Formal Verification of the Heatseeker Technique for Detection of Code-Reuse Attacks

Téo Biton[1,2][0009−0001−6985−0244], Nikolai Kosmatov[1][0000−0003−1557−2813], Olivier Gilles[1][0000−0002−3776−2071], Daniel Gracia Pérez[1][0000−0002−5364−8244], and Sébastien Pillement[2][0000−0002−9160−2896]

[1] Thales Research & Technology, cortAIx Labs, Palaiseau, France
{firstname.name}@thalesgroup.com
[2] Nantes Université, CNRS, IETR UMR 6164, Nantes, France
{firstname.name}@univ-nantes.fr

Abstract. Formal verification of attack detection algorithms is essential to provide strong guarantees of correctness for security-critical protection mechanisms. Such guarantees are particularly important for signature-based defenses against code-reuse attacks, where undetected exploits can lead to the compromise of the entire system. These protection mechanisms provide interesting targets for formal verification, as they are typically implemented at the hardware or hardware/software interface. This paper presents a formal verification of a previously proposed signature-based detection mechanism, called heatseeker, designed to identify jump-oriented programming (JOP) attacks by monitoring control-flow transfers during program execution. We model both the detection algorithm and the target JOP attacks in C, and verify that the considered mechanism effectively detects any JOP attack using the bounded model checker CBMC. Thanks to the definition of suitable invariants on the algorithm's internal state, bounded verification is sufficient to ensure the desired property for all executions. We present the formal model, the invariants, the proof methodology, and discuss the assumptions and limitations of the approach.

Keywords: formal verification · bounded model checking · code-reuse attacks · jump-oriented programming · CBMC

1 Introduction

Software vulnerabilities represent a significant threat to modern computing systems. While traditional security mechanisms like data execution prevention (DEP) [32] and address space layout randomization (ASLR) [43] have mitigated many attack vectors, adversaries have adapted by leveraging *existing code* within vulnerable applications. This type of attacks, called *code-reuse attacks* (CRA) [22], include return-oriented programming (ROP) [37] and jump-oriented programming (JOP) [9]. Unlike *code injection attacks* [33], CRAs do not introduce new executable code; instead, they orchestrate short instruction sequences

present in the code, known as *gadgets* [40], to achieve attacker-controlled behavior. A ROP attack [40] chains gadgets ending in return instructions with manipulated stack data, while a JOP attack relies on a so-called *dispatcher gadget* that orchestrates the attack by coordinating control flow through indirect jumps, effectively neutralizing defenses based on stack monitoring [38]. Memory protections alone are insufficient against such exploits [6], requiring more advanced defenses [1,41,2,23].

Among the existing attack detection techniques, *signature-based detection methods* proceed by identifying specific attack patterns such as gadget length, frequency, and the sequential execution of gadgets forming a chain [46,26]. It is crucial to ensure the correctness of such protection mechanisms, which therefore represent relevant targets for formal verification. Their verification is however rarely done today since these mechanisms are typically implemented at the level of hardware or software/hardware interface.

In a recent work [7], we proposed the *heatseeker* algorithm, a signature-based detection technique tailored to detect JOP exploits through a precise dynamic uncovering of the dispatcher gadget. The algorithm is based on a set of expected properties of JOP attacks, focusing on specific repetitive jumps from the dispatcher gadget and back into it. The detection is based on the frequency of such edges, which makes it resistant to attacks using longer gadgets that defeat some previously proposed methods. We realized a prototype implementation of the method at the hardware level and integrated it into an open-souce RISC-V processor, namely the CVA6 [47].

The purpose of this work is to present the proof[3] that the heatseeker algorithm is able to effectively detect JOP attack signatures defined by our attack model and operational assumptions. We model both the detection algorithm and the target JOP attacks in C, and verify that the heatseeker algorithm effectively detects any JOP attack. We define suitable invariants on the algorithm's internal state, which allow us to apply the bounded model checker CBMC [12] to ensure the desired property for all executions. We describe the formal model and the considered invariants, present the proof methodology, and discuss its assumptions and limitations. The model of the algorithm and attacks used for the proof, as well as the tooling necessary to reproduce the proof, are available in the companion artifact [8].

Contributions. The contributions of this paper include the following:

- the definition of a formal model in C[4] for the heatseeker mechanism that dynamically detects jump-oriented programming attacks;

[3] This proof was only briefly outlined in the previous submission [7] that focused on the design and the experimental evaluation of the solution and did not present its model.

[4] Referring to a C model as a *formal model* may be debatable. We adopt the convention used in certification projects (e.g., [14, Sec. 10.6.1], [18]), where the term *formal model* denotes the model to which formal verification is applied.

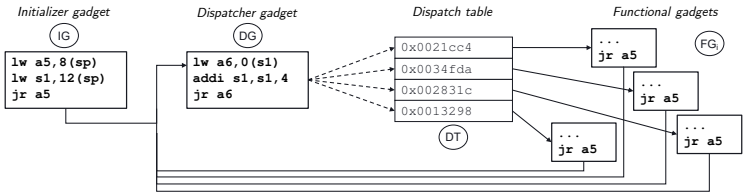

Fig. 1: Generic structure of a jump-oriented programming exploit. It relies on a *dispatcher gadget* that coordinates control flow through indirect jumps to *functional gadgets* after an initial setup with an *initializer gadget*.

- a detailed presentation of the formal verification based on this model using the CBMC bounded model checker that the mechanism effectively detects target attacks;
- a description of the verification methodology, including its main steps, key invariants, assumptions and limitations.

Outline. The remainder of this paper is organized as follows. Section 2 provides a brief background on code-reuse attacks and presents their formalization in our model. Section 3 describes the heatseeker detection mechanism and details its representation in our model. Section 4 presents the proof methodology. Finally, conclusions and perspectives are given in Sect. 5.

2 Jump-Oriented Programming

This section presents the structure of JOP attacks, the attack model we consider, and the expression of JOP attacks used in our C model. For convenience of the readers, in the code extracts shown in the paper we preserve the line numbers of the full model available in the companion artifact. We intentionally inverse the order of presentation with respect to the C model because we believe that the detection mechanism is easier to understand for the readers after the presentation of the attacks.

2.1 Overview and Example of a JOP Attack

JOP [9] is a class of code-reuse attacks that chain short instruction sequences, called *gadgets* [40], already present in a target binary and its libraries. Figure 1 shows the typical structure of a JOP exploit. (A detailed understanding of the instructions written here in RISC-V is not necessary to follow the paper). First, an *initializer gadget* (IG) configures the state of registers and memory. This gadget typically performs operations such as loading values into registers, setting up a *dispatch table*, and preparing necessary memory locations. Once executed, the initializer gadget transfers control to the *dispatcher gadget* (DG) [9,10]. In Fig. 1, it is realized by a jump (`jr` instruction) to the address of the dispatcher gadget stored in register `a5`.

Functional gadgets (FG_i) are the core computational units in JOP: they execute the actual exploit payload, performing operations such as data loads,

$$\begin{aligned}
\text{end}_{\text{IG}} &\rightarrow \text{start}_{\text{DG}} \quad \text{(using register a5)} && (0)\\
\text{end}_{\text{DG}} &\rightarrow \text{start}_{\text{FG}_1} \quad \text{(using register a6)} && (1)\\
\text{end}_{\text{FG}_1} &\rightarrow \text{start}_{\text{DG}} \quad \text{(using register a5)} && (2)\\
\text{end}_{\text{DG}} &\rightarrow \text{start}_{\text{FG}_2} \quad \text{(using register a6)} && (3)\\
\text{end}_{\text{FG}_2} &\rightarrow \text{start}_{\text{DG}} \quad \text{(using register a5)} && (4)\\
\text{end}_{\text{DG}} &\rightarrow \text{start}_{\text{FG}_3} \quad \text{(using register a6)} && (5)\\
&\ldots
\end{aligned}$$

Fig. 2: A sequence of jumps in a JOP attack in Fig. 1, where start_g and end_g denote, resp., the addresses of the start and end instructions of gadget g. For an odd i, it contains a jump from DG to FG_i, and for an even i, a jumps to DG. In real-life attacks, the latter can use a set aR of nR (attack) registers. In our model, each jump is represented by a triple (As_i, Ad_i, R_i) with $i \in [0, NumOp[$, where the start and end addresses are denoted by As_i and Ad_i, and the register by R_i.

arithmetic, memory stores, and system calls. They transfer control back to the dispatcher gadget at the end.

The *dispatch table* (DT) contains the addresses of the functional gadgets in the order in which they should be executed.

The dispatcher gadget (DG) plays a central role in a JOP attack, as it governs the execution flow by invoking functional gadgets using the dispatch table. In Fig. 1, this invocation (at the last line in DG) is performed by a jump to the address stored in register `a6`. Prior to the jump, this address is loaded (`lw` instruction) to `a6` from address `s1` that refers to the current cell in the dispatch table, and `s1` is increased (`addi` instruction) to prepare the jump at the next iteration.

Thus, a JOP attack in our example contains a typical sequence of jumps shown in Fig. 2. The heatseeker method observes the information about indirect jumps and recognizes a signature of such attack sequences. In addition to *repetitions of some addresses and registers*, an important characteristic of such sequences is a *limited distance* between start_{DG} and end_{DG} since the dispatcher gadget typically has a relatively small size, so we assume their difference to be at most equal to a positive integer D, a parameter of the algorithm.

Moreover, in real-life attacks, functional gadgets can use different registers for jumps back to the dispatcher gadget (instead of the same register `a5` in our example). We also refer to them as *attack registers* since the attacker has to control all of them. We assume that the jumps back to the dispatcher gadget use a set aR of nR attack registers. The heatseeker algorithm is capable to detect attacks with up to $MaxR$ attack registers used for such jumps (we explain how this number is determined by the parameters of the algorithm in Sect. 3.1).

```
159 /****** ATTACK MODEL ******/
160 // Assume that given step has a non-zero value
161 void _h_step_non_zero_int(u1 s) {
162   __CPROVER_assume(s != 0);
163 }
164 // Program coherency:
165 // - No jump with same source and target addresses (loop)
166 // - Make sure identical source addresses have identical registers
167 // Non mandatory for the proof but it makes the model more realistic
168 void _h_program_coherency(u1 As[], u1 Ad[], u1 R[]) {
169   for (u1 i = 0; i < NumOp; i++) {
170     __CPROVER_assume(As[i] != Ad[i]);
171     for (u1 j = i + 1; j < NumOp; j++) {
172       if (As[i] == As[j]) __CPROVER_assume(R[i] == R[j]);
173     }
174   }
175 }
176 // Assume number of attack registers nR to be in [0; MaxR]
177 void _h_number_attack_registers(u1 nR) {
178   __CPROVER_assume(nR > 0); __CPROVER_assume(nR <= MaxR);
179 }
180 // Assume Ad to be in [As - D; As[
181 void _h_dispatcher_gadget_bounds(u1 As, u1 Ad) {
182   __CPROVER_assume(As > Ad); __CPROVER_assume(As - Ad <= D);
183 }
184 // Ensure that functional gadgets transfer control to the dispatcher gadget
185 // To avoid spurious counterexamples, ensure that N consecutive functional
186 // gadgets have different source addresses As
187 void _h_functional_gadgets_addresses(u1 As[], u1 Ad[]) {
188   for (u1 i = 0; i < NumOp; i += 2) {
189     __CPROVER_assume(Ad[i] == Ad[0]);  // Repeated dispatcher gadget start address
190   }
191   for (u1 i = 0; i < NumOp; i += 2) {
192     u1 limit = i + 2 * N;
193     for (u1 j = i + 2; j < NumOp && j < limit; j += 2) {
194       __CPROVER_assume(As[i] != As[j]); // New functional gadget end address
195     }
196   }
197 }
198 // Helper function to verify that register R is in the set of nR registers aR
199 u1 __r_in_ar(u1 R, u1 nR, u1 aR[]) {
200   u1 result = 0;
201   for (u1 k = 0; k < nR && k < MaxR; k++) {
202     result |= (R == aR[k]);
203   }
204   return result;
205 }
```

Fig. 3: Attack model, part 1/2.

2.2 Symbolic Representation of Attacks

We represent a JOP attack as a sequence of $NumOp$ steps (As_i, Ad_i, R_i) with $i \in [0, NumOp[$, where each step (As_i, Ad_i, R_i) represents an indirect jump formed by a source address As_i, a destination address Ad_i and a register R_i used in the jump instruction. Let D be a positive integer (a parameter of the algorithm) that will denote a maximal distance between the start and the end of the considered dispatcher gadgets. Let nR denote the number of attack registers, that is, registers used in the jumps from the functional gadgets, and let array aR contain their identifiers.

To prove our main result for any attack sequence, we have to construct a symbolic representation of any attack sequence in CBMC. As usual, a symbolic representation in CBMC is created by assigning non-deterministic values to relevant

```
206 // Constrain the registers of functional gadgets aR and of dispatcher gadget R[1]
207 // Registers in R used by functional gadgets must belong to the nR registers in aR
208 void _h_functional_gadgets_registers(u1 nR, u1 aR[], u1 R[]) {
209   for (u1 i = 0; i < nR && i < MaxR; i++) {
210     __CPROVER_assume(aR[i] != 0);    // Functional gadget registers are non zero
211     __CPROVER_assume(R[1] != aR[i]); // and differ from dispatcher gadget register
212   }
213   for (u1 i = 0; i < NumOp; i += 2) {
214     __CPROVER_assume(__r_in_ar(R[i], nR, aR));
215   }
216 }
217 // Ensure dispatcher gadget transfers control to functional gadgets
218 // To avoid spurious counterexamples, ensure that N consecutive functional
219 // gadgets have different destination addresses Ad
220 void _h_dispatcher_gadget_addresses(u1 As[], u1 Ad[], u1 R[]) {
221   for (u1 i = 1; i < NumOp; i += 2) {
222     __CPROVER_assume(As[i] == As[1]);  // Repeated dispatcher gadget end address
223     __CPROVER_assume(R[i]  == R[1]);   // Repeated dispatcher gadget register
224   }
225   for (u1 i = 1; i < NumOp; i += 2) {
226     u1 limit = i + 2 * N;
227     for (u1 j = i + 2; j < NumOp && j < limit; j += 2) {
228       __CPROVER_assume(Ad[i] != Ad[j]); // New functional gadget start address
229     }
230   }
231 }
232 // Hypothesis on JOP attack with transfers of the dispatcher gadget address
233 void _h_attack_model(u1 nR, u1 aR[],
234   u1 As[], u1 Ad[], u1 R[]) {
235   // Assume the attack model starts with a functional gadget
236   _h_dispatcher_gadget_bounds(As[1], Ad[0]);
237   _h_functional_gadgets_addresses(As, Ad);
238   _h_functional_gadgets_registers(nR, aR, R);
239   _h_dispatcher_gadget_addresses(As, Ad, R);
240   _h_program_coherency(As, Ad, R);
241 }
```

Fig. 4: Attack model, part 2/2.

variables (using calls to `nondet_int`), and by adding necessary assumptions for them. We focus here on the assumptions.

Figures 3 and 4 show the attack model with the assumptions we make to construct a symbolic attack sequence. The model defines an alias `u1` to type **`unsigned int`** on line 2 for the sole purpose of code readability. Some definitions of constants used throughout the model are provided on lines 3–7 of Fig. 5. Lines 233–241 show the function that assumes the whole set of hypotheses by calling other hypothesis functions. It takes as parameters the number nR and the array aR of identifiers of registers used in the jumps from the functional gadgets, and the sequence of jumps (As_i, Ad_i, R_i). As a simple example, function `_h_step_non_zero_int` on lines 162–163 (used in other parts of the model) assumes that its argument is nonzero. We use the CBMC instruction `__CPROVER_assume(cond)` to assume that its argument `cond` is true. Let us describe the key assumptions in this section.

Program Coherency. Function `_h_program_coherency` shown on lines 168–175 (and called on line 240) ensures the coherency of the program. It rules out self-loops (a jump whose source equals its destination) and enforces that identical source addresses in the attack sequence use identical registers (since

the same jump instruction always uses the same register). These assumptions are not mandatory for the proof but make the model more realistic since the excluded situations do not occur in real-life programs. In other words, the only purpose of these assumptions is to make the model closer to real-world programs.

Relative Position of the Start and End Addresses of the Dispatcher Gadget. The attack sequence begins with the initializer gadget, hence the destination address Ad_0 is the start address of the dispatcher gadget, and the source address As_1 of the next step is the end address of the dispatcher gadget (cf. the example in Sect. 2.1 and Fig. 2). The start address must be before the end address, and the distance between these two addresses must be bounded by the algorithm parameter D. These constraints are assumed by function `_h_dispatcher_gadget_bounds` on lines 181–183, called on line 236 with arguments (As_1, Ad_0).

Functional Gadgets' Registers. Recall that $MaxR$ denotes the maximal number of registers used by the jumps from functional gadgets for which the algorithm detects an attack. For simplicity, we used a unique register `a5` in our example in Sect. 2.1. Therefore, we assume that nR belongs to $[0, MaxR]$ by function `_h_number_attack_registers` on lines 177–179 (called from an attack initialization function, which is not shown in Figs. 3 and 4). Function `_h_functional_gadgets_registers` on lines 208–216, called on line 238, constrains the registers of functional gadgets: it assumes in particular that each aR_i is different from the register R_1 used by the dispatcher gadget, and that each functional gadget uses indeed a register from aR. The latter assumption relies on the helper function on lines 199–205.

Functional Gadgets' Addresses. To respect the JOP attack pattern presented in Sect. 2.1 and Fig. 2, we also constrain each even step of the attack sequence to ensure that the functional gadgets transfer control back to the dispatcher gadget start address, i.e. all Ad_i for even i must be equal. This is expressed by the function `_h_functional_gadgets_addresses` on lines 187–197, called on line 237. An additional uniqueness assumption on lines 191–197 will be described later in Sect. 4.3.

Dispatcher Gadget Addresses and Registers. Similarly, the JOP pattern contains the repeatedly used end address of the dispatcher gadget (i.e. all As_i for odd i must be equal), and the repeated use of the same dispatcher register (i.e. all R_i for odd i must be equal). This is expressed by the function `_h_dispatcher_gadget_addresses` on lines 220–231, called on line 239. Symmetrically to functional gadgets, we also make an additional uniqueness assumption on lines 225–230 that will be described below in Sect. 4.3.

3 The Heatseeker Algorithm and its Model

This section presents the heatseeker detection algorithm together with its formal model, which is the basis of the formal verification presented in Sect. 4. The

algorithm is designed to be implemented as a hardware component that monitors the processor's instruction trace in parallel with execution, without impacting performance. Although the formal model presented in this paper is expressed in C[5] to enable the use of the bounded model checker CBMC, the algorithm's structure maps directly to a hardware description: its inputs, i.e. the program counter, next program counter, source register, and instruction type, are all readily available from any processor's execution pipeline. Therefore, a proof of properties on the level of the model provides a strong confidence of correctness of the same properties for any faithful implementation, whether in hardware or software. A detailed presentation of the algorithm and its evaluation in a realistic operational context is provided in [7]. Throughout this section, we refer directly to the C model rather than the corresponding hardware artifacts. The complete source code of the algorithm is provided in Figs. 5 and 6.

3.1 Inputs, Parameters and Main Principle

The heatseeker algorithm processes a stream of indirect jump instructions, each characterized by three values (As, Ad, R): a source address As (the current program counter), a destination address Ad (the next program counter), and the identifier of the source register R that holds the destination address.

The goal of the heatseeker algorithm is to converge towards the start address and end address of a dispatcher gadget during an ongoing attack. Indeed, while the attack signature might seem easy to recognize when looking at the example in Fig. 2, the difficulty of detecting a JOP attack in a real-life setting is that it can start at any moment in the middle of a sequence of perfectly legitimate jumps and rely on different registers. The key idea of the algorithm is to track control-flow transfers by carefully maintaining two internal caches and updating them for each extracted jump data. Following an update, it compares the cache states and raises an alarm when the conditions for a JOP attack are met. Readers interested mostly in the verification aspects of this work can skip the details of jump tracking in the algorithm in Sects. 3.2–3.4 (which is the most original but also the most technical part of it).

The parameters (cf. lines 3–7) include the number of steps $NumOp$ in a sequence of jumps and a maximal dispatcher start-end distance D (already explained in Sect. 2.2) as well as the number of cache lines (entries) N and the minimal number of various gadgets G. In addition, by design of the algorithm, the maximal number $MaxR$ of attack registers for which we are able to prove the target properties is defined as $\lfloor N/2 \rfloor$ (cf. line 7).[6]

[5] Various design choices are possible for the model. Our modeling approach uses local variables and pointers for function arguments. Another design choice could be to use variables instead of pointers for arguments of some functions and to use global variables to model caches.

[6] The intuition behind this definition is that the more attack registers are used in the attack (and controlled by the attacker), the more cache lines are necessary to detect it. Attacks with several registers are more complex to realize and are less likely.

```
#include <stdlib.h>
typedef unsigned int u1;
#define NumOp   5      // Number of steps in attack sequence
#define N       3      // Number of cache lines
#define G       2      // Gadget threshold
#define D      64      // Max dispatcher start-end distance
#define MaxR  (N / 2) // Max number of attack registers (used in func. gadgets)
/******  HEATSEEKER ALGORITHM ******/
// Update given cache with new jump data
void __update_cache(u1 * Ci,                 // Number of recorded cache lines
  u1 Ta[], u1 Tr[], u1 Tga[], u1 Tg[],       // Cache line entries
  u1 uTa, u1 uTr, u1 uTga) {                 // New jump data
  for (u1 i = 0; i < N; i++) {
    if (Ta[i] == uTa && Tr[i] == uTr) {      // Jump data found in cache at index i
      u1 nG = Tg[i];
      if (Tga[i] != uTga && nG < G) nG++;    // New address: increase counter
      // Shift cache lines from 0 (top of the cache) to i (index of cache hit)
      for (u1 j = i; j > 0; j--) {
        Ta[j]  = Ta [j-1]; Tr[j] = Tr[j-1];
        Tga[j] = Tga[j-1]; Tg[j] = Tg[j-1];
      }
      // Record updated jump entry at the top of the cache (index 0)
      Ta[0]  = uTa;      // Update tag address
      Tr[0]  = uTr;      // Update tag register
      Tga[0] = uTga;     // Update gadget address
      Tg[0]  = nG;       // Update gadget counter
      return;            // Return if jump data found
    }
  }
  // Jump data not found in cache: shift cache lines from 0 to N
  for (u1 i = N - 1; i > 0; i--) {           // Line at index N-1 is evicted
    Ta[i]  = Ta [i-1]; Tr[i] = Tr[i-1];
    Tga[i] = Tga[i-1]; Tg[i] = Tg[i-1];
  }
  // Record new jump entry at the top of the cache (index 0)
  Ta[0]  = uTa;            // Update tag address
  Tr[0]  = uTr;            // Update tag register
  Tga[0] = uTga;           // Update gadget address
  Tg[0]  = 1;              // First occurrence of this jump data
  if ((*Ci) < N) (*Ci)++; // Update number of recorded cache lines
  return;
}
// Return index of a full entry, considered as potential dispatcher gadget
u1 __get_most_recent_full_entry(u1 Tg[]) {
  u1 i;
  for (i = 0; i < N; i++) {
    if (Tg[i] == G) return i; // Entry recorded G times with a new address
  }
  return i; // Returns N if no entry found
}
```

Fig. 5: Model of the heatseeker algorithm, part 1/2. Cache-update function implementing LRU management with gadget-counter tracking, and function returning the index of the first entry whose gadget counter equals G.

```
51 // Update caches F, D with new jump data and detect potential dispatcher gadget
52 u1 __handle_jump_instruction(
53   u1 * Fi, u1 * Di,                    // Number of recorded cache lines in F, D
54   u1 Fa[], u1 Fga[], u1 Fr[], u1 Fg[], // Entries of functional cache F
55   u1 Da[], u1 Dga[], u1 Dr[], u1 Dg[], // Entries of dispatcher cache D
56   u1 As, u1 Ad, u1 R                   // New jump data
57 ) {
58   // Update both caches F, D with the input jump information
59   __update_cache(Fi, Fa, Fr, Fga, Fg, Ad, R, As);
60   __update_cache(Di, Da, Dr, Dga, Dg, As, R, Ad);
61   // Try to extract most recent full entries (mrfe) in both caches F, D
62   u1 Fmrfe = __get_most_recent_full_entry(Fg);
63   u1 Dmrfe = __get_most_recent_full_entry(Dg);
64   // Raise no alarm (return 0) if some detection conditions are not met:
65   if (Fmrfe == N || Dmrfe == N) return 0;     // No suspicious entries
66   if (Fr[Fmrfe] == Dr[Dmrfe]) return 0;       // Identical registers
67   if (Fa[Fmrfe] > Da[Dmrfe]) return 0;        // End precedes Start address
68   if (Da[Dmrfe] - Fa[Fmrfe] > D) return 0;    // End-Start exceeds D
69   return 1;     // Raise alarm if all detection conditions are met
70 }
```

Fig. 6: Model of the heatseeker algorithm, part 2/2. Jump-handling function, where both caches are updated with mirrored arguments (lines 59–60), then full entries are retrieved (lines 62–63) and the four detection conditions are evaluated (lines 65–68).

3.2 Cache Architecture

The algorithm maintains two structurally identical caches: the *functional cache* $\mathcal{F}$ and the *dispatcher cache* $\mathcal{D}$. Both have N entries. An entry of index i (with $i \in [0, N[$) of cache $\mathcal{F}$ contains four fields:

- a *tag address* Fa_i and a *tag register* Fr_i, referred together as *tag* (Fa_i, Fr_i),
- a *gadget address* Fga_i,
- a *gadget counter* Fg_i.

Similarly, an entry of index i (with $i \in [0, N[$) of cache $\mathcal{D}$ contains four fields:

- a *tag address* Da_i and a *tag register* Dr_i, referred together as *tag* (Da_i, Dr_i),
- a *gadget address* Dga_i,
- a *gadget counter* Dg_i.

The current number of recorded cache entries (which gives also the index i of the next available line) in $\mathcal{F}$ and $\mathcal{D}$ is denoted, respectively, by Fi and Di. The fields are represented in the model by arrays prefixed, respectively, by F or D. We consider index 0 to be on top of each cache. The use of these elements is illustrated by the parameters of the jump-handling function on lines 52–57. Let us now detail the cache update logic.

3.3 Cache Update Logic

The *tag* of a cache entry is the pair (tag address, tag register). The $\mathcal{F}$ and $\mathcal{D}$ caches mirror the role of source and destination addresses. When an indirect jump (As, Ad, R) is processed, the $\mathcal{F}$ cache receives the tag (Ad, R) with gadget address As, and the $\mathcal{D}$ cache receives the tag (As, R) with gadget address Ad.

This mirroring is visible in the two calls to cache update functions on lines 59–60 inside the jump-handling function.

The intuition beyond this design is that the $\mathcal{F}$ cache captures the convergence of functional gadgets with *repeated jumps to a common destination* (a potential dispatcher gadget's start), while the $\mathcal{D}$ cache captures *repeated departures from a common source* (a potential dispatcher gadget's end) towards distinct functional gadgets. Each new jump is analyzed for potential repetitions of the tag. That is why the tag for $\mathcal{F}$ contains (Ad, R) and that for $\mathcal{D}$ contains (Ad, R), and the notion of tag enables a unified treatment for both caches during cache updates.

Both caches implement a least-recently-used (LRU) replacement policy. The cache update function is shown on lines 10–42. The same function is applied to both caches, whose entries are given as parameters (prefixed here with T instead of F or D, cf. lines 11, 59–60). Other parameters include the number Ci of recorded lines in the considered cache, the tag (uTa, uTr) and the gadget address $uTga$ (cf. lines 10, 12, 59–60).

The function iterates over existing entries (line 13) to check whether the given tag is present in the cache, and handles two cases. On a *cache hit*, i.e. when the input tag matches an existing entry at index i (lines 14–28), the matched entry is promoted to the top of the cache at index 0 (lines 23–26) and shifts entries 0 through $i-1$ downward (lines 18–21). If the incoming gadget address differs from the stored one, the gadget counter is incremented up to its maximum G, indicating that a new gadget address was observed for this tag (lines 15–16, 26). The algorithm does not maintain a history of gadget addresses; this is deliberate, since a JOP attack might alternate between two gadgets and must still be detected.

On a *cache miss*, a new entry is inserted at index 0 with a counter of 1 (lines 36–39), and all existing entries are shifted downward (lines 31–34); the entry at index $N-1$ is evicted. The number of recorded lines is incremented if the cache is not yet full (line 40).

3.4 Detection Conditions

Once both caches are updated (cf. lines 59–60), the algorithm evaluates whether an attack is in progress. It first retrieves (on lines 62–63) from each cache the index of the most recent full entry (mrfe), that is, the most recently used entry—i.e. closest to the top of the cache—whose gadget counter has reached G, via the function on lines 44–50. If such entry is not found, the function returns N.

Variables *Fmrfe* and *Dmrfe* denote the indexes returned for the $\mathcal{F}$ and $\mathcal{D}$ caches, respectively. An alarm is raised (by returning value 1, cf. lines 65–69) if and only if the following four conditions hold simultaneously.

1. *Both caches have a full entry* (line 65): $Fmrfe \neq N$ and $Dmrfe \neq N$, meaning that both caches $\mathcal{F}$ and $\mathcal{D}$ contain an entry whose gadget counter equals G, i.e. a sufficient number of distinct gadget addresses—source addresses for $\mathcal{F}$ and destination ones for $\mathcal{D}$—were observed for each tag.
2. *Distinct registers* (line 66): $Fr_{Fmrfe} \neq Dr_{Dmrfe}$. In the context of a JOP attack, these registers must differ for a correct execution of the JOP chain.

3. *Address ordering* (line 67): $Fa_{Fmrfe} \leq Da_{Dmrfe}$. The tag address from cache $\mathcal{F}$ (a potential dispatcher start address) must precede or be equal to the tag address from cache $\mathcal{D}$ (a potential dispatcher end address).
4. *Bounded distance* (line 68): $Da_{Dmrfe} - Fa_{Fmrfe} \leq D$. The distance between the dispatcher's start and end addresses must not exceed the maximum distance, chosen as a parameter tuned to the characteristics of dispatcher gadgets in target binaries. Although the algorithm does not constrain the size of functional gadgets, bounding the size of the dispatcher gadget is necessary to ensure that the addresses output by the two caches are correlated, thereby reducing the likelihood of false positives.

3.5 Initial State

We define the initial state of the algorithm as a state in which all cache lines have no recorded entries. This is represented with default values: all fields are zero and the counters of recorded cache lines Fi and Di are zero.

4 Invariants and Proof of Attack Detection

This section presents the invariant properties of the heatseeker algorithm's internal state and the proof of attack detection using bounded model checking. The proof strongly relies on the definition of the invariants. We first present these invariants, then describe the proof methodology and results, and finally discuss the limitations of the proof through the analysis of a concrete counterexample.

4.1 Expressing the Invariant

We define the invariant $\mathcal{I}$ as the conjunction of properties $\mathcal{I}_1, \ldots, \mathcal{I}_5$ (listed below) and show that they are indeed maintained after executing any step. They describe the intrinsic properties of the two caches, mostly regarding their LRU management policy.

- $\mathcal{I}_1$ constrains the numbers of recorded lines Fi and Di in caches $\mathcal{F}$ and $\mathcal{D}$ to be in $[0, N]$, where N denotes the number of entries (lines) in a cache.
- $\mathcal{I}_2$ states that each non-recorded cache line (i.e. for index $i \geq Fi$ in $\mathcal{F}$ and for index $i \geq Di$ in $\mathcal{D}$) has default (zero) values.
- $\mathcal{I}_3$ states that each recorded cache line (i.e. for index $i < Fi$ in $\mathcal{F}$ and for index $i < Di$ in $\mathcal{D}$) has non-null tag address and tag register.
- $\mathcal{I}_4$ states that in each cache, each recorded cache line has a unique tag. In other words, two recoded lines in the same cache cannot share the same tag.
- $\mathcal{I}_5$ states that each recorded cache line has a gadget counter in $[1, G]$.

In the model, these properties are expressed as CBMC assumptions when we need to assume the invariant as a hypothesis, and as assertions when we need to verify it. We already illustrated CBMC assumptions in Sect. 2. Stating the same properties as assertions (with **`assert`** instructions) is straightforward. We refer the reader to the complete model in [8] for detailed definitions.

4.2 Proof with CBMC

Main Theorem. To formally verify the capacity of the algorithm to detect attacks, we use CBMC [12], a well-known bounded model-checker for C programs. We apply it on the model of the heatseeker algorithm and the considered JOP attacks written in C, described, respectively, in Sects. 3 and 2.2.

The main theorem we seek to prove is the following:

$\mathcal{P}_{\text{main}}$: Consider the heatseeker algorithm implemented with N cache lines, and consider JOP attack sequences in which $NumOp > 0$ denotes the number of steps, $D > 0$ a maximal dispatcher start-end distance, $G > 0$ the number of different gadgets, and set $MaxR = \lfloor N/2 \rfloor$. For any execution containing a JOP attack sequence with the considered constraints, the heatseeker algorithm effectively detects an attack by reporting an alarm.

Our goal is to prove this theorem in relevant configurations, that is, for relevant values of parameters.

Proof Steps. A bounded model-checker cannot directly prove properties for executions of any length, but it is able to deal with bounded executions, with a (relatively small) finite number of steps. It is also able to deal with symbolic states. This explains the methodology we apply in this proof, which relies on three proof harnesses, each corresponding to one of the following properties.

$\mathcal{P}_1$: The initial state satisfies $\mathcal{I}$. This is checked by the first harness, which asserts $\mathcal{I}$ directly after initialization.

$\mathcal{P}_2$: If the algorithm starts in a state satisfying $\mathcal{I}$ and proceeds any single jump (by executing the jump-handling function), the resulting state still satisfies $\mathcal{I}$. In other words, $\mathcal{I}$ is indeed an invariant. This is checked by the second harness, which assumes $\mathcal{I}$, executes one symbolic step, and asserts $\mathcal{I}$.

$\mathcal{P}_3$: Starting from any state satisfying $\mathcal{I}$, the execution of an attack sequence of $NumOp$ steps always triggers an alarm. This is checked by the third harness, which assumes $\mathcal{I}$, constrains the symbolic attack sequence using the attack model, iterates the jump-handling function, and asserts detection.

Notice that the proof of these properties only requires the tool to consider execution fragments with a finite number of steps: a direct implication with no steps for $\mathcal{P}_1$, an execution with only one step for $\mathcal{P}_2$, and an execution of at most $NumOp$ steps for $\mathcal{P}_3$.

The proof of $\mathcal{P}_{\text{main}}$ follows from properties $\mathcal{P}_1$–$\mathcal{P}_3$. Indeed, by $\mathcal{P}_1$ and $\mathcal{P}_2$, when starting from the initial state, we always remain in a state respecting $\mathcal{I}$. By $\mathcal{P}_3$, we always detect an attack starting in a state respecting $\mathcal{I}$. Therefore, we are able to detect an attack sequence starting at any moment.

Proven Configurations. We were able to prove each of these properties for configurations $(N, MaxR, NumOp) \in \{(3,1,5),(4,2,9),(5,2,11),(6,3,15)\}$. CBMC terminated without finding any counterexample to $\mathcal{P}_1$–$\mathcal{P}_3$ on our model. Hence,

N	$MaxR$	$NumOp$	Time (s)
3	1	5	1.002
4	2	9	3.543
5	2	11	8.214
6	3	15	32.216

Fig. 7: CBMC verification times for different parameter configurations.

every exploit that belongs to our definition of the attacks, including attacks with up to $MaxR$ registers used by functional gadgets, will be detected by the algorithm.

$N = 3$ allows detection of the JOP pattern described in Sect. 2.1 with $MaxR = 1$. Increasing N increases $MaxR$ and allows detection of more complex and less likely JOP attacks. The value $N = 5$ does not increase $MaxR$ in comparison to $N = 4$ but we provide it for more complete statistics. The value of $NumOp$[7] indicated in the triple is the lowest value for which $\mathcal{P}_{\text{main}}$ is proven for the provided $(N, MaxR)$. Parameters $G = 2$ and $D = 64$ are fixed. They are defined from previous experiments [7]. $G = 2$ allows detection with at least 2 gadgets: it is the optimal value as it enables the fastest detection of attacks. $D = 64$ is empirically defined from the analysis of the distribution of dispatcher gadgets in RISC-V binaries and could be adapted for other architectures: its precise value has no impact on the proof results.

Proof Statistics. For each of the considered triples $(N, MaxR, NumOp)$, the proof terminates within a few seconds on a desktop computer running Ubuntu 22.04.5 LTS, with an Intel® Core™ i5-1145G7 CPU @ 2.60 GHz, featuring 4 cores 8 threads, with 32 GB RAM. Figure 7 summarizes the required time to prove $\mathcal{P}_1$–$\mathcal{P}_3$ with different parameter configurations.

We estimate the overall effort to realize this verification case study as 4 person-months, out of which 2 person-months were spent to elaborate a C model and 2 person-months to debug the proof and analyze generated counterexamples.

4.3 Proof Limitations

We were able to prove the correctness of the algorithm under well-defined constraints on the attack model. While these constraints simplify some aspects of a real-world JOP exploit, they still allow us to obtain meaningful proof results. In particular, we assumed that any N consecutive functional gadgets must have pairwise different source addresses, and symmetrically that any N consecutive dispatcher invocations must target pairwise different destination addresses. In the C model, these assumptions correspond to the inner loops in the functional

[7] The value $NumOp$ increases with N because by design of the algorithm, with a greater number of cache lines, attack detection may require more steps depending on the initial state of the caches.

gadgets' addresses function (lines 191–196 in Fig. 3) and in the dispatcher gadget addresses function (lines 225–230 in Fig. 4). They enforce, respectively, the uniqueness of functional-gadget end addresses and the uniqueness of functional-gadget start addresses selected by the dispatcher.

Interestingly, without these constraints, during the attempt to prove $\mathcal{P}_3$, CBMC constructs counterexamples. We analyzed one counterexample and found that it uses repetitive functional gadget addresses. In a realistic JOP exploit, these additional constraints must be satisfied, otherwise the exploit is trivially useless. Therefore these additional assumptions do not diminish the ability to detect real-world JOP attacks. In this way, CBMC helped us to better understand the limitations and corner cases of the algorithm. By adopting this slightly more conservative model, we prove that the algorithm detects attacks regardless of its internal state, laying a solid foundation for a hardware implementation.

5 Related Work and Conclusion

Related Work. The heatseeker algorithm is an alternative to previous heuristic detection algorithm of code-reuse attacks. JOP-alarm [46], which detects JOP attacks based on indirect jumps frequency, provides no formalization of attacks. On the other hand, SCRAP [26] expressed code-reuse attacks in formal language, but did not go all the way to the formal verification of the detection technique based on dynamic gadget identification. Formal verification is rarely applied to control-flow security mechanisms, though recent work has verified defenses against speculative attacks [4], proved memory isolation guarantees for capability hardware [21] and established side-channel security contracts for open-source processors [45,42]. In the embedded domain, VRASED achieved the first formally verified hardware/software co-design of a security service [34]. Other threat models such as fault injections have similarly benefited from formal methods [44,31]. To the best of our knowledge, formal verification of JOP detection mechanisms was not performed in previous work.

More generally, this work is related to other verification case studies on real-life code and empirical evaluations of verification tools [24]. Among other examples, the KeY tool was used for verification of several libraries and applications in Java [16,5]. Verification of a traffic tunnel control system [35] was realized with VerCors [3]. Verification for a real-world avionics example [19] and for security properties [17] provided useful feedback on using Frama-C. SPARK was used in the verification of a TCP Stack [13] and complex datastructures [20]. Verification of the Hyper-V hypervisor with VCC [29] highlighted some issues specific to hypervisor verification. Deductive verification of smart contracts [11] was realized with Dafny [30]. Several case studies [36] were performed using VeriFast [25]. CBMC [12] was used in other industrial verification projects [28,15,39,27]. Each new case study contributes to improve verification tools by pointing out their limitations and by increasing the number of formally verified algorithms and products.

Conclusion and Future Work. We presented a formal verification of the heat-seeker algorithm, a signature-based detection mechanism for JOP attacks that monitors indirect control-flow edges to identify the repeated use of a dispatcher gadget. By formalizing both the algorithm and the targeted class of JOP attacks in a C model, we were able to apply the CBMC bounded model checker to establish that any attack sequence conforming to our formalization is detected by the algorithm. These results were verified for several configurations of the model. Yet, the approach inherits the limitations of bounded model checking, notably the exponential complexity as parameter values are increased. This work contributes to the design of defense mechanisms with strong security guarantees and illustrates how software verification techniques can be applied to models of hardware implementations. Future work includes extending the formal model to cover more complex JOP attacks, the construction of general models that enable the detection of larger CRA variants, and experimenting with other verification tools (in particular, from the SV-COMP competition).

Acknowledgment. Part of this work was supported by ANR (grants ANR-22-CE39-0014, ANR-22-CE25-0018, ANR-24-ASM2-0001). We thank the anonymous referees for helpful comments.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-Flow Integrity Principles, Implementations, and Applications. ACM Transactions on Information and System Security **13**(1) (2009). https://doi.org/10.1145/1609956.1609960
2. Ahmed, S., Xiao, Y., Snow, K.Z., Tan, G., Monrose, F., Yao, D.D.: Methodologies for Quantifying (Re-)randomization Security and Timing under JIT-ROP. In: Proc. of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS 2020). p. 1803–1820. ACM (2020). https://doi.org/10.1145/3372297.3417248
3. Armborst, L., Bos, P., van den Haak, L.B., Huisman, M., Rubbens, R., Sakar, Ö., Tasche, P.: The VerCors verifier: A progress report. In: Proc. of the 36th International Conference on Computer Aided Verification (CAV 2024). LNCS, vol. 14682, pp. 3–18. Springer (2024). https://doi.org/10.1007/978-3-031-65630-9_1
4. Baumann, J., Kim, Y., Farba, Y., Hriţcu, C., Leatherman-Brooks, J.: Triosecuris: Formally Verified Protection Against Speculative Control-Flow Hijacking. In: Proc. of the 39th IEEE Computer Security Foundations Symposium (CSF 2026). pp. 544–559. IEEE Computer Society (Jul 2026). https://doi.org/10.1109/CSF68417.2026.00036
5. Beckert, B., Sanders, P., Ulbrich, M., Wiesler, J., Witt, S.: Formally Verifying an Efficient Sorter. In: Proc. of the 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2024). LNCS, vol. 14570, pp. 268–287. Springer (2024). https://doi.org/10.1007/978-3-031-57246-3_15
6. Binosi, L., Barzasi, G., Carminati, M., Zanero, S., Polino, M.: The Illusion of Randomness: An Empirical Analysis of address Space Layout Randomization

Implementations. In: Proc. of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS 2024). p. 1360–1374. ACM (2024). https://doi.org/10.1145/3658644.3690239
7. Biton, T., Gilles, O., Gracia Pérez, D., Kosmatov, N., Pillement, S.: Heatseeker: Uncovering Dispatcher Gadgets on the Fly. ACM Trans. Archit. Code Optim. (2026), to appear.
8. Biton, T., Kosmatov, N., Gilles, O., Gracia Pérez, D., Pillement, S.: Formal verification of the heatseeker technique for detection of code-reuse attacks. Companion artifact. (2026). https://doi.org/10.5281/zenodo.19794964
9. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: Proc. of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2011). p. 30–40. ACM (2011). https://doi.org/10.1145/1966913.1966919
10. Brizendine, B., Babcock, A.: Pre-built JOP chains with the JOP ROCKET: bypassing DEP without ROP. Black Hat Asia (2021)
11. Cassez, F., Fuller, J., Quiles, H.M.A.: Deductive verification of smart contracts with Dafny. In: Proc. of the 27th International Conference on Formal Methods for Industrial Critical Systems (FMICS 2022). LNCS, vol. 13487, pp. 50–66. Springer (2022). https://doi.org/10.1007/978-3-031-15008-1_5
12. Clarke, E., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004). pp. 168–176. Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15
13. Cluzel, G., Georgiou, K., Moy, Y., Zeller, C.: Layered formal verification of a TCP stack. In: Proc. of the IEEE Secure Development Conference (SecDev 2021). pp. 86–93. IEEE (2021). https://doi.org/10.1109/SecDev51306.2021.00028
14. Common Criteria for Information Technology Security Evaluation: Part 3: Security assurance components, https://www.commoncriteriaportal.org/files/ccfiles/CC2022PART3R1.pdf
15. Cook, B., Khazem, K., Kroening, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Model checking boot code from AWS data centers. Formal Methods in System Design **57**(1), 34–52 (2021). https://doi.org/10.1007/s10703-020-00344-2
16. de Boer, M., de Gouw, S., Klamroth, J., Jung, C., Ulbrich, M., Weigl, A.: Formal specification and verification of JDK's identity hash map implementation. Formal Aspects Comput. **35**(3), 18:1–18:26 (2023). https://doi.org/10.1145/3594729
17. Djoudi, A., Hána, M., Kosmatov, N.: Formal Verification of a JavaCard Virtual Machine with Frama-C. In: Proc. of the 24th International Symposium on Formal Methods (FM 2021). LNCS, vol. 13047, pp. 427–444. Springer (2021). https://doi.org/10.1007/978-3-030-90870-6_23
18. Djoudi, A., Hána, M., Kosmatov, N., Kříženecký, M., Ohayon, F., Mouy, P., Fontaine, A., Féliot, D.: A Bottom-Up Formal Verification Approach for Common Criteria Certification: Application to JavaCard Virtual Machine. In: Proc. of the 11th European Congress on Embedded Real-Time Systems (ERTS 2022) (2022), https://hal.science/hal-03695829
19. Dordowsky, F.: An experimental study using ACSL and Frama-C to formulate and verify low-level requirements from a DO-178C compliant avionics project. Electronic Proc. in Theoretical Computer Science **187**, 28–41 (2015). https://doi.org/10.4204/EPTCS.187.3

20. Dross, C., Moy, Y.: Auto-active proof of red-black trees in SPARK. In: Proc. of the 9th International Symposium on NASA Formal Methods (NFM 2017). LNCS, vol. 10227, pp. 68–83 (2017). https://doi.org/10.1007/978-3-319-57288-8_5
21. Duque Antón, A.L., Müller, J., Schmitz, P., Jauch, T., Wezel, A., Deutschmann, L., Rahmani Fadiheh, M., Stoffel, D., Kunz, W.: VeriCHERI: Exhaustive Formal Security Verification of CHERI at the RTL. In: Proc. of the 43rd IEEE/ACM International Conference on Computer-Aided Design. ICCAD 2024, ACM (2025). https://doi.org/10.1145/3676536.3676841
22. El-Zoghby, A.M., Azer, M.A.: Survey of code reuse attacks and comparison of mitigation techniques. In: Proc. of the 9th International Conference on Software and Information Engineering. pp. 88–96. ICSIE 2020, ACM (2020). https://doi.org/10.1145/3436829.3436865
23. Fu, A., Ding, W., Kuang, B., Li, Q., Susilo, W., Zhang, Y.: FH-CFI: Fine-grained hardware-assisted control flow integrity for ARM-based IoT devices. Comput. Secur. **116**(C) (2022). https://doi.org/10.1016/j.cose.2022.102666
24. Hähnle, R., Huisman, M.: Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools. In: Computing and Software Science – State of the Art and Perspectives, LNCS, vol. 10000, pp. 345–373. Springer (2019). https://doi.org/10.1007/978-3-319-91908-9_18
25. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Proc. of the Third International Symposium on NASA Formal Methods (NFM 2011). LNCS, vol. 6617, pp. 41–55. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_4
26. Kayaalp, M., Schmitt, T., Nomani, J., Ponomarev, D., Abu-Ghazaleh, N.: SCRAP: Architecture for signature-based protection from Code Reuse Attacks. In: 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA 2013). pp. 258–269 (2013). https://doi.org/10.1109/HPCA.2013.6522324
27. Kosmatov, N., Longuet, D., Soulat, R.: Formal Verification of an Industrial Distributed Algorithm: an Experience Report. In: Proc. of the 9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. (ISOLA 2020). LNCS, vol. 12476, pp. 525–542. Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_30
28. Lee, D.A., Yoo, J., Lee, J.S.: A systematic verification of behavioral consistency between FBD design and ANSI-C implementation using HW-CBMC. Reliability Engineering & System Safety **120**, 139–149 (2013). https://doi.org/10.1016/j.ress.2013.06.006
29. Leinenbach, D., Santen, T.: Verifying the microsoft Hyper-V hypervisor with VCC. In: Proc. of the Second World Congres on Formal Methods (FM 2009). LNCS, vol. 5850, pp. 806–809. Springer (2009). https://doi.org/10.1007/978-3-642-05089-3_51
30. Leino, K.R.M.: Program Proofs. The MIT Press (2023)
31. Martin, T., Kosmatov, N., Prevosto, V.: Verifying Redundant-Check Based Countermeasures: A Case Study. In: Proc. of the 37th Annual ACM/SIGAPP Symposium on Applied Computing, Software Verification and Testing Track (SAC-SVT 2022). pp. 1849–1852. ACM (2022). https://doi.org/10.1145/3477314.3507341

32. Microsoft: Microsoft security toolkit delivers new bluehat prize defensive technology. https://news.microsoft.com/2012/07/25/microsoft-security-toolkit-delivers-new-bluehat-prize-defensive-technology/ accessed April, 2024 (2012)
33. Noman, H.A., Abu-Sharkh, O.M.: Code injection attacks in wireless-based Internet of Things (IoT): A comprehensive review and practical implementations. Sensors **23**(13), 6067 (2023). https://doi.org/10.3390/s23136067
34. Nunes, I.D.O., Eldefrawy, K., Rattanavipanon, N., Steiner, M., Tsudik, G.: VRASED: a verified hardware/software co-design for remote attestation. In: Proc. of the 28th USENIX Conference on Security Symposium (SEC 2019). p. 1429–1446. USENIX Association (2019), https://www.usenix.org/system/files/sec19-nunes.pdf
35. Oortwijn, W., Huisman, M.: Formal verification of an industrial safety-critical traffic tunnel control system. In: Proc. of the 15th International Conference on Integrated Formal Methods (IFM 2019). LNCS, vol. 11918, pp. 418–436. Springer (2019). https://doi.org/10.1007/978-3-030-34968-4_23
36. Philippaerts, P., Mühlberg, J., Penninckx, W., Smans, J., Jacobs, B., Piessens, F.: Software verification with VeriFast: Industrial case studies. Sci. Comput. Program. **82**, 77–97 (2014). https://doi.org/10.1016/J.SCICO.2013.01.006
37. Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-Oriented Programming: Systems, Languages, and Applications. ACM Trans. Inf. Syst. Secur. **15**(1) (2012). https://doi.org/10.1145/2133375.2133377
38. Sadeghi, A., Aminmansour, F., Shahriari, H.: Dwarf Frankenstein is still in your memory: tiny code reuse attacks. The ISC International Journal of Information Security **9**(1), 53–72 (2017). https://doi.org/10.22042/isecure.2017.0.0.4
39. Schrammel, P., Kroening, D., Brain, M., Martins, R., Teige, T., Bienmüller, T.: Incremental bounded model checking for embedded software. Formal Aspects Comput. **29**(5), 911–931 (2017). https://doi.org/10.1007/S00165-017-0419-1
40. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Proc. of the 14th ACM Conference on Computer and Communications Security (CCS 2007). p. 552–561. ACM (2007). https://doi.org/10.1145/1315245.1315313
41. Spang, C., Lavan, Y., Hartmann, M., Meisel, F., Koch, A.: DExIE - An IoT-Class Hardware Monitor for Real-Time Fine-Grained Control-Flow Integrity. In: Workshop on Design and Architectures for Signal and Image Processing, 14th Edition (DASIP 2021). p. 26–34. ACM (2021). https://doi.org/10.1145/3441110.3441146
42. Tan, Q., Yang, Y., Bourgeat, T., Malik, S., Yan, M.: RTL Verification for Secure Speculation Using Contract Shadow Logic. In: Proc. of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS 2025). p. 970–986. ACM (2025). https://doi.org/10.1145/3669940.3707243
43. Team, P.: PaX address space layout randomization (ASLR). http://pax. grsecurity. net/docs/aslr. txt (2003)
44. Tollec, S., Asavoae, M., Couroussé, D., Heydemann, K., Jan, M.: µARCHIFI: Formal Modeling and Verification Strategies for Microarchitectural Fault Injections. In: Formal Methods in Computer-Aided Design (FMCAD 2023). pp. 101–109 (2023). https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_18

45. Wang, Z., Mohr, G., von Gleissenthall, K., Reineke, J., Guarnieri, M.: Specification and Verification of Side-channel Security for Open-source Processors via Leakage Contracts. In: Proc. of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS 2023). p. 2128–2142. ACM (2023). https://doi.org/10.1145/3576915.3623192
46. Yao, F., Chen, J., Venkataramani, G.: JOP-alarm: Detecting jump-oriented programming-based anomalies in applications. In: Proc. of the 31st IEEE International Conference on Computer Design (ICCD 2013). pp. 467–470 (2013). https://doi.org/10.1109/ICCD.2013.6657084
47. Zaruba, F., Benini, L.: The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **27**(11), 2629–2640 (2019). https://doi.org/10.1109/TVLSI.2019.2926114