAX_FLIPS`, `user_pass` and `stored_pass`. For each configuration we ran the query

```
Pr[<=500;1000000] (<> SUDO.Auth_Succ).
```

It queries for the probability that the attacker puts the SUDO template in the `Auth_Succ` location within at most 500 time units. Uppaal estimates the probability with 1 000 000 samples.

In Table 1 we show the results of these queries along with the hamming distance between `user_pass` and `stored_pass`. To validate whether there is a significant difference between the old version (`new_ver = 0`) and the bit-flip

MAX_FLIPS	user_pass	stored_pass	Hamming	new_ver == 0	new_ver == 1	PValue
1	1245	2245	2	0	0	NAN
2	1245	2245	2	24	23	8.84E-01
3	1245	2245	2	326	332	8.15E-01
4	1245	2245	2	1995	1951	4.83E-01
5	1245	2245	2	5767	5605	1.28E-01
1	1245	6789	9	0	0	NAN
2	1245	6789	9	28	0	1.21E-07
3	1245	6789	9	372	0	6.63E-83
4	1245	6789	9	2061	0	0.00E+00
5	1245	6789	9	5750	0	0.00E+00
1	1245	2289	6	0	0	NAN
2	1245	2289	6	15	18	6.02E-01
3	1245	2289	6	317	354	1.53E-01
4	1245	2289	6	2032	2001	6.25E-01
5	1245	2289	6	5705	5551	1.45E-01
1	1245	1367	3	11169	11217	7.47E-01
2	1245	1367	3	99691	100011	4.50E-01
3	1245	1367	3	99208	99723	2.24E-01
4	1245	1367	3	99081	97397	6.31E-05
5	1245	1367	3	95262	94687	1.65E-01

Table 1. Successful attacks in the new version and old version for different configurations

hardened version (`new_ver = 1`) we furthermore performed a Welchs t-test [17]. In Table 1 the PValue column shows the p-value of this test. A Welchs t-test is a *classic* statistical hypothesis test to test if the means of two distributions are equal; and the p-value is the probability of getting a result as extreme as the observed given the two means are equal.

The results indicate that allowing the attacker more flips increases the chance of successfully attacking both versions in most cases. However it should be noted that the amount of successful bit-flip attacks does not strictly increase in the last password configuration in Table 1. This could be due to the fact that, for some traces, further bit-flips beyond a threshold value might flip non-useful values or overwrite values achieved with a "correct" bit-flip. This could be interesting to explore further, but we leave it for future work. The results also indicate that there is a significant difference between the old system and the hardened system when `user_pass` and `stored_pass` are sufficiently far away from each other. On the other hand, if the passwords are "close" to each other then there are no observed difference between the two versions. The reason for this is that the majority of the attacks (with Hamming column < 3) is of type Attack A (Comparison Shortcut) and Attack B (Index Skip) and these are possible in both versions. As the hamming distance between `user_pass` and `stored_pass` gets larger these attack become less likely and the feasible attacks in the old version are protected against in the hardened version.

6 OpenSSH

In addition to SUDO, the Mayhem attack was also applied to the `auth_password()` function in OpenSSH, which plays a similar role to that of `sudo_passwd_verify()` in SUDO, showing that it too is vulnerable to RowHammer attacks. OpenSSH is the widely used open source implementation of the SSH protocol for secure communication and connection. To demonstrate that our method of modelling bit-flip attacks can be generalized we have also modelled `auth_password()` using the same approach as described in Section 3 except that it was not necessary to inline any functions. Our model confirms the attacks found in the Mayhem attacks. We have furthermore created and modelled a new version of `auth_password()` that is not susceptible to these attacks. We will not go into further detail with OpenSSH here, merely refer to the code along with the models which can be found on GitHub[6].

7 Related Work

Since their discovery, RowHammer attacks and defences against them have been investigated and explored in-depth [12]. Much of the recent work has focused on either on making the attacks better, e.g., more precise or less resource intensive [20,14,11], or on possible mitigation against the attacks [3,10,16]. See [15] for an overview of recent research. However, most of the proposed defences mentioned depend entirely on either novel hardware designed specifically to handle RowHammer or on hardware that has been modified to make RowHammer mitigation easier or even possible.

There is little work on applying formal methods, and statistical model checking in particular, to model and analyse applications for bit-flip vulnerabilities. Some works [6,19,7] simulate hardware failures by modifying the binary code and demonstrate the program no longer produces the expected result. However, this is not validated by model checking and it does not consider runtime-induced bit-flips.

8 Conclusion

In this paper we have shown how statistical model checking can be used to model, analyse, and evaluate the effects of RowHammer-like bit-flip attacks on security critical code. We believe that this work also shows the great potential of formal methods in general, and statistical model checking in particular, for helping developers assessing whether an application is vulnerable to bit-flip attacks and

[6] `https://github.com/dannybpoulsen/at_refutation_models`

also in choosing and assessing potential countermeasures against these attacks, without requiring or investing in special hardware. Since the proposed method analyzes the code at the level of basic blocks it requires less time to get a model up and running. This, of course, means that individual instructions are not checked for potential bit-flip attacks. However, as shown in this paper, it is still possible to get valuable results. As with most model checking problems it is important to discern which parts of the system are critical and focus on modeling these. It is not feasible to model an entire system as the problem would explode. Furthermore, in the current models the attacker may choose to flip between the execution of any basic block up to a specified number of total bit-flips. This means that the attacker might run out of allowed bit-flips before getting to vulnerable parts of the code. In larger models it might therefore be necessary to "guide" the attacker by allowing and disallowing bit-flips in certain sections of the model. However, this has not been necessary for our cases.

Acknowledgments. This work has been partially supported by both Innovation Fund Denmark and the Digital Research Centre Denmark (DIREC) through the bridge project *Secure Internet of Things* (SIoT); and also through the VILLUM Investigator grant S4OS (Scalable analysis and Synthesis of Safe, Secure and Optimal Strategies for Cyber-Physical Systems).

References

1. Adiletta, A.J., Tol, M.C., Doröz, Y., Sunar, B.: Mayhem: Targeted corruption of register and stack variables. CoRR **abs/2309.02545** (2023), https://doi.org/10.48550/arXiv.2309.02545, to be presented at ASIACCS 2024.
2. David, A., Larsen, K.G., Legay, A., Mikucionis, M., Poulsen, D.B.: Uppaal SMC tutorial. Int. J. Softw. Tools Technol. Transf. **17**(4), 397–415 (2015). https://doi.org/10.1007/S10009-014-0361-Y
3. Dio, A.D., Koning, K., Bos, H., Giuffrida, C.: Copy-on-Flip: Hardening ECC memory against Rowhammer attacks. In: Proceedings of the 30th Annual Network and Distributed System Security Symposium (NDSS 2023) (2023), https://www.ndss-symposium.org/ndss-paper/copy-on-flip-hardening-ecc-memory-against-rowhammer-attacks/
4. Gadellaa, K.O.: Fault Attacks on Java Card. Master's thesis, Technical University of Eindhoven (Aug 2005)
5. Ghosh, A.K., O'Connor, T., McGraw, G.: An automated approach for identifying potential vulnerabilities in software. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P 1998). pp. 104–114 (May 1998), https://www.cigital.com/papers/download/ieees_p98_2col.pdf
6. Given-Wilson, T., Jafri, N., Legay, A.: Combined software and hardware fault injection vulnerability detection. Innov. Syst. Softw. Eng. **16**(2), 101–120 (2020). https://doi.org/10.1007/S11334-020-00364-5, https://doi.org/10.1007/s11334-020-00364-5
7. Given-Wilson, T., Legay, A.: Formalising fault injection and countermeasures. In: Volkamer, M., Wressnegger, C. (eds.) ARES 2020: The 15th International Conference on Availability, Reliability and Security, Virtual Event, Ireland, August 25-28,

2020. pp. 22:1–22:11. ACM (2020). https://doi.org/10.1145/3407023.3407049, https://doi.org/10.1145/3407023.3407049
8. Govindavajhala, S., Appel, A.W.: Using memory errors to attack a virtual machine. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P 2003) (2003), http://www.cs.princeton.edu/~sudhakar/papers/memerr.pdf
9. Hansen, R.R., Larsen, K.G., Olesen, M.C., Wognsen, E.R.: Formal modelling and analysis of Bitflips in ARM assembly code. Information Systems Frontiers **18**(5), 909–925 (October 2016). https://doi.org/10.1007/s10796-016-9665-7
10. Juffinger, J., Lamster, L., Kogler, A., Eichlseder, M., Lipp, M., Gruss, D.: Csi:rowhammer - cryptographic security and integrity against rowhammer. In: Proceedings of the 44th IEEE Symposium on Security and Privacy (SP 2023). pp. 1702–1718 (2023). https://doi.org/10.1109/SP46215.2023.10179390
11. Kaur, A., Srivastav, P., Ghoshal, B.: Flipping bits like a pro: Precise Rowhammering on embedded devices. IEEE Embed. Syst. Lett. **15**(4), 218–221 (2023). https://doi.org/10.1109/LES.2023.3298737
12. Kim, Y., Daly, R., Kim, J.S., Fallin, C., Lee, J., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In: Proceedings of the 41st ACM/IEEE International Symposium on Computer Architecture (ISCA 2014). pp. 361–372 (2014). https://doi.org/10.1109/ISCA.2014.6853210
13. Miller, T.C.: Try to make sudo less vulnerable to ROWHAMMER attacks. Commit number 7873f8334c8d31031f8cfa83bd97ac6029309e4f for the sudo project on GitHub. (Sep 2023), https://github.com/sudo-project/sudo/commit/7873f8334c8d31031f8cfa83bd97ac6029309e4f
14. Mondal, P., Kundu, S., Bhattacharya, S., Karmakar, A., Verbauwhede, I.: A practical key-recovery attack on lwe-based key-encapsulation mechanism schemes using rowhammer. In: Applied Cryptography and Network Security - 22nd International Conference, ACNS 2024, Abu Dhabi, United Arab Emirates, March 5-8, 2024, Proceedings, Part III. pp. 271–300 (2024). https://doi.org/10.1007/978-3-031-54776-8_11
15. Mutlu, O., Olgun, A., Yaglikçi, A.G.: Fundamentally understanding and solving RowHammer. In: Proceedings of the 28th Asia and South Pacific Design Automation Conference (ASPDAC 2023). pp. 461–468 (2023). https://doi.org/10.1145/3566097.3568350
16. Saxena, A., Saileshwar, G., Juffinger, J., Kogler, A., Gruss, D., Qureshi, M.K.: PT-Guard: Integrity-protected page tables to defend against breakthrough Rowhammer attacks. In: Proceedings of the 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Network (DSN 2023). pp. 95–108 (2023). https://doi.org/10.1109/DSN58367.2023.00022
17. Welch, B.: The generalization of 'student's' problem when several different population varlances are involved. Biometrika (1947)
18. Xu, J., Chen, S., Kalbarczyk, Z., Iyer, R.K.: An experimental study of security vulnerabilities caused by errors. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN 2001). pp. 421–430 (Jul 2001). https://doi.org/10.1109/DSN.2001.941426
19. Zavalyshyn, I., Given-Wilson, T., Legay, A., Sadre, R.: Brief announcement: Effectiveness of code hardening for fault-tolerant iot software. In: Devismes, S., Mittal, N. (eds.) Stabilization, Safety, and Security of Distributed Systems - 22nd International Symposium, SSS 2020, Austin, TX, USA, November 18-21,

2020, Proceedings. Lecture Notes in Computer Science, vol. 12514, pp. 317–322. Springer (2020). https://doi.org/10.1007/978-3-030-64348-5_25, https://doi.org/10.1007/978-3-030-64348-5_25

20. Zhang, Z., He, W., Cheng, Y., Wang, W., Gao, Y., Liu, D., Li, K., Nepal, S., Fu, A., Zou, Y.: Implicit Hammer: Cross-privilege-boundary Rowhammer through implicit accesses. IEEE Trans. Dependable Secur. Comput. **20**(5), 3716–3733 (2023). https://doi.org/10.1109/TDSC.2022.3214666

A RISC-V Code for SUDO

The RISC-V code for the new version of `sudo_passwd_verify()` can be seen below.

This code is for the patched version of SUDO that includes mitigation for the RowHammer attacks. The Comparison Shortcut attack (Attack A) may happen after the execution of the code in `.L2` (before the `beq` instruction). The Index Skip attack (Attack B) may happen right before the execution of the code in label `sudo_passwd_verify` (before function call). The Result Flip attack (Attack C) may happen right before execution of the second line in `.L2` (the `bne` instruction).

```
sudo_passwd_verify:
  addi    sp,sp,-48    ; allocate stack frame
  sw      ra,44(sp)    ; save return address
  sw      s0,40(sp)    ; save (old) frame pointer
  addi    s0,sp,48     ; new frame pointer
  sw      a0,-36(s0)   ; save arg 0 (*pass)
  sw      a1,-40(s0)   ; save arg 1 (*auth)
  lw      a5,-40(s0)   ; \
  lw      a5,12(a5)    ; | pw_passwd = auth->data
  sw      a5,-20(s0)   ; /
  li      a5,4096      ; \
  addi    a5,a5,-1     ; | strCmpRes = 4095
  sw      a5,-28(s0)   ; /
  j       .L2
.L4:
  lw      a5,-36(s0)   ; \
  addi    a4,a5,1      ; | pass++
  sw      a4,-36(s0)   ; /
  lbu     a5,0(a5)     ; \
  bne     a5,zero,.L2  ; / if (*(old)pass != 0) goto L2
  sw      zero,-28(s0) ; strCmpRes = 0
  j       .L3
.L2:
  lw      a5,-36(s0)   ;
  lbu     a4,0(a5)     ;
  lw      a5,-20(s0)   ; \
  addi    a3,a5,1      ; | pw_passwd++
  sw      a3,-20(s0)   ; /
  lbu     a5,0(a5)     ; \
  beq     a4,a5,.L4    ; / if(*pass == *(old)pw_passwd) goto L4
.L3:
  lw      a5,-28(s0)   ; \
  beq     a5,zero,.L5  ; / if(strCmpRes == 0) goto L5
  lw      a5,-36(s0)   ; \
  lbu     a5,0(a5)     ; |
  mv      a4,a5        ; |
  lw      a5,-20(s0)   ; | strCmpRes = *pass - *(pw_pass - 1)
  addi    a5,a5,-1     ; |
  lbu     a5,0(a5)     ; |
```

```
  sub     a5,a4,a5     ; |
  sw      a5,-28(s0)   ; /
.L5:
  lw      a5,-28(s0)   ; \
  bne     a5,zero,.L6  ; / if (strCmpRes != 0) goto L6
  li      a5,86650880  ; \
  addi    a5,a5,-1755  ; |
  sw      a5,-24(s0)   ; / ret = AUTH_SUCCESS
  j       .L7
.L6:
  li      a5,181784576 ; \
  addi    a5,a5,1754   ; | ret = AUTH_FAILURE
  sw      a5,-24(s0)   ; /
.L7:
  lw      a5,-24(s0)   ;
  mv      a0,a5        ;
  lw      ra,44(sp)    ; restore return address
  lw      s0,40(sp)    ; restore frame pointer
  addi    sp,sp,48     ; pop stack frame
  jr      ra           ; return
```