Hardware-Assisted Program Execution Integrity: HAPEI

Ronan Lashermes, Hélène Le Bouder, Gaël Thomas

To cite this version:

Ronan Lashermes, Hélène Le Bouder, Gaël Thomas. Hardware-Assisted Program Execution Integrity: HAPEI. NordSec 2018 : 23rd Nordic Conference on Secure IT Systems, Nov 2018, Oslo, Norway. 10.1007/978-3-030-03638-6_25 . hal-01978181

HAL Id: hal-01978181

https://inria.hal.science/hal-01978181

Submitted on 11 Jan 2019

Hardware-Assisted Program Execution Integrity: HAPEI

Ronan Lashermes[1], Hélène Le Bouder[2], and Gaël Thomas[3]

[1] INRIA-RBA SED&LHS, France
ronan.lashermes@inria.fr
[2] IMT-Atlantique, France
helene.le-bouder@imt-atlantique.fr
[3] DGA-MI, France

Abstract. Even if a software is proven sound and secure, an attacker can still insert vulnerabilities with fault attacks. In this paper, we propose HAPEI, an Instruction Set Randomization scheme to guarantee Program Execution Integrity even in the presence of hardware fault injection. In particular, we propose a new solution to the multi-predecessors problem. This scheme is then implemented as a hardened CHIP-8 virtual machine, able to ensure program execution integrity, to prove the viability and to explore the limits of HAPEI.

Keywords: Program Execution Integrity, Control Flow Integrity, Hardware Fault Attacks, Instruction Set Randomization

1 Introduction

In order to ensure the security of an application, developers have to do every thing they can to reduce the number of bugs that could lead to vulnerabilities. But for the most critical applications, software must be proven correct. Yet one bug missed and an attacker can, in some cases, execute arbitrary code. Moreover, this bug can be absent in the binary but created at runtime with a hardware fault injection, breaking software proofs assumptions.

Motivation The problem is illustrated below with a simple example.

Listing 1.1. A simple loop in C.

```
int count = 0;
for(int i = 0; i < 100; i++) {
    count++;
}
```

The assembly code corresponding with the loop in listing 1.1 can be seen in listing 1.2.

Listing 1.2. The same loop in x86 assembly.

```
movl     $0, -8(%rbp) // count = 0
movl     $0, -4(%rbp) // i = 0
jmp      .L2
.L3:
addl     $1, -8(%rbp) // count++
addl     $1, -4(%rbp) // i++
.L2:
cmpl     $99, -4(%rbp) // compare i and 99
jle      .L3 // if i <= 99, jump to .L3 else continue
```

In this case the program execution must ensure:

- instructions are executed in order,
- it is not possible to jump to an arbitrary instruction of the loop. Only the "landing" instructions can be jumped to (first ones after L2 or L3).
- When executing a "landing" instruction, the previous state of the program must be correct. E.g. the program state is one of the two authorized ones (a proper definition of program state is given in section 4). This implies that the jumps are all legitimate.
- No instruction can be overwritten, no instruction can be skipped.

These guarantees must be valid even if the attacker is able to arbitrarily modify instructions at runtime. In this case, execution must stop to prevent further damages.

Contribution This work proposes HAPEI to ensure that the intended software is what is actually running on the chip. Inspired by SOFIA [13], the solution is a hardware Instruction Set Randomization (ISR) scheme that ensures Instructions Integrity (II) and Control Flow Integrity (CFI), even in the case of a Hardware Fault Attacks on Instructions (HFAoI). We demonstrate that we can harden a binary without any modification in the compilation chain with a CHIP-8 virtual machine implementation. It means that the Instruction Set Architecture (ISA) does not have to change in the compiler's view: HAPEI is transparent at the software level. During application installation (also called packing), the instructions are encrypted. The encryption scheme encodes the authorized program states and the transitions from one state to the next (effectively encoding the Control Flow Graph (CFG)). During execution, instructions are decrypted on-the-fly. The decryption is correct only if the program state is correct and the control flow graph is followed.

Organization In this paper, we start with the context and the necessary definitions in section 2. After a review of similar relevant works in section 3, the theory behind HAPEI is presented in section 4. A discussion on the security of the scheme follows in section 5. We propose an implementation detailed in section 6 and finally our conclusion is drawn in section 7.

2 Context

In order to specify and verify our proposal, we must precisely define the capacities of the attacker and the integrity guarantees we provide.

Attacker models Several models are considered:

- Code Injection Attack (CIA): an attacker tries to divert the control flow to execute its own malicious payload.
- Code-Reuse Attack (CRA): an attacker tries to execute a malicious payload composed by a sequence of legitimate pieces of programs (often called widgets in Return-Oriented Programming).
- Hardware Fault Attacks on Instructions (HFAoI): the attacker can edit the program, at runtime, by modifying instruction values.
- Hardware Fault Attacks on Data (HFAoD): the attacker can edit the program, at runtime, by modifying data values.

Integrities To protect against these attacks, the execution of the software must enforce guarantees:

- Control Flow Integrity (CFI): the control flow cannot be modified (no arbitrary jumps). The control flow follows the valid CFG.
- Instructions Integrity (II): the instruction values must not be altered.
- Data Integrity (DI): the data handled by the program must not be altered.
- State Integrity (StI): the processor state (configuration, registers, program counter, . . .) must not be altered.

Often, DI, II and StI are considered together under the name of System Integrity (SI). Here we call Program Execution Integrity (PEI) the combination of CFI and II.

Our scheme, HAPEI, ensures PEI in order to protect against CIA, CRA and HFAoI. Yet to be complete, a solution should also ensure DI. In our opinion, one of the best solution would be to encrypt all data with one secret key per application. Since in the following we consider only one application executing, we suppose that data integrity is guaranteed at a higher level.

StI is probably the most difficult to guarantee in presence of an attacker with hardware fault injection means. It is also very implementation specific, we discuss in section 5.3 how an attack can be achieved on our implementation because StI is not guaranteed.

If numerous works discuss how to ensure integrities, most consider only CIA and CRA attacker models. Yet, hardware fault attacks are a reality and must be mitigated. Unfortunately, HFAoI is a much more powerful attacker model and most previous schemes do not protect against it (cf section 3).

3 Previous works

This work inherits from a long list of proposals to ensure CFI and SI in the literature. In this section, we present the most relevant works (that we know of) and show where they differ with HAPEI. In most cases, CFI and SI are viewed as orthogonal and protected with different solutions.

CFI [3] consists in ensuring that the control flow cannot be tampered with. A large literature exists on the subject, a recent review article by Burrow *et al.* [4] compares many solutions. Paraphrasing Burrow *et al.* [4], ISAs usually have two forward control-flow transfers: jumps and routine calls. We consider separately direct jumps (where the destination address is static) and indirect jumps where the destination address can only be determined at runtime. Most CFI solutions try to verify that jumps can only reach legitimate addresses (forward edges). A special case is the `RETURN` instruction that jump to the value on top of the stack, to return from a routine call (backward edges). As a consequence, a part of the CFG can be determined statically, but another part cannot due to indirect jumps.

Abadi *et al.* [3] shows software CFI implementations: they propose code snippets to replace dangerous instructions (indirect jumps. . .) in order to guarantee CFI.

Tice *et al.* [15] demonstrate a software solution that leverages the compiler to automatically insert the appropriate protections at jump sites (forward edges only). In particular, they tackle the problem caused by virtual method tables, necessary in some programming languages (*e.g.* C++) to enforce runtime polymorphism. In this situation, the method to call is decided at runtime and thus requires an indirect call.

Backward edges (*e.g.* `RETURN` instruction) are traditionally protected with a shadow stack [9]: the call stack is duplicated. On a `RETURN` instruction, the return address on both stacks are read then compared. If they differ, an alarm is triggered. Another possibility explored by Davi *et al.* [7,6] is to add instructions to the ISA for the sole purpose of validating function calls and returns. On any indirect function call, the processor switch to a particular state. The next instruction must be a special `CFIBR label` instruction in order to continue execution. The label is used to keep track of which functions are currently executing.

These methods are efficient but focus on preventing CRAs. Because they are more difficult to achieve, Hardware Fault Attacks are not considered. However, hardware fault attacks have been known for some time. Dehbaoui *at al.* [8] shows that electromagnetic pulses allow to recover a cryptographic secret key. Then Moro *et al.* [12] describes the faulting mechanism and how it translates in a software model. Hardware fault attacks can be generated by software. The recent CLKSCREW work by Tang *et al.* [14], where the authors modify a phone's energy and clock controller to inject faults, demonstrates such an attack. Another illustration is given by the RowHammer attack [16]. These attacks are relevant and must be mitigated: HAPEI must ensure CFI even in the presence of HFAoI.

CFI ensures that jumps, routine calls and returns are legitimate, but it does not prevent an attacker to alter any other instruction. New mitigation techniques should be used for that: II and DI must be ensured.

Most system integrity techniques rely on the encryption of memories, preferably with dedicated hardware. Danger *at al.* [5] introduce a new instruction to selectively randomize parts of a program. Closer to us, Hiscock *et al.* [10] propose a scheme that encrypt the whole application using a stream cipher. In order to deal with multi-predecessors (where one instruction may have several predecessors, thus breaking the stream pattern), the authors re-init the stream cipher.

When specifically applied to instructions, which must be therefore decrypted on-the-fly at execution, the technique is called Instruction Set Randomization (ISR) [11]. Without the secret key, the attacker is unable to alter an instruction and predict the result after decryption. One of the most complete solution, ensuring both CFI and II is SOFIA [13]. This work is the main inspiration for HAPEI.

In SOFIA, to ensure CFI we must encode the authorized state of the program. The solution, proposed in [13], is to mask the instructions with a key stream depending on the Program Counter (PC) and the previous Program Counter PC_{prev} (for the previous instruction). Let i be an opcode (instruction value) and i' the corresponding encrypted opcode. Let E_k be an symmetric encryption function with secret k.

$$i' = E_k(PC_{prev}||PC||...) \oplus i \tag{1}$$

This elegant solution ensure that an instruction can be decoded only if PC and PC_{prev} are correct. Effectively, it encodes all the possible successions of instructions and the correct instruction can be decoded only upon correct PC and PC_{prev} values. In order to ensure II, a Message Authentication Code (MAC) is computed and verified per batch of up to 6 instructions. The MAC value is stored as two words at the beginning of each block. If an instruction has two predecessors, a special case must be made: the multiplexor block. In this block, the two first words correspond to the encrypted MAC values for the two possible predecessors: $M'_{1e1} = E_k(PC^1_{prev}||PC||...) \oplus M1$ and $M'_{1e2} = E_k(PC^2_{prev}||PC||...) \oplus M1$. The encrypted MAC value not used (corresponding to the wrong predecessor) must be skipped in a software transparent way.

We acknowledge the power of this solution, and build our own upon it. Our main issue with SOFIA is the separation between CFI and II. Since the CFI mechanism relies on the Program Counter and not on the instruction value, an additional mechanism is needed to ensure II. Finally the multiplexor block must deal with possible predecessors in a non trivial way. It modifies the control flow according to the actual predecessor where it should be predecessor agnostic (all legitimate predecessors must be dealt with in the same way).

Our paper proposes an new solution to these problems, by relaxing the efficiency requirements.

4 HAPEI

4.1 Phases

In order to harden the application, a preparatory step is required to encrypt the instructions. Only then, the application can be executed.

Packing Packing is required to create the encrypted program, enriched with the necessary metadata, following the scheme detailed below. It must be done on the final device, since it requires a device-specific secret key k.

Execution During execution, the encrypted instructions are deciphered on-the-fly and executed. The decryption can only occur if the program state is correct.

4.2 Program Execution Integrity

SOFIA encodes the state of the program as the succession of PC_{prev} and PC. We propose instead to encode the state of the program as the history of all previous executed instructions. Our proposal does not depend on the PC value (apart when encoded in an instruction value). As such, the machine code is ensured to be executed correctly: instructions integrity is ensured together with control flow integrity.

Secondly, it becomes easy to check at anytime during execution that the current state of the program is valid.

We suppose that the Control Flow Graph (CFG) of our program is perfectly defined at compile time. There is no ambiguity on the destination address of jumps and calls. This condition is trivially satisfied if there are no indirect jumps or calls in our program. In the other case, it can be more tricky.

Let acc_n (standing for accumulator at instruction n) be a value representing the state of the program when instruction i_n is about to be executed (n uniquely identify one instruction). i_n and acc_n can be seen as values in $\mathbb{F}_{2^w}$ and $\mathbb{F}_{2^b}$ respectively for some w and b. b is the instruction size (considered fixed) and w is the security parameter.

Bootstrap To bootstrap the encoding, one has to use an initialization vector IV as input for the first executed instruction.

$$acc_{init} = HMAC_k(IV) \tag{2}$$

acc_{init} is considered as a predecessor program state to the entry instruction. It may be used in a multi-predecessor scheme or in the 1-predecessor one.

1-*predecessor* The easy case is the 1-predecessor case. Our program snippet is a succession of instructions $[i_1, i_2, \cdots, i_n, \cdots]$ where all instructions are executed in order.

The instructions are encoded as

$$i'_n = C\left(acc_n\right) \oplus i_n, \tag{3}$$

where C is a compression function: a projection from $\mathbb{F}_{2^b}$ to $\mathbb{F}_{2^w}$. C must ensure that x cannot be deduced from $C(x)$.

Obviously the state of the program must be updated, using secret key k:

$$acc_{n+1} = HMAC_k(acc_n || i_n). \tag{4}$$

You can see that the state of the program is encoded with a hash chain depending on all previous instructions. The encoded instruction i'_n can only be decoded when the previous state of the program acc_n is correct. This is possible only when instruction i_n is due. Decoding necessitates the same operations:

$$acc_n = HMAC_k(acc_{n-1} || i_{n-1}), \tag{5}$$
$$i_n = C\left(acc_n\right) \oplus i'_n. \tag{6}$$

2-predecessors, a naive and limited solution Most programs necessitate branching. As a consequence, some instructions have 2 predecessors (2 possible previous instructions at two different locations in the program).

As a consequence the previous state of the program may have 2 different values: acc^1_n or acc^2_n. 1 out of the 2 possible values is required to decode i_n.

Let $\Sigma = acc^1_n \oplus acc^2_n$, we can encode our instruction as:

$$\{\Sigma, i'_n = C\left(acc^1_n \cdot acc^2_n\right) \oplus i_n\}, \tag{7}$$

i.e. the previous state is encoded as $acc^1_n \cdot acc^2_n$.

Two cases are possible: the previous state is $acc_n = acc^1_n$ or $acc_n = acc^2_n$. Either case, the decoding is:

$$i_n = C\left(acc_n \cdot (acc_n \oplus \Sigma)\right) \oplus i'_n \tag{8}$$

Yet this scheme has a huge weakness: it is impossible to encrypt the program if cycles are present in the control flow. E.g. a loop's first instruction has two predecessors acc^1_n and acc^2_n where acc^2_n is the state of the program at the end of the loop. Then it becomes infeasible to compute acc^2_n: it depends on $acc^1_n \cdot acc^2_n$. The self-reference cannot be solved, since in this case it would violate Hash-based Message Authentication Code (HMAC) security requirements: one should not be able to find a preimage given an output.

So this scheme works only if the control flow graph is a Direct Acyclic Graph (DAG) which is very limiting in real life scenarios. Instead two solutions (A and B) are proposed below with different security implications developed in section 5.

p-predecessors, solution A It is possible to generalize in order to allow up to p predecessors for an instruction and for a control flow with cycles.

In order to allow cycles, we must "rebase" our program state for all instructions having several predecessors. In this case, the program state is a new uniformly random value (noted r below). The problem is now to map valid predecessor states to this same new state.

Let r be a random value in $\mathbb{F}_{2^b}$. Let $acc^i_n, i \in [\![1, p]\!]$ be the allowed previous accumulator values for current instruction i_n. A polynomial P can be defined such that $\forall i \in [\![1,p]\!], P(acc_n^i) = r$ using Lagrange interpolation. Since the generated polynomial is minimal, it is constant if we do not define an additional point. $P(0) = d$ for d a random value (different than r) in $\mathbb{F}_{2^b}$.

The p coefficients of P are stored as program metadata with the corresponding instruction i_n. At packing, HAPEI encrypts with $i'_n = C(r) \oplus i_n$. To decrypt instruction i_n, we use $acc_n = P(acc_{n-1})$. Note that the polynomial evaluation replaces the HMAC call.

p-predecessors, solution B $\mathbb{F}_{2^b}$ can be decomposed in different subgroups μ_p where

$$\forall x \in \mu_p, x^p = 1 \tag{9}$$

(subgroup of pth-root of unity). Such subgroups exist for all p such that $p \mid 2^b - 1$.

For all valid p (depending on b), we can define a scheme that allows p predecessors. For example, $b = 16$ allows a scheme with 5 predecessors ($p = 5$ divides $65535 = 2^{16} - 1$).

Let $acc_n^i, i \in [\![1,p]\!]$ be the allowed previous accumulator values for current instruction i_n. Let r be a random value in $\mathbb{F}_{2^b}$. Let $m \in \mu_p$ be a generator value for the subgroup. We construct a polynomial P (in $\mathbb{F}_{2^b}$) using Lagrange interpolation such that $\forall i \in [\![1,p]\!], P(acc_n^i) = r \cdot m^i$. The $p-1$ coefficients of P are stored as program metadata with the corresponding instruction i_n. At packing, HAPEI encrypts with $i'_n = C(r^p) \oplus i_n$. To decrypt instruction i_n, we use $acc_n = P(acc_{n-1})^p$. Indeed, by construction

$$\forall i \in [\![1,p]\!], P(acc_n^i)^p = \left(r \cdot m^i\right)^p = r^p \cdot \left(m^p\right)^i = r^p. \tag{10}$$

In this scheme, polynomials have degree $p-1$ instead of p in solution A: the memory overhead is lower.

Ensuring Instruction Integrity To check the II, it is enough to check an acc_n against a truth value pre-generated at packing time. The more frequent the check, the sooner a tempering is detected but the bigger is the required metadata.

A second strategy is to have valid instructions forming a set $I_v \subset \mathbb{F}_{2^w}$. If $card(I_v) << card(\mathbb{F}_{2^w})$ a wrongly decoded instruction will have a very low probability of belonging to I_v, of being valid.

4.3 Key management

In this scheme, the component responsible for managing the secret key k is critical. In most cases, the binary encryption cannot be performed at compilation on the binary provider machine since it would requires to ship the (then shared) secret along with the binary. As a consequence, any Instruction Set Randomization (ISR) scheme using a secret key must have a packing phase that transform an unmodified binary (or extended with the CFG information) into a hardened one.

The only other possibility is for the binary provider to encrypt the application for each intended recipient, then to use public-key cryptography to share the corresponding secret key with the targeted hardware.

4.4 Limitations

Apart from the performances overhead, our solution has severe limitations. Since the CFG must be perfectly known at packing time, indirect jumps and calls should be avoided. In particular, the scheme is not compatible with virtual method tables required for runtime polymorphism in several languages (C++, java, ...). Additionally, the scheme is tailored for self-contained applications. If the program must call external code (shared library, OS system call), things do not play well. How to lift these limits should be investigated by the community.

5 Security Assessment

In this section, the security of the solution is analysed. As with most equivalent schemes, the details are critical. In section 5.1, we discuss about the security of the proposed schemes. In section 5.2, we analyse the security problems due to the use of a stream cipher and how to overcome it. Finally, in section 5.3, the limits of the Program Execution Integrity (PEI) guarantees are shown.

5.1 Scheme security

The scheme security relies on the secrecy of the key stream, the accumulator values must remain secret. Can the attacker gain information on one accumulator value, given she knows the encrypted instructions, the clear instructions (in the most advantageous case for her) and the polynomials ? If she learn a given acc_{n-1}, then no information is gained on acc_n without the knowledge of the secret key k per the cryptographic properties of the HMAC.

First, she can deduce $C(acc_n) = i'_n \oplus i_n$. If C is a cryptographically secure hash function, then no information is gained on acc_n. Lower constraints on C are possible, since we only care about the correct preimage security: the attacker must find the correct preimage, not just a satisfying one.

p-predecessors, solution A Let i_n be a p-predecessors instruction:

$$\exists P \in \mathbb{F}_{2^b}\left[X\right] | \forall i \in [\![1, p]\!], P(acc_n^i) = r \tag{11}$$

with r a random value. P is a public non-constant polynomial but all acc_n^i and r are secret.

Knowing P, the attacker cannot find any acc_n^i nor r: r can be any value in $\mathbb{F}_{2^b}$ and for most r she can find corresponding valid acc_n^i. Yet if she learn r, then finding the roots of $P(X) - r$ is trivial. If she learn a given acc_n^i, then she can compute $r = P(acc_n^i)$, then find the other accumulator values. As a consequence, a polynomial links all corresponding secrets together. If one value is discovered, all the others are too.

A same accumulator value can be used as a legitimate input to several polynomials. Yet the resulting systems of equation are always underdetermined. There is one unknown per polynomial corresponding to the random r value, plus at least 1 secret acc_n^i value. But there again, in this case all secrets are linked: discovering one may mean discovering the others.

The problem with solution A is that P is constructed in a very specific way: Lagrange interpolation ensures that $P(X) - r$ has p distinct roots (P has degree p), the maximum possible. The attacker can use the peculiarity to gain information on r and $acc_n^i, \forall i$. Given a random polynomial Q of degree p, the probability that Q has p distinct roots corresponds to the number of combinations to distribute p roots over 2^b values divided by the total number of polynomials of degree p.

$$\frac{\binom{2^b}{p}}{(2^b)^{p+1}}. \tag{12}$$

Since in our case, $p << 2^b$, equation (12) becomes

$$\frac{1}{p!\,(2^b)^{p+1}}. \tag{13}$$

As a conclusion, a proportion of $\frac{1}{p!}$ random polynomials have p roots. The greater the p, the better for the attacker that becomes able to discriminate r. In most cases, p is low and 2^b is high ($b \geq 128$) so the security should not be compromised since the attacker cannot possibly enumerate all possible r.

p-predecessors, solution B This possibility for the attacker to discriminate r given P is the main motivation for the alternative solution B. In this proposition, r is not a special value with respect to P: $P(X) - r \cdot m^i$ may have any number of roots (≥ 1). But then, it means that additional roots may be considered valid program states. Some random illegitimate accumulator values could be mapped to the legitimate one. Since the attacker should not be able to control the accumulator value, the security is not compromised.

Finally, the choice between solutions A and B depends on the attacker model: if she can control acc_n, then solution A must be chosen. If not but she has a huge computation power, solution B should be preferred.

5.2 Differential Attack

If the attacker is able to see the plain/decrypted instructions (or deduce from observed behaviour), she can execute one arbitrary instruction i_a.

$$i_n' \oplus e = C\,(acc_n) \oplus i_n \oplus e \tag{14}$$

To execute i_a, simply choose $e = i_n \oplus i_a$. But the next state of the program is

$$acc_{n+1} = HMAC_k(acc_n || i_a)$$

which is unpredictable for the attacker by the required properties of $HMAC_k$. This attack is present in all schemes using a key stream (xoring a secret data with the text).

The mitigation is to wait for i_{n+1} valid decryption before executing i_n. In this case, if the attacker tries to force execution of i_a instead of i_n, II detects the bad behaviour (cf section 4.2).

5.3 Multi-successors attack

In the proposed scheme, several instructions can have the same associated program state. An example is given in listing 1.3.

Listing 1.3. A pseudo assembly program

```
i0: CMP R0, #0 // Compare register R0 with 0
i1: BEQ 3 // Go to i3 if equal
i2: JUMP 4 // Go to i4
i3: ...
```

In this example, the possible transitions from instruction i_1 are $i_1 \Rightarrow i_2$ or $i_1 \Rightarrow i_3$. i_1 has two successors but both i_2 and i_3 have one predecessor. As a consequence, $acc_2 = acc_3$ and encrypted instructions differential is conserved: $i'_2 \oplus i'_3 = i_2 \oplus i_3$.

The attacker can switch these instructions at will and they will be correctly decoded. A mitigation would require to includes a unique identifier in the accumulator update formula:

$$acc_{n+1} = HMAC_k(acc_n || i_n || n). \tag{15}$$

But such an attack does not break Program Execution Integrity: the execution where the attacker switches the instructions is indistinguishable from a legitimate execution apart from instruction addresses. The program is semantically correct. And if the next instructions do not correspond to a legitimate program execution, they cannot be correctly decrypted. In conclusion, this attack illustrates the limits of the PEI guarantees. The Program Counter PC is part of the processor state: StI is the guarantee that should prevent this attack.

6 Implementation

In order to test HAPEI, we implement it by modifying a CHIP-8 virtual machine to run hardened programs. The sources for the reference and the hardened implementations can be found at `https://gitlab.inria.fr/rlasherm/HAPEI`. Licenses are MIT for software and CC-BY-4.0 for non-software.

6.1 CHIP-8

CHIP-8 is an Instruction Set Architecture (ISA) initially intended to be run in a virtual machine on 8-bits microcomputers (from the 1970s). Its purpose is to run the same video games on different hardware. It is a good candidate for a hardened implementation because of its simplicity: 35 instructions with only 1 indirect branch instruction. Binaries (called roms in the video game emulation tradition) can be freely found on the internet. Additionally, its age means that it can easily be run on any modern computer, even with additional cryptographic computations, in real-time.

Our goal is therefore to run these roms in a hardened virtual machine. A simulated fault injection process, a key press modifies the next opcode by a randomly valid one, must be detected. In order to validate the hardened implementation, we compare it to a reference implementation (without the hardening) and compare the behaviours.

The implementation is modularized: **chip8lib** contains all common structure definitions and the machinery to parse opcodes into instructions (sum type values). **chip8ref** is the reference implementation, able to run, display and interact with emulated video games. **chip8hard** is the hardened implementation, it packs the current rom at startup then executes its encrypted version according to the scheme presented in this paper (solution A).

6.2 Reference implementation

The two implementations have been done in the rust language. Rust has great performances and allows a simple representation of the virtual machine by using sum types. The implementation has been inspired by the previous work at [2], but modularized to factor code between the reference and hardened implementations. The virtual machine is a 8-bit machine (word size) with 16-bit addresses.

6.3 Hardened implementation

Packing The hardened implementation must pack the application before execution. This step requires a precise control flow graph extraction. This extraction is done in a classical way. First we define a method *cfg_next* that given an instruction, its address and the call stack (stack to keep track of routine calls and returns) return all addresses that can possibly be executed next (and update the call stack). Then starting from the first address, we recursively call *cfg_next* on the next instruction for all possible call stacks. Meaning that if the next instruction has already been included in the CFG previously but the call stack is different than the one during the previous CFG inclusion, we continue the analysis with the new call stack.

The difficulty lies in indirect branches that make CFG extraction difficult. In the CHIP-8, there is only one such instruction `JP V0, addr` that jumps to address `addr` plus the content of register `V0` (8-bit register). In our CFG extraction, we consider that the possible successors for this instruction are all addresses between `addr` and `addr` + 255. Fortunately, all roms do not use this instruction.

Once the CFG has been extracted, we compute all accumulator values (program states), polynomials and finally encrypt our instructions in the following order:

1. acc_{init} from IV.
2. For all multi-predecessors instructions, draw a new random accumulator value. acc_{init} is a predecessor for the entry instruction.
3. Compute recursively all remaining accumulator values.
4. Compute and store polynomials for all multi-predecessors instructions.
5. For all instructions, encrypt it using corresponding accumulator value.
6. Delete all accumulator values, we have to compute them on-the-fly at execution.

Execution At execution, the binary is already encrypted. At each tick of the virtual machine, the following actions are performed in order to execute instruction i_n at address PC:

1. Is there a polynomial P associated with address PC ?
2. If yes, then its a multi-predecessors case: update the accumulator state $acc = P(acc)$.
3. If there is no polynomial, then simply update the accumulator with the HMAC function: $acc = HMAC_k(acc||i_{previous})$
4. Then decrypt the instruction to be executed: $i_n = i'_n \oplus C(acc)$.
5. If i_n is valid we can execute it, in the other case we are under attack.

Only one accumulator value must be remembered throughout the computation, lowering the cost of our solution. This cost is both a big performance hit due to the on-the-fly decryption and accumulator update, and a memory overhead required to store the polynomials. Since our implementation is a virtual machine, the performance overhead cannot be meaningfully measured. But the memory overhead can be precisely measured as shown on table 1.

In this table, the hardening is performed for a set of binaries found in [1]. We can see that the memory requirements at the 128-bit security level (size of one field element) is important: more memory is required to store the polynomials than the initial binary size.

Additionally, the roms are run in the reference virtual machine and in the hardened virtual machine to confirm functional equivalence. Then a simulated hardware fault injection mechanism is inserted. When a specific key is pressed, the next opcode is replaced in memory with a random valid opcode. On the reference implementation, the results of this fault injection is unpredictable: strange patterns are displayed on screen, nothing happen, another game screen is unlocked or we get a crash. In the hardened machine, the fault injection means that all subsequent instructions will be wrongly decoded: a crash always follows because of an invalid instruction value.

Table 1. Hardening memory usage for a set of CHIP-8 roms found in [1] (solution A).

ROM name	ROM byte size	Instructions count	Polynomials count	Field elements count	Polynomials byte size (128-bit)
INVADERS	1283	202	28	99	1584
GUESS	148	49	8	25	400
KALEID	120	59	10	32	512
CONNECT4	194	67	5	19	304
WIPEOFF	206	101	15	47	752
PONG2	264	126	19	60	960
15PUZZLE	384	116	17	54	864
TETRIS	494	189	32	106	1696
BLINKY	2356	856	84	310	4960
VBRIX	507	218	27	93	1488
SYZYGY	946	414	44	149	2384
BRIX	280	134	17	57	912
TICTAC	486	194	23	89	1424
MAZE	34	13	3	10	160
PUZZLE	184	87	10	34	544
BLITZ	391	121	15	47	752
VERS	230	103	24	73	1168
PONG	246	117	18	57	912
UFO	224	106	15	48	768
TANK	560	236	42	139	2224
MISSILE	180	75	12	37	592
HIDDEN	850	258	24	81	1296
MERLIN	345	124	13	45	720

7 Conclusion

In this paper, a solution to ensure Program Execution Integrity is presented. More precisely, Control Flow Integrity and Instructions Integrity are guaranteed against Code Injection Attack, Code-Reuse Attack and Hardware Fault Attacks on Instructions. This solution uses the program state, a hash chain of all previously executed instructions, in order to encrypt the program. Correct decryption can only be achieved if the program state is correct.

The difficulty lies in the multi-predecessors case: how to handle the stream cipher when an instruction has several predecessors ? Here, the program state is reinitialized with a random value and a polynomial is computed that maps all previous program states to this new value.

An implementation has been proposed as a CHIP-8 virtual machine. It shows that the memory overhead is important and validates that the proposed scheme is functional.

Further work can be done to optimize the performances: can we find better mapping function than polynomials ? Is there a more compact representation of the program state, offering the same security level ?

This work shows that instruction set randomization has a lot to offer in order to provide guarantees at the hardware level.

References

1. CHIP-8 games pack, https://www.zophar.net/pdroms/chip8.html
2. Mike zaby's CHIP-8 implementation, https://github.com/mikezaby/chip-8.rs
3. Abadi, Martín and Budiu, Mihai and Erlingsson, Úlfar and Ligatti, Jay: Control-flow integrity principles, implementations, and applications 13(1), 1–40, http://portal.acm.org/citation.cfm?doid=1609956.1609960
4. Burow, N., Carr, S.A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., Payer, M.: Control-flow integrity: Precision, security, and performance 50(1), 1–33, http://dl.acm.org/citation.cfm?doid=3058791.3054924
5. Danger, J.L., Guilley, S., Praden, F.: Hardware-enforced protection against software reverse-engineering based on an instruction set encoding. pp. 1–11. ACM Press, http://dl.acm.org/citation.cfm?doid=2556464.2556469
6. Davi, L., Hanreich, M., Paul, D., Sadeghi, A.R., Koeberl, P., Sullivan, D., Arias, O., Jin, Y.: HAFIX: hardware-assisted flow integrity extension. pp. 1–6. ACM Press, http://dl.acm.org/citation.cfm?doid=2744769.2744847
7. Davi, L., Koeberl, P., Sadeghi, A.R.: Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. pp. 1–6. IEEE, http://ieeexplore.ieee.org/document/6881460/
8. Dehbaoui, A., Dutertre, J.M., Robisson, B., Tria, A.: Electromagnetic transient faults injection on a hardware and a software implementations of AES. pp. 7–15. IEEE, http://ieeexplore.ieee.org/document/6305224/
9. Frantzen, M., Shuey, M.: StackGhost: Hardware facilitated stack protection. USENIX
10. Hiscock, T., Savry, O., Goubin, L.: Lightweight software encryption for embedded processors. pp. 213–220. IEEE, http://ieeexplore.ieee.org/document/8049788/
11. Kc, G.S., Keromytis, A.D., Prevelakis, V.: Countering code-injection attacks with instruction-set randomization p. 10
12. Moro, N., Dehbaoui, A., Heydemann, K., Robisson, B., Encrenaz, E.: Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller. pp. 77–88. IEEE, http://ieeexplore.ieee.org/document/6623558/
13. Ruan de Clercq, Ronald De Keulenaer, Bart Coppens, Bohan Yang, Pieter Maene, Koen de Bosschere, Bart Preneel, Bjorn de Sutter, Ingrid Verbauwhede: SOFIA: Software and control flow integrity architecture. IEEE
14. Tang, A., Sethumadhavan, S., Stolfo, S.: CLKSCREW: Exposing the perils of security-oblivious energy management. In: Proceedings of the Second Workshop on Real, Large Distributed Systems. USENIX, OCLC: 255334142
15. Tice, Caroline and Roeder, Tom and Collingbourne, Peter and Checkoway, Stephen and Erlingsson, Úlfar and Lozano, Luis and Pike, Geoff: Enforcing forward-edge control-flow integrity in GCC & LLVM p. 15
16. van der Veen, V., Fratantonio, Y., Lindorfer, M., Gruss, D., Maurice, C., Vigna, G., Bos, H., Razavi, K., Giuffrida, C.: Drammer: Deterministic rowhammer attacks on mobile platforms. pp. 1675–1689. ACM Press, http://dl.acm.org/citation.cfm?doid=2976749.2978406