HEAPSTER: Analyzing the Security of Dynamic Allocators for Monolithic Firmware Images

Fabio Gritti†, Fabio Pagani†, Ilya Grishchenko†, Lukas Dresel†, Nilo Redini‡⊛, Christopher Kruegel†, Giovanni Vigna†
† University of California, Santa Barbara, ‡ Qualcomm Technologies Inc.
{degrigis, pagani, grishchenko, lukasdresel, chris, vigna}@ucsb.edu, nredini@qti.qualcomm.com

***Abstract*—Dynamic memory allocators are critical components of modern systems, and developers strive to find a balance between their performance and their security. Unfortunately, vulnerable allocators are routinely abused as building blocks in complex exploitation chains. Most of the research regarding memory allocators focuses on popular and standardized heap libraries, generally used by high-end devices such as desktop systems and servers. However, dynamic memory allocators are also extensively used in embedded systems but they have not received much scrutiny from the security community.**

In embedded systems, a raw firmware image is often the only available piece of information, and finding heap vulnerabilities is a manual and tedious process. First of all, recognizing a memory allocator library among thousands of stripped firmware functions can quickly become a daunting task. Moreover, emulating firmware functions to test for heap vulnerabilities comes with its own set of challenges, related, but not limited, to the re-hosting problem.

To fill this gap, in this paper we present HEAPSTER, a system that automatically identifies the heap library used by a *monolithic firmware image*, and tests its security with symbolic execution and bounded model checking. We evaluate HEAPSTER on a dataset of *20* synthetic monolithic firmware images — used as ground truth for our analyses — and also on a dataset of *799* monolithic firmware images collected in the wild and used in real-world devices. Across these datasets, our tool identified 11 different heap management library (HML) families containing a total of 48 different variations. The security testing performed by HEAPSTER found that *all* the identified variants are vulnerable to at least *one* critical heap vulnerability. The results presented in this paper show a clear pattern of poor security standards, and raise some concerns over the security of dynamic memory allocators employed by IoT devices.

***Index Terms*—Computer Security, Firmware Analysis, Vulnerability Research.**

⊛ The author contributed before joining Qualcomm Technologies Inc.

I. INTRODUCTION

Dynamic memory allocators are procedures responsible for reserving appropriately sized chunks of memory whenever a running program requires them. The number of chunks or, the amount of memory a program needs depends on a multitude of factors that are usually unknown at compilation time. For example, a browser needs to allocate some memory when the user opens a web page, and in general, any non-trivial program can make thousands of memory requests during its execution. Dynamic allocators are ubiquitous in modern systems, and they are present on different devices, ranging from Internet of Things (IoT) devices to high-performance servers.

The development of memory management libraries must take into consideration different aspects. On the one hand, dynamic allocators need to be designed with performance in mind; on the other hand, they need to be secure to avoid increasing the attack surface of a system. In fact, when maliciously manipulated, memory allocators can provide attackers with powerful exploitation primitives [53].

Software developers strive to find the optimal balance between security and performance, and, sometimes, they decide to trade the former for the latter. For example, security patches have been recently rejected – even if they were meant to remove *practical* attack vectors – to avoid any impact on the overall system's performance [17]. In another case, the introduction of new performance-tailored data structures in a heap library compromised its security, undermining years of security hardening [23]. Finally, some developers tend to favor performance because they consider heap protections as post-attack mitigations, and argue that the root cause for an attack should be addressed in the vulnerable application, rather than in the heap library itself [46]. While this is a possible point of view, we argue that heap libraries are a critical building block for many applications, and hence, should protect against cases where a simple programming mistake (i.e., a one-byte overflow) leads to a complete application compromise.

The balance between performance and security is an even more important issue for embedded systems, such as IoT devices. In fact, dynamic memory allocators for embedded systems usually need to operate with limited resources, sometimes under strict time constraints, and, in addition, might be deployed in industrial control systems, where their security becomes critical.

Unfortunately, evaluating dynamic memory allocators for embedded systems is challenging. While the previous statement is true for both Linux-based and *monolithic firmware images* (from now on, also referred to as *firmware blobs* or *blobs*), analyzing the latter is particularly cumbersome. First, the only information at hand when analyzing a monolithic firmware image extracted from a device is a raw binary (i.e., no source code or debugging symbols are available). Second, monolithic firmware images are not built on top of a traditional general-purpose OS, and the boundary between the application and the libraries' code is difficult to locate – if it exists at all. Finally, as different IoT devices might have different requirements, embedded developers might decide to implement custom allocators, rather than using popular and security-vetted heap implementations.

In the past few years, researchers from both industry and academia have proposed different approaches to assess the security of heap implementations. For example, Eckert et al. presented HeapHopper [24], a system that leverages symbolic execution and Bounded Model Checking (BMC) to test the security of a dynamic allocator. In a similar vein, Insu et al. [63] used fuzzing techniques to explore the attack surface of different heap library implementations, eventually detecting novel security violations. Researchers have also proposed multiple patches and refactoring of existing heap libraries [27], and re-engineered the heap management approaches, introducing strong security foundations [6], [39], [45], [48], [55], [56].

However, all the aforementioned approaches have been tailored to heap implementations used for "classic" systems, while little has been done to study the security of dynamic allocators used in embedded systems. Traditionally, the use of dynamic memory allocators in embedded systems has been considered a bad

practice, only to be limited to those cases where an allocator was absolutely necessary [12], [25]. However, advancements in Real-time Operating System's (RTOS) technologies and the increase of on-board computing resources, are driving embedded developers to leverage dynamic allocators more often than before, potentially exposing IoT devices to more complex types of exploits.

To shed some light on this issue, we developed HEAPSTER, a framework to recognize, categorize, and test the security of the Heap Management Library (from now on referred to as *HML*) used by a monolithic firmware image. HEAPSTER identifies the functions that are part of the dynamic allocator interface (i.e., `malloc` and `free`) by studying how pointers are generated (and used) inside a firmware blob, and by dynamically executing these functions to monitor their run-time behavior. Then, HEAPSTER identifies the prototypes of both the allocator and the de-allocator, and understands how to initialize and call these functions with appropriate arguments. Finally, HEAPSTER performs a bounded model checking analysis to check for the presence of common classes of heap vulnerabilities, and, when one is found, it generates a proof-of-vulnerability (PoV) that can be used to reproduce the security violation. To perform this task, we leverage a custom version of HeapHopper [24], which we adapted to support the analysis of monolithic firmware images. To evaluate the techniques we present in this paper, we use two different datasets. The first dataset contains 20 different monolithic firmware images collected from previous work [21], [29], [33] for which debugging symbols are available. We use this dataset (*ground-truth*) to evaluate HEAPSTER performance, and easily confirm its results employing the debug symbols of the firmware images. The second dataset contains 799 blobs collected from the wild (*wild* dataset). Given its significant heterogeneity in representing a multitude of different IoT devices [52], [59], we use these firmware samples to gain more insight into the state of memory allocators in the embedded world. Our results show that (1), across all the firmware samples analyzed by HEAPSTER there are *11* different HML families combining a total of *48* different variations (i.e., versions, customizations, or, simple configuration differences), and (2), for *all* the HMLs it is possible to generate a PoV for *at least one* class of heap vulnerability. This result highlights how the security of the heap management for embedded systems is far behind the standards expected from modern allocators.

In summary, we make the following contributions:

- We advance the state of program analysis over monolithic firmware images by proposing different techniques to reason about pointer creation and firmware function emulation, and we scale them to test a total of *819* firmware blobs. While these techniques are used to recover the HML from a blob, they can be leveraged for other analyses and applications.
- We present HEAPSTER, the first system able to automatically identify a heap management library inside a monolithic firmware image, and test its security.
- We implement the heap security testing by modifying HeapHopper to apply symbolic execution and BMC techniques over target functions in monolithic firmware images.
- We demonstrate a general pattern of poor security standards in the HMLs recovered from a heterogeneous set of firmware blobs. Specifically, in *all* of the 48 identified library variants, we find at least one critical heap vulnerability class.

All the artifacts are made available at `github.com/ucsb-seclab/heapster`.

II. CHALLENGES AND GOALS

In this section, we discuss the main challenges that we must overcome to successfully test the security of the heap management library of a monolithic firmware image.

Firmware Loading. Monolithic firmware images can have custom formats (e.g., a custom base address, a custom entry point, and a custom memory layout), and are often built for the specific hardware configuration of a given device. Moreover, when the instruction set architecture (ISA) of a firmware sample is not supported by the analysis tool, even basic emulation tasks can become challenging [35]. To simplify the problem, in this paper we focus on the ARM `CortexM` architecture, the most popular solution for consumer IoT devices [29], [37]. Therefore, we assume the required firmware metadata to be either architecturally known (e.g., the memory layout, including the boundaries of the heap memory region and the MMIO region [7]), or recoverable with state-of-the-art techniques [59].

Once the target firmware image is loaded in memory we need to identify the boundaries of the functions defined in it. This problem, exacerbated by the intrinsic nature of a monolithic firmware image, is well-known in literature [3], [11], [36], [38], [41], [58], and we consider it orthogonal to our research. Therefore, in this paper, we simply rely on state-of-the-art tools to retrieve this information.

HML Identification. Monolithic firmware images can contain hundreds of different functions, belonging either to the main application or to the supporting libraries (e.g., hardware abstraction libraries, or RTOS libraries). In this scenario, manually identifying the functions related to the heap management is a daunting task, and automated approaches become necessary. While previous work on automated identification of allocator code exists [19], it focuses on "classic" systems (e.g., personal computers and servers) and it relies on the dynamic execution of the *entire program* under analysis, which is particularly challenging — if possible at all — when dealing with monolithic firmware. The usage of dynamic allocators in embedded devices has been the center of multiple discussions through the years [12], [25]. In particular, the main concerns raised by embedded developers are related to memory fragmentation and non-determinism of the firmware's code. In fact, when numerous cycles of memory allocation and de-allocation are performed without rebooting the system (a very common situation for embedded devices), the memory can become heavily fragmented and the allocator might be unable to service further allocation requests. Overcoming this situation requires implementing a "graceful degrade" mechanism, which developers argue to be extremely challenging when an HML is being used as part of the application code. For these reasons, the usage of heap libraries has often been discouraged for real-time embedded systems, in particular when deployed in safety-critical environments. Despite this, HMLs are frequently used by embedded systems developers, as we will show in Section V.

HML Identification Scope. In this paper, we focus on the identification of low-level dynamic memory allocators that (1)

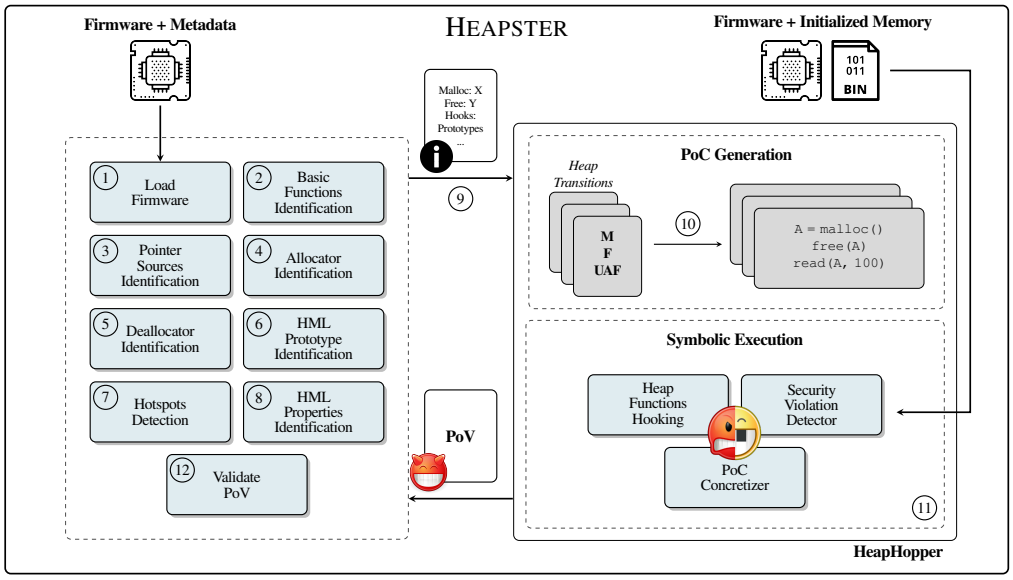

Fig. 1: HEAPSTER Overview. The analysis pipeline follows the order of the circled numbers.

have at least two primitives (one to allocate, and one to de-allocate chunks), (2) serve every request "on-the-fly" by calculating the next chunk to be returned at every invocation of the allocator, (3) implement a memory management strategy where chunks are reused for future allocations, (4) receive the requested allocation size by argument (allocator), and the requested chunk-to-free by argument (deallocator), and, (5), pass the allocated memory address to a caller as a return value inside a register.

Firmware Initialization. HML functions do not live in a vacuum, but instead use in-memory data structures and global variables to keep track of freed memory chunks or to determine the heap base address. Many of these heap-related constants and data structures are either unpacked during the boot process or defined by auxiliary functions that initialize the heap (known as *heap initializers*). Therefore, before our system can start testing the security of a heap library, it also needs to identify and execute any initializer function, to bring the firmware memory into a consistent and initialized state.

Firmware Re-hosting. Our system leverages dynamic execution of *targeted* firmware functions rather than leveraging full emulation of the firmware blob. However, these targeted functions are still executed inside an emulator rather than the hardware for which the firmware was designed. Therefore, multiple details related to peripherals models and the original execution environment are missing, and this can potentially hinder our analyses. This problem is known in the firmware community as *firmware re-hosting* [28], [60]. Even if recent efforts proposed different solutions [15], [22], [26], [42], [51], [65], monolithic firmware re-hosting remains a challenge when scaled on a heterogeneous set of blobs [60]. Since in this paper we focus on the emulation of a limited number of functions, we use carefully configured *execution models* (detailed in Appendix A) to resolve the common pitfalls related to a partial execution environment.

Symbolic Execution Scalability. Our system uses symbolic execution and bounded model checking to discover classes of heap vulnerabilities in the heap library of a firmware image. However, symbolic analysis is afflicted by scalability issues related to path explosion and related to the overhead of constraint solving [14]. Therefore, even testing a single `malloc` implementation with an unconstrained symbolic size argument might require an unrealistic amount of computational resources. For this reason, when performing the symbolic analysis, we define boundaries to contain the scope of the analysis.

III. APPROACH

In this section, we present HEAPSTER, our automated approach for identifying the HML used by a monolithic firmware image and for testing its security. An overview of the HEAPSTER architecture is illustrated in Figure 1.

Our system starts from a firmware blob along with related metadata, and uses this information to load the firmware in an emulator ①. Then, HEAPSTER identifies the *basic functions*: a set of functions that receive memory addresses as arguments, and perform simple memory operations over them (e.g., `memcpy`) ②. Once the *basic functions* are identified, HEAPSTER leverages an inter-procedural source-sink analysis to detect the *pointer sources*: functions whose return values flow inside an argument of a *basic function* ③. Intuitively, a *pointer source* is potentially "generating" a memory pointer used by a *basic function*. Then, HEAPSTER emulates the *pointer sources* one by one, and monitors their behavior to decide whether the function is a memory allocator or not ④. Given a list of potential allocators, HEAPSTER tries to find the correspondent de-allocator. Practically, HEAPSTER leverages syntactic features to first filter the functions in the firmware blob to a set of candidate de-allocator functions, and after that, it executes every allocator paired with every candidate de-allocator, in an attempt to observe chunks being re-used as explained in Section II ⑤. When a working allocator/de-allocator pair (i.e., the HML) is identified, HEAPSTER collects more information regarding their prototypes ⑥, and about their implementation ⑦-⑧. This information includes: the detection of any hotspot that would hinder the symbolic execution, the HML properties (such as the heap base address, the heap growing

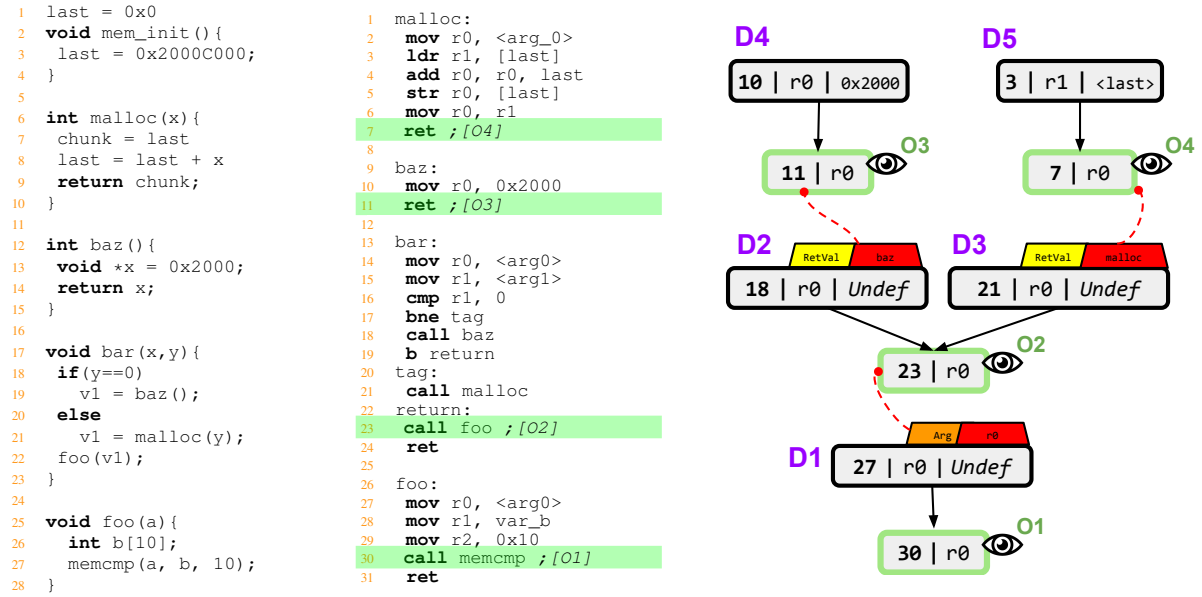

Fig. 2: Running example of the *pointer sources* identification analysis. On the left, the C code from which the assembly on the right is generated. We highlight the assembly lines where we start the backward slice. Ds represent register definitions, Os represent observation points.

direction), and finally, the presence of inline heap metadata [24] (i.e., control information data commonly used by HML to manage the memory chunks, and stored alongside user data).

To test the security of the HML, HEAPSTER provides the firmware image along with all the information collected in the previous steps ⑨ to a custom version of HeapHopper [24] modified to support the analysis of firmware blobs (we discuss the main differences with respect to the original work in Section III-I). In particular, HeapHopper automatically generates *Proof-of-Concept* programs (PoCs) that interact with the discovered HML ⑩ with benign transitions — i.e., `malloc` and `free` — and malicious ones – e.g., use-after-free (UAF). Then, by using bounded model checking and symbolic execution, HeapHopper traces the execution of the generated PoCs and produces a *Proof-of-Vulnerability* (PoV) when a security violation is detected ⑪. Finally, to filter any false positives, HEAPSTER validates the generated PoVs by re-executing them inside the emulator ⑫.

A. Firmware Loading

To correctly load the firmware image inside the emulator and start its analysis, HEAPSTER needs four pieces of information: (1) the base address, (2) the entry point, (3) the memory range covered by the dynamic memory (heap memory region), and (4) the memory range where peripherals are mapped (i.e., the MMIO range). To extract the first two pieces of information from a target firmware blob, we implemented and integrated into our system, the technique presented by Wen [59]. The memory ranges are instead standardized by the CPU architecture [7], and we therefore safely assume the heap and MMIO memory ranges to be mapped according to the official specification. Finally, we extract the initial value of the firmware's stack pointer by reading the first DWORD of the firmware image [7].

B. Firmware Functions Classification

Basic Functions Identification. To identify the *basic functions*, we use a technique similar to the one implemented in Sibyl [16], which is based on the idea that *basic functions* have a predictable behavior (e.g., *memcpy* accepts a source buffer, a size, and a destination buffer, then copies the former into the latter). For instance, by calling a function with a set of arguments that comply with the prototype of *memcpy*, i.e., `void *memcpy(void *dest, const void *src, size_t n)`, we expect that, when the function terminates, the memory at the address in `dest` contains exactly `n` bytes from the address in `src`, while the buffer at `src` is left unmodified. HEAPSTER tests every function contained in the firmware image and matches their behaviors against nine different models, including two variants of `memcpy` and `memset`, one for `memcmp`, and four string-related functions (namely, `strlen`, `strcat`, `strncat`, and `strncpy`). Whenever a match is found, we tag the function accordingly and save the inferred prototype.

Pointer Source Identification. To identify the *pointer sources*, HEAPSTER starts from the call sites of the *basic functions* and applies a Reaching Definitions (from now on referred to as RD) data-flow analysis to understand how the arguments of the *basic functions* are generated. When applied to a function, RD builds a directed graph where nodes are register definitions and edges represent definition dependencies. Every node in the definition graph contains the name of the register being defined and the code location of the definition. Additionally, certain nodes also contain the value that the register holds at the code location — when it can be statically determined — and, optionally, a *tag* that includes metadata regarding the current definition. In particular, we leverage the tags to understand whether the definition is inter-procedural: `RetVal` and `Arg`. The former indicates that the definition

originates from the return value of a function call, while the latter indicates that the definition is provided by the callers of the current function via an argument. With this information, given a location in a function and a register of interest (i.e., an *observation point*), HEAPSTER finds the corresponding node in the definition graph and computes the backward slice for the definition in question. When the exploration reaches a node with an inter-procedural tag, the analysis continues leveraging the information contained in the tag itself.

In particular, to handle nodes tagged as `Arg`, the analysis searches for every caller of the current function (F), builds their definition graphs, and resumes the backward slice from the definition of the target register argument at F's call site. To handle nodes tagged as `RetVal`, the analysis enters the called function, builds its definition graph, and resumes the backward slice from the return locations of the function, using the register holding the return value (i.e., `r0`) as the observation point. This exploration strategy provides our analysis an *inter-procedural* view of node dependencies and, more importantly, the ability to identify which functions are involved in a node's definition.

Figure 2 shows an example of the analysis when applied to the assembly code in the left part of the figure[1]. In this example, our analysis starts by setting an observation point (O1) at line 30 for register `r0` (i.e., the register that represents the first `memcmp`'s argument). By looking at the dependency graph of `foo`, HEAPSTER discovers that the definition of `r0` depends on D1. Since D1 is tagged as `Arg`, HEAPSTER generates the dependency graph for all the callers of `foo` (in this example only `bar` at Line 22 in the C code), and resumes the backward slice starting from all the call sites to `foo`. In our example, the computation resumes from observation point O2 (assembly line 23), targeting the register `r0`. At this point, the analysis discovers that the definition of `r0` is derived either from the return value of `malloc` (D3) or from the return value of `baz` (D2). As D2 and D3 are tagged as `RetVal`, the analysis recovers the callees (i.e., `baz` and `malloc`), generates their definition graphs, and resumes the backward slice at observation point O3 and O4, targeting the register `r0` (i.e., the return value of `baz` and `malloc`). On one side, HEAPSTER discovers that D4 has a constant value and terminates the exploration. On the other side, HEAPSTER detects that the definition of `r0` is coming from D5, which again has a constant value, and therefore, terminates the exploration.

During the computation of backward slice, we label every function whose return value is used to define a target register in the explored definitions chain as a *pointer source*. In our example, the return values of `baz` and `malloc` respectively define `r0` at line 18, and `r0` at line 21, therefore, both functions are labeled as *pointer sources*.

The termination of the analysis is guaranteed by the monotonicity property of reaching definitions when creating the dependency graphs. Moreover, we terminate any recursive behavior by not exploring the same portion of a dependency graph more than once across the entire analysis, since this would not add any new information regarding *pointer sources* discovery.

[1]For space constraints, the assembly code is simplified and the graphs only contain the nodes relevant to this example.

C. Allocator Identification

Not every identified *pointer source* is a memory allocator. For instance, in the running example described in Figure 2, the function `baz` is merely returning a constant value, but it is labeled as *pointer source* by the analysis. Similarly, functions that wrap `malloc` (such as `calloc` or `realloc`), are also be labeled as *pointer source*, despite not being in scope of the security testing. Therefore, we need to reason about the extracted *pointer sources* and their dynamic behavior to filter false positives. We label a *pointer source* as an allocator if, after executing the function several times, we observe different memory addresses being returned that are all contained within the heap memory region. The intuition behind this definition is that a dynamic memory allocator should serve every request with a different memory chunk within the heap region. In our example, this property would not be respected by `baz`, while it is true for `malloc`.

Pointer Sources Execution. To classify a *pointer source* as an allocator, we need to be able to call it with the proper arguments. To simplify the analysis at this step, we statically collect the arguments' values from the call sites of a *pointer source*. Based on the assumption that the correct `malloc` receives as an argument an integer value representing the requested size, we expect to extract *at least* one valid value across the call sites. If this is not true, we use a placeholder integer value for each argument.

Pointer Sources & Heap Initializers. As previously mentioned in Section II, a successful emulation of a *pointer source* can depend on the initial values of specific global variables. For instance, in the example in Figure 2, `malloc` uses a global variable named `last` (Line 7 in the C source code). However, the global variables are often unpacked during the bootstrap of the firmware on the hardware device, and, therefore, are not always statically available beforehand. Therefore, we need to initialize every global variable used by a *pointer source* function before its emulation. At this step, we focus on *heap global variables* as they are the ones necessary to successfully execute the *pointer source* representing `malloc`. In particular, heap global variables can be (1) unpacked by the firmware's entry point by compiler-injected stubs (i.e., loops that initialize specific memory regions), (2) written by custom auxiliary heap procedures of the employed heap library (i.e., simple procedures that write constant values at specific addresses), and, in some cases, even by a combination of both strategies. Additionally, sometimes the heap global variables initialization can be performed by the *pointer source* itself as these cases are trivially handled when emulating the *pointer source*. To solve this issue, we first dynamically execute the firmware's entry point (i.e., the `ResetHandler`). Then, if we still detect missing heap global variables during a *pointer source* execution, we select a set of functions as possible initializers using intuitive syntactic heuristics (e.g., they should accept zero arguments) and execute them one by one right before the execution of the *pointer source*. In case none of the selected functions initialize the heap global variables used by a target *pointer source*, we discard the *pointer source*. On the contrary, if the values returned by the *pointer source* respect the allocator definition, we save the memory state associated with the *heap initializer* execution ("Initialized Memory" at the top-right corner in Figure 1), and label the *pointer source* as an allocator. To execute the `ResetHandler` and *heap initializers* we

use the execution model described in Appendix A. In our example, the function `malloc` is labeled as an allocator after we execute its *heap initializer* `mem_init`.

D. De-allocator Identification

Given an identified allocator, HEAPSTER tries to find the corresponding de-allocator routine. To achieve this goal, HEAPSTER first statically identifies a set of possible de-allocator candidate functions by pre-filtering all the firmware blob's functions (e.g., by not considering the *pointer sources*, or the *basic functions*), which results in the *de-allocator candidate set*. Then, HEAPSTER leverages dynamic execution to analyze the behavior of these procedures when paired with the target allocator. When executing a de-allocator candidate, we assume that the heap has already been initialized during the identification of the correspondent allocator.

Deallocator Candidate Test. As we discussed in Section II, in this paper we target dynamic allocators that re-use freed memory chunks. With this insight in mind, given the allocator A, for any candidate de-allocator D in the *deallocator candidate set*, we perform the following test:

1) call A for X times and collect the return values `a1`, `a2`, ..., `aX`;
2) call D for X times passing `a1`, `a2`, ..., `aX` as arguments;
3) call A one more time and verify the new return value `aX+1` is equal to one of `a1`, `a2`, ..., `aX`.

Specifically, in our configuration we set X to a value of three[2]. When this test is successful, we consider the pair (A, D) to be the blob's HML.

Deallocator Candidate Execution. De-allocators consume values dynamically generated by `malloc`, and therefore, we cannot simply collect their arguments from the call sites as we do when executing *pointer sources*. However, as we are always testing a de-allocator candidate together with a target allocator, we set the de-allocator's arguments to the value previously returned by a call to the allocator. In our experiments, this is always enough to observe the de-allocation behavior without compromising the execution of the primitive.

HML Pairs Filtering. Wrappers around the real allocator and de-allocator (i.e., functions that merely call the HML primitives and forward their results) can still be mistaken for a valid HML. Since the target of our analysis is the low-level interface of the main allocator of the firmware blob, we need to further filter the identified (A, D) pairs to remove false positives. To do this, we analyze the callgraph of each allocator in an HML pair, and we recursively remove the ones that call other allocators in another HML pair. We repeat the same analysis for the de-allocators to eventually identify the final HML pair.

E. HML Analysis

Once HEAPSTER detects a valid HML pair for a firmware image (i.e., `malloc` and `free`), it needs to collect more information to support the HML's security testing. In this section, we discuss how HEAPSTER recovers the prototypes of the heap functions, how it identifies hotspots that might impact the symbolic exploration, and how it collects additional properties of the HML to configure the bounded model checking.

Prototype Recovery. To be able to support diverse implementations of `malloc` and `free`, we can not assume that the HML primitives always follow the standard prototypes (i.e., `malloc(size)`, and `free(*ptr)`). For instance, we observed firmware images in our dataset using an extra argument to store error codes in case an HML primitive happens to fail. In the previous analyses, when executing `malloc` and `free`, we were simply re-using a valid set of arguments collected at the respective call sites, without any additional information about their semantics (e.g., which argument is the requested size). However, HeapHopper's security testing requires semantic information for the arguments to create the programs (i.e., PoC) that interact with the identified HML. Therefore, in presence of multiple arguments, we need to understand which argument represents the size in `malloc`, and which one represents the pointer to be freed in `free`. To extract this information, we execute the primitive with symbolic arguments constrained to concrete values[3], and, then, we select the argument with the highest number of observable constraints. This technique is based on the intuition that the requested size and the pointer to be freed generally go through several different checks, and therefore, multiple constraints are set over the symbolic variables. For example, it is very common that the requested size value falls inside a specific integer range as it is used later to select an appropriately sized memory chunk.

Hotspot Detection. During the execution of HML functions – in particular when erroneous conditions are met – the code might call functions that put the firmware in a stalling state (e.g., an infinite loop or a sleeping procedure). These procedures are generally meant to protect the device from damage, and usually require an external intervention to be solved (i.e., a reboot). However, these behaviors are problematic when scaling symbolic execution because the analysis can be stuck analyzing redundant code, with little to no progress. To detect these "problematic" functions, we *profile* several runs of `malloc` and `free` by supplying both legitimate and invalid parameters (e.g., we set the `free` argument to an invalid pointer). Whenever the emulation reaches a configured timeout, we identify the sub-function in which the exploration spent the most time, and we mark it to be skipped when future executions reach its address. This process is repeated until `malloc` and `free` can be entirely emulated within the configured timeout. The list of problematic functions is saved, and used when performing the security testing to inform the symbolic engine to skip these functions.

However, skipping functions that may be responsible for heap state changes could interfere with the results of the security evaluation, and introduce false positives/negatives. During our evaluation, we confirmed that the functions marked as "problematic" always bring the firmware code in a stalling state and never return. This means that the code marked to be skipped does not change the heap state, reducing the possibility of false negatives. For this reason, we focus our efforts on removing false positives by re-executing the generated PoV programs *without* skipping any function.

HML Properties. To configure the symbolic testing and identify security violations in an HML, we need to extract some of its

[2]We chose X=3 to allow for different allocator chunk management policies. We did not observe any difference in de-allocators identification when using bigger values.

[3]These arguments are the ones found at the call sites, or valid placeholders.

implementation details. In particular: the heap base address, the heap growth direction (i.e., if it grows toward higher or lower memory addresses), and the size of inline metadata [24]. Extracting the first two proprieties is straightforward: since we can execute the `malloc` function, the base address corresponds to the first value returned by the allocator, while the growth direction is inferred by comparing the result of two subsequent `malloc` allocations. To detect the presence of inline metadata, we place memory breakpoints in the surrounding of a previously allocated chunk, and then we invoke `free` over it. Whenever one of these breakpoints is triggered during `free`'s execution, we tag the corresponding memory location as containing heap metadata.

F. HML Model Definition

The core component behind the security testing of the identified HML is a custom version of HeapHopper [24] that we adapted to test monolithic firmware images.

Heap Transitions. HeapHopper models the heap as a state machine where nodes represent heap states, and edges represent *transitions* between states. These *heap transitions* can be benign operations — i.e., calls to `malloc` (M) and `free` (F) — or malicious exploitation primitives. In particular, HeapHopper implements the following heap exploitation primitives:

Double-free (DF) Calling `free` on a memory chunk that was already been freed, and not re-allocated.

Fake-free (FF) Calling `free` with a memory address that points to an invalid or manipulated heap chunk.

Use-after-free (UAF) Writing data to a memory chunk that has been freed, and has not been re-allocated yet.

Heap overflow (O) Writing data beyond the size of a target chunk.

Heap Vulnerabilities. If the HML does not have defenses against the exploitation primitives, the data structures holding the state of the heap can contain erroneous information. This can lead to a *vulnerable* state, which, in turn, can serve as a focal point in an attack against the system. In this paper, we focus on the following heap vulnerabilities:

Overlapping Chunks (OC) Allocation of a memory chunk that overlaps with other allocated chunks.

Non-Heap Allocation (NHA) Allocation of a memory chunk outside the heap region.

Arbitrary Write (AW) A memory write where the attacker can control the destination address *and* the content.

Restricted Write (RW) A memory write with restricted capabilities over the destination address *or* the content being written.

G. HML Bounded Model Checking

HeapHopper Analysis. Given the heap modeling described in the previous section, we leverage HeapHopper to perform the security analysis of a heap library using bounded model checking and symbolic execution. HeapHopper generates proof of concept programs (PoCs) that contain permutations of different benign heap transitions (i.e., legitimate calls to `malloc` and `free`) and malicious transitions (i.e., the exploitation primitives), and while symbolically executing them, it checks if any state becomes vulnerable to one of the heap vulnerability classes defined in Section III-F. When a vulnerability is detected, all the symbolic values in memory are concretized to obtain a fully concrete *Proof-of-Vulnerability* (PoV), which we used later to confirm the finding. To keep the analysis tractable during exploration, HeapHopper bounds the symbolic values inside a PoC using specific domain knowledge (e.g., by limiting the `malloc` requested size parameter to a list of few integer values). We discuss the main HeapHopper configuration parameters in Appendix G.

HeapHopper Setup. To test the identified HML, we first load a PoC inside HeapHopper, and, then, we side-load the monolithic firmware image as the PoC's library. Finally, we hook any call to `malloc` and `free` executed by the PoC so that they are redirected to the respective HML functions identified in the firmware blob.

H. PoV Validation

As acknowledged by Eckert et al. [24], and more recently by Yun et al. [63], HeapHopper can produce false positives. In other words, a PoC might trigger a heap vulnerability during the symbolic tracing, which is not confirmed when re-executing the PoV. To mitigate this issue, HEAPSTER always checks that a PoV generated by HeapHopper actually triggers the heap vulnerability. To do this, we re-emulate the concrete PoV *without* skipping any identified "problematic" functions (as explained in Section III-E), and we check whether the execution effectively triggers the discovered heap vulnerability. Otherwise, we discard the PoV.

I. HeapHopper Modifications

In this work, we build upon the implementation of HeapHopper [24], in particular, we modified parts of it to support the analysis of monolithic firmware images based on ARM `CortexM` CPUs.

PoC Generation. We changed how HeapHopper generates the PoCs to: (1) generate ARM binaries, and (2) support the generation of programs with custom prototypes for `malloc` and `free` (as explained in Section III-E).

PoC Loading. We modified how HeapHopper loads the binaries under testing inside the underlying emulator. In particular, since a stand-alone binary HML is not available (e.g., in the original version of HeapHopper this binary was represented by the GNU libc [32]), we instruct HeapHopper to use the monolithic firmware image as a library, which is side-loaded with the PoC under testing inside the emulator.

Arbitrary Write Model. The original HeapHopper categorized attacker-controlled writes in Arbitrary Write and Arbitrary Write Constrained. A vulnerability is classified as an Arbitrary Write when we can control the destination address *and* the content to be written. Arbitrary Write Constrained refers to an attacker-controlled write that can be redirected only to memory locations where specific content is present. In our version, we decided to replace Arbitrary Write Constrained with Restricted Write. We use the Restricted Write to indicate attacker-controlled write with limited capabilities over the destination address *or* the content.

IV. IMPLEMENTATION

We implemented HEAPSTER in Python on top of the binary analysis framework `angr` [54].

Firmware Preparation. A mandatory requirement to analyze a monolithic firmware is to know both its entry point and base address. This information can be either provided as input to our

system, or extracted with the analysis proposed in FirmXRay [59]. To identify the functions in the firmware image, we extract the entry point from the Interrupt Vector Table stored in the blob [8], and we leverage `angr`'s recursive traversal disassembly to build its Control Flow Graph (CFG). After recovering the firmware's CFG, we also use `angr`'s *CallingConvention* analysis to determine the number of arguments and possible return values of every identified function.

Static Analyses. To implement the backward slice analysis presented in Section III-B, we leverage `angr`'s *ReachingDefinition* engine. This analysis enables HEAPSTER to apply the classic RD dataflow analysis on binaries and to reason about register definitions and, in a limited way, memory definitions.

Functions Emulation. To dynamically execute the functions of the firmware blob (symbolically and concretely), we leverage the VEX execution engine provided by the `angr` framework. In particular, the binary code is lifted into the VEX Intermediate Representation (IR) [44]. Later, the IR instructions are executed to mimic the effect of the real assembly instructions over the state of the program. The VEX engine supports both symbolic execution and fully concrete execution (i.e., no symbolic variables in the program's state). When using symbolic execution, we leverage a Depth-First-Search (DFS) exploration strategy. We discuss the different execution models that HEAPSTER uses to emulate the functions in the firmware image in Appendix A. In particular, these execution models have been implemented by combining together different `angr`'s *Exploration Techniques*, and by instrumenting memory access operations triggered during the functions' emulation.

V. EVALUATION

We run the first (small-scale) evaluation on a dataset created by consolidating 20 images from previous research on firmware rehosting and firmware security [22], [29], [33], [43]. We refer to this dataset as the *ground-truth* dataset because, for each firmware image, we are provided with the stripped binary image, and the ELF file with debug symbols. Consequently, since we have labels for every function in the firmware, we can confirm whether HEAPSTER is able to correctly locate the *basic functions* and the HML primitives.

Our second (large-scale) evaluation is instead conducted on a collection of monolithic firmware images shared by previous research [59] and extracted from popular fitness devices [52]. We refer to this collection as the *wild* dataset. This dataset comprises a total of 799 monolithic firmware images, none of which is accompanied by debug symbols. To study the distribution of the heap libraries used by monolithic firmware in the wild, and to avoid wasting resources to test the security of identical HML implementations, we leverage BinDiff [66] to calculate the similarity between the allocator functions and to cluster them. For each dataset, we split the evaluation into four parts: (1) HML identification in each firmware blob, (2) coarse-grained HML clustering, (3) fine-grained HML clustering, and (4) security analysis with the modified version of HeapHopper (discussed in Section III-I) for each fine-grained cluster.

HML Identification. We run the HML identification analysis, as discussed in Section III, on every firmware blob in both datasets with a time limit of 72 hours, and a memory limit of 70GB per firmware.

Coarse-grained Clustering. The goal of this analysis is the identification of allocators belonging to the same heap library implementation. For this clustering, we run the similarity analysis *only* considering the bodies of the functions identified as `malloc`. We consider two implementations part of the same cluster when BinDiff reports similarity and confidence scores $\geqslant 0.7$. We empirically chose the threshold of 0.7 with the help of our *ground-truth* dataset: this value resulted in the lowest number of misclassifications. Nevertheless, since BinDiff uses several heuristics for its analysis [31], we compensate for misclassifications with additional manual refinements. In particular, we look for clusters containing a single or few HMLs, since these cases can either represent scarcely used heap implementations or BinDiff imprecisions. When we detect a potential misclassification of this kind, we manually compare the outliers with other HMLs and include them inside the correct cluster. Finally, we look for situations where two clusters are connected by a few HMLs with a low similarity score, and we manually adjust the clusters after confirming that they represent two different heap libraries. In total, we had to manually correct the classification of only 18 blobs.

Fine-grained Clustering. The goal of this analysis is the identification of *identical* HMLs. In particular, since identical HML implementations are affected by the same vulnerabilities, this clustering reduces the number of security evaluations performed with HeapHopper. The fine-grained clustering is based on the similarity analysis over the body of the `malloc`'s function and additionally all of its *callees* up to a depth level of two[4]. Whenever the similarity score reported from BinDiff is equal to 1.0 for *all* of these functions, we reported the two HMLs to be in the same fine-grained cluster. Intuitively, the fine-grained clustering defines sub-clusters inside the coarse-grained clusters identified in the previous step. Throughout the rest of the evaluation, we use the notation A_N to indicate an HML belonging to the fine-grained cluster N within the coarse-grained cluster A. In other words, we refer to A_N as an HML "variant", since it represents a different version of the library corresponding to coarse cluster A, or a customization of the library itself due to developer choices, or, differences introduced by tool-chains/software development kits.

Security Evaluation. When testing a target HML, we leverage HeapHopper to generate a PoC that is compliant with the recovered prototype (as explained in Section III-E). Given an exploitation primitive (e.g., double-free), we generate all the possible permutations of *meaningful* transitions up to a maximum depth of 7 actions (i.e., δ=7). This value was chosen empirically, as it was enough to discover vulnerabilities in all heap implementations tested in our evaluation, maintaining the scalability of our analysis across all the samples within the heterogeneous firmware datasets.

For every depth δ, we use a single *malicious heap transition*, and $\delta-1$ legitimate calls to `malloc` and `free`. During the symbolic tracing, whenever HeapHopper produces a PoV, we filter any false positive using the re-tracing methodology presented in Section III-H.

A. Ground-Truth Dataset

To evaluate HEAPSTER on the *ground-truth* dataset, we use the stripped binary images with no debug symbols. These samples are compiled for different microcontroller units (MCU) and implement

[4]Since the boundaries between the allocator and other libraries code can not be easily identified, we need a heuristic to limit the callees depth.

a heterogeneous set of different applications. The average size of a firmware image in the *ground-truth* dataset is 66KB (median 45KB) with, on average, ∼20 thousand (median ∼15 thousand) opcodes.

HML Identification. We parallelized the HML identification analysis and completed it on all 20 firmware images in 48 hours. We report detailed statistics about memory consumption and time required for each analysis step in Table V of Appendix D. The biggest average memory consumption was 3.5GB (median 3GB) during the ③ Pointer Source Identification stage, while the biggest average time of 3.5 hours (median 1.9 hours) was spent during the ⑤ Deallocator Identification stage. Using the debugging symbols we can confirm that HEAPSTER was able to identify the correct HML (i.e., `malloc` and `free` functions) for *every* firmware image in this dataset. In Section III-B, we show that the *basic functions* are starting points for *pointer sources* identification, ultimately providing allocator candidates. In this evaluation, for 18 samples *memcpy* successfully marked the final allocator as a possible *pointer source*, followed by 9 samples when considering `memset`, and 1 using `strncpy`. Note that, in a given blob, several *basic functions* can mark the final allocator as a *pointer source*. As detailed in Section III-E, when `malloc` or `free` has more than one parameter in their prototypes, we use a constraint counting heuristic to differentiate the parameter representing the size or the pointer to free. This analysis reports that the size parameter has 2.6 more constraints on average than its closest competitor (median of 2), while the parameter containing the pointer to free has, on average, 4.1 more constraints (median 6). In *all* 20 blobs, memory and time limits were never triggered, the heap was growing towards higher addresses, and we always found uses of heap global variables. We report the results of this analysis in Table I.

Coarse-grained Clustering. The coarse clustering divided the identified HMLs into three groups: 8 belong to A, 9 are in B, and 3 in C. We confirm, using the debugging symbols of the related firmware blobs, that these implementations correspond to `nano_malloc` [2], newlib's `malloc` [1] and `lwip_malloc` [49], respectively.

Fine-grained Clustering. The fine-grained clustering identified 8 different sub-clusters in cluster A, 6 in cluster B, and 2 in cluster C, for an overall total of *16* implementation variants. In particular, Table I shows that none of the HMLs in cluster A are reported to be identical, while 2 HMLs belong to cluster B_1, and 3 to cluster B_2. Finally, 2 HMLs in C are reported to be identical (C_1). We manually verified the coarse-grained and fine-grained clusters identified in the ground truth and confirmed the results of this analysis.

HML Security Evaluation. For the *ground-truth* dataset, we tested *all* 20 HMLs with HeapHopper (i.e., even when they are part of the same fine-grained cluster), analyzing an average of 2k PoCs per HML, with a maximum analysis time of 10 minutes per PoC. The complete analysis of a single HML took, in the worst case, a maximum of 3 hours. Overall, the evaluation time for the entire *ground-truth* dataset took 27 hours. Table II summarizes the results.

Results Discussion. The security analysis reported that all heap libraries in clusters A and B are vulnerable to Overlapping Chunks, Non-Heap-Allocation, and Restricted Writes using most of the available exploitation primitives. HeapHopper reported also Arbitrary Write vulnerabilities for clusters A_3, A_4, and all the HMLs in cluster B. Notably, our results show that the HMLs in cluster C are vulnerable *only* to Overlapping Chunks.

TABLE I: Evaluation results for the *ground-truth* dataset with *Size* in KB for each sample. *Functions* reports the *total* number of functions of the firmware, the number of identified *basic functions* and the number of *pointer sources*. *Cluster* shows the HMLs' categorization results. *Patched* indicates if any allocator's functions had to be patched to avoid symbolic execution roadblocks.

Sample			**Functions**			**HML**	
Name	Ref.	Size (KB)	Total	Basic	Pointer Sources	Cluster	Patched
p2im_controllino_slave	[29]	24	280	1	3	A_1	✗
p2im_console	[29]	30	256	4	8	A_2	✗
p2im_gateway	[29]	44	425	4	8	A_3	✗
p2im_drone	[29]	31	190	3	3	A_4	✗
p2im_car_controller	[29]	20	266	3	6	A_5	✗
expat_panda	[43]	94	422	4	48	A_6	✗
atmel_6lowpan_udp_tx	[22]	70	515	3	49	A_7	✗
samr_21_http	[22]	154	311	5	16	A_8	✗
csaw_esc19_csa	[57]	49	204	4	13	B_1	✗
csaw_esc19_csb	[57]	51	212	6	13	B_1	✗
stm32_tcp_echo_server	[22]	115	452	4	25	B_2	✗
stm32_tcp_echo_client	[22]	121	451	4	26	B_2	✗
stm32_udp_echo_server	[22]	112	437	3	23	B_2	✗
st-plc	[22]	167	795	4	25	B_3	✗
rf_door_lock	[33]	41	247	4	16	B_4	✗
thermostat	[33]	40	226	4	17	B_5	✗
nucleo_blink_led	[33]	33	159	4	14	B_6	✗
nxp_lwip_tcpecho	[22]	39	243	3	9	C_1	✓
nxp_lwip_udpecho	[22]	36	227	3	8	C_1	✓
nxp_lwip_http	[22]	74	360	3	12	C_2	✓

B. Wild Dataset

We built the *wild* dataset by using firmware images collected in the wild by previous research on monolithic firmware images. In particular, we use the collection built by Wen et al. [59] by crawling the Google Play Store and searching for Android applications containing firmware images for IoT devices (e.g., a smart watch with a companion app on a mobile phone). This set contains a total of 794 *unique* ARM monolithic firmware images: 769 from Nordicsemi (Nordic), and 25 from Texas Instruments (TI). Furthermore, we use 5 more firmware images related to Fitbit [30] fitness devices [52], for a total of 799 monolithic firmware images. Overall, the firmware images we use in this experiment represent an *extremely* heterogeneous set of different real-world applications, as also shown in the categorization presented in Table III of the Appendix B. The average and median size of a firmware in the *wild* dataset are bigger than the ones reported for the *ground-truth* dataset, 101KB and 76KB, respectively. Moreover, these firmware samples have a bigger number of opcodes, both average ∼29 thousand and median ∼27 thousand (cf. Table V in Appendix D).

HML Identification. Developers are generally discouraged to use HML in embedded systems code (as mentioned in Section II). However, we find that 340 of the 799 firmware images (42%) in the *wild* dataset actually include such a library. Out of these 340 samples, 253 (75%) were automatically identified by HEAPSTER, while the remaining ones were identified by our clustering analysis.

When considering the 253 blobs identified by HEAPSTER, we find the distribution of the *basic functions* (found in all but 7

TABLE II: Security evaluation of HMLs found in the *ground-truth* and *wild* datasets. This table includes *all* the blobs for which we identified an HML leveraging both HEAPSTER and the similarity analysis with BinDiff. We group by sub-clusters of HML types that are affected by the same set of heap vulnerability classes. *Num* represents the cumulative number of firmware samples in the grouped sub-clusters. OC/NHA/RW/AW represent the heap vulnerability classes, while DF/FF/O/UAF the exploitation primitives (as presented in Section III-G). The values represent the total number of *heap transitions* that must be executed (including the *single* exploitation primitive) to trigger the vulnerability.

			OC				**NHA**			**RW**			**AW**		
	HML	Num	DF	FF	O	UAF	FF	O	UAF	FF	O	UAF	FF	O	UAF
Ground Truth Dataset	A_1,A_5	2	7		5	7	2	5	6	2	4	6			
	A_2,A_6	2	7		5	5	2	5	4	2	4	4			
	A_3	1	7		5	7	2	5	6	2	4	6	4	5	6
	A_4	1	7		5	5	2	5	4	2	4	4	4	5	6
	A_7,A_8	2	7		5	7	2	5	7	2	4	6			
	B_1,B_2,B_3,B_4	7	7		5	6	2	7	7	2	4	5			5
	B_5,B_6	2	7		5	6	2		7	2	4	5			5
	C_1	2			6	7									
	C_2	1			6	6									
	Total	**20**													
Wild Dataset	$D_{0\text{-}2}$,M_0	117	7		6	5		5	5		5	5			
	$E_{0\text{-}1}$,$E_{4\text{-}8}$,E_{10}	70	7		5	7	2	5	6	2	4	6			
	$E_{2\text{-}3}$,E_9	21	7	3	5	7	2	5	6	2	4	6			
	E_{11}	2	7		5	7	3	5	6	2	4	6			
	F_0	51			6			6			5	5			
	G_0	19			6	6				2	6	7			
	G_1	7			7	6				2	6	7			
	$H_{0\text{-}2}$	19			4	6									
	I_0	7			5	6		5	6		5	5			
	I_1	5			6	6		5	6		5	5			
	I_2	4			6	6		5	7		5	5			
	I_3	2			5	6		5	6		6	6			
	J_0	3	7		5	6	2	7	7	2	4	5			5
	$J_{1\text{-}2}$	3	7		5	6	2		6	2	4	5			5
	K_0	3	7		6	6		5	6		5	5			
	K_1	2	7		6	6		6	6		5	6			
	L_0	5	7		6	6					7	6			
	Total	**340**													

samples) used to correctly identify `malloc` as a *pointer source* to be quite similar to the ones found in the *ground-truth* dataset. In particular, the `memcpy` *basic function* marked the final allocator as a *pointer source* in 237 blobs, `memset` — in 80, `memcpy` and `strlen` — in 3 each, and `strncpy` — in 2. During our HML analysis, the ④ Allocator Identification step took the most average and median memory, 4.5GB and 2.2GB, respectively. This is also the step where HEAPSTER spent the most time: 2.4 hours average and 1 hour median. It is worth noting that the analysis of only 17 samples was terminated because of a timeout was reached, i.e., non-terminating steps ⑥ and ⑦, as discussed in Section V-C, but the memory limit was never exhausted. Although the firmware images in the *wild* dataset have larger average sizes than the ones in *ground-truth* dataset, HEAPSTER performed better (considering average and median numbers) on the *wild* dataset, as the detailed statistics reported in Appendix D, Table V. Finally, in the *wild* dataset, the `malloc` parameter and the pointer to `free` have both on average 2.3 (median of 2) more constraints than the closest competitor.

As was reported for the *ground-truth* dataset, we also identified that every firmware image uses packed heap global variables. Finally, in all but 5 samples the heap grows towards higher addresses.

To understand if HMLs are predominant for specific applications, we classify our firmware images using the categories described by Wen et al. [59]. The results of this classification are presented in Appendix B, Table III. This classification shows that HMLs are used in numerous firmware images that span across different categories: from Wearable (83 blobs) and Sensors (24), to Medical Devices (22).

Coarse-grained Clustering. The coarse clustering analysis of the identified HMLs yielded 10 different implementations, which we label as clusters D to M. According to our results, the most used HML is D, which is embedded in 115 firmware blobs, followed by E with 93 samples, and F with 51 (all the cluster sizes are reported in Appendix C, Figure 4). After considering the similarity across datasets, we observed that cluster E represents the same HML as A, while J is the same as B. Therefore, the *wild* dataset contains 8 *new* HML implementations. The graph in Appendix F shows a visualization of the identified coarse-grained clusters.

Fine-grained Clustering. When searching for identical implementations with the fine-grained clustering, we discovered a total of *32* HML variants. Interestingly, E seems to have the biggest number of reported variants (12), which might suggest that this implementation is popular among different tool-chains (we break down the number of variants identified in each coarse-grained cluster in Appendix C, Figure 4). Moreover, as we previously identified that E and J match libraries A and B respectively, we checked for identical variants in E-A and J-B and found no overlaps between them.

HML Security Evaluation. To reduce the number of analyses performed by HeapHopper, we test *one* HML per identified variant, for a total of 32 HMLs. To select these samples, which are reported in Table IV of Appendix C, we randomly chose a representative firmware for every HML variant. For this experiment, we tested on average 2k PoCs per HML, with a time constraint of 10 minutes each. Every single HML has been analyzed by HeapHopper in less than 3 hours, with a total analysis time of 36 hours to analyze all the selected firmware images. All the PoVs produced by HeapHopper are checked using the approach presented in Section III-H with a time limit of 5 minutes per PoV re-execution. If the PoV execution does not trigger the reported heap vulnerability, or does not respect the configured time limit, we consider it a false positive and continue the analysis to find another PoV. Table II summarizes the results of the security evaluation across the 32 HML variants.

Results Discussion. Similar to the *ground-truth* dataset, the security analysis results show that *all* tested HMLs variants are vulnerable to *at least* Overlapping Chunks and Restricted Write. This means that *all* the 340 firmware blobs that are using these libraries can be exploited by a heap attack if the right exploitation primitive is found in the application's code. In particular, as all of the analyzed HMLs leverage heap inline metadata (discussed in Section III-E), without implementing *any* safety measure to protect them, *heap overflow* primitives are always very effective in giving an attacker the possibility to manipulate the heap, opening the door for a complete takeover of the application.

C. HML Identification: False Negatives

In this subsection, we discuss the false negatives of our HML identification on the *wild* dataset as well as the ways we mitigate them.

Similarity Match. We use the coarse-grained clustering algorithm on the *wild* dataset, leveraging the 253 HMLs detected by HEAPSTER as a target library of known HML functions. Thanks to the binary similarity algorithm used by BinDiff, we detected 85 additional blobs that contain an HML but that were not detected by the initial HML identification analysis. When investigating these new 85 blobs, we discover that 69 of them perfectly match HMLs in a known fine-grained cluster, while 16 need further attention.

Imperfect Match. We manually investigated the 16 firmware images that were included in the graph presented in Appendix F, but we could not assign them to a fine-grained cluster (i.e., no perfect match with any other HML was reported by BinDiff, and HEAPSTER did not identify a working allocator). After a manual investigation, we discovered that these blobs actually did belong to existing fine-grained clusters, and the reason for the imperfect match was related to imprecision in the disassembler employed by BinDiff when analyzing the target functions. Therefore, we proceeded to include these blobs into the correspondent fine-grained cluster.It is worth noting that adding a blob to a fine-grained cluster means that there exists a firmware image with the exact same HML that we were able to test with HeapHopper. This increased the number of firmware images with an HML to a total of 338.

Package Name. Using the metadata information released by Wen et al. [59], we group firmware images depending on the Android packages' names of the application containing the firmware blob. By leveraging this information, we identify groups where only a subset of blobs was reported containing an HML by HEAPSTER. Then, we pinpoint the samples for which we could not identify an HML (40 firmware images) and performed a manual investigation over them. We discovered that, on the one hand, 17 cases were indeed false negatives, but they did not affect our results as they were included in a fine-grained cluster (i.e., their HML has already been evaluated by using the fine-grain cluster representative). On the other hand, for the remaining 23, we confirmed that there was indeed no HML within the firmware (i.e., they are true negatives).

Random Sampling. Finally, we randomly selected an additional 50 firmware images that were not part of any cluster nor were they part of the set of blobs that we check during the package name investigation. Out of the 50 firmware images, we confirmed 46 samples to be true negative (i.e., no HML is apparently used by the blob), 2 contain an allocator not in scope as discussed in Section II, and 2 are confirmed to be false negatives. We investigated the latter false negatives by running a more relaxed coarse-grained function binary similarity (we use a similarity score of 0.6) as explained in V-B. As a result, BinDiff reported a similarity score of 0.63 with 0.71 confidence with one of the blobs in cluster L. We confirmed this by manually comparing the HML implementations used by the binaries and eventually added these two blobs to the fine-grained cluster L_0. After this investigation, we ended up with a total of 340 firmware images containing an HML.

False Negatives Reasons. The main source of false negatives for our system is related to the failure to identify allocator functions (40 blobs). This means that either HEAPSTER was not able to find the correspondent heap initializer, or it could not find a connection from a *pointer source* to a *basic function*, preventing the analysis discussed in III-E to identify the correct function. The second most common reason is the failure to detect a working allocator-deallocator pair (29 blobs). This can occur when `free` is not correctly identified, or when the HML execution does not respect our criteria (as discussed in Section II). Finally, steps ⑥ and ⑦ did not terminate for 8 and 9 firmware images, respectively. However, as we explained in the previous paragraphs, none of these false negatives affected our final results, as we always managed to identify the correct HML with the help of our clustering approach.

D. Security Impact of Vulnerabilities in HML for Applications

Threat Model for Applications. We assume that an attacker has control over the input data that a firmware image receives via its peripherals. This usually corresponds to data received over any kind of network interface (e.g., Bluetooth or WiFi), serial interface, or from sensors. For our evaluation, we consider all MMIO read functions as possible entry-points and threat vectors for the firmware. In particular, we focus our attention on identifying paths within the firmware code that connect a data read from an MMIO function to a `malloc` or `free`. This is because such a path might allow an attacker to allocate and de-allocate memory on demand, and potentially, manipulate a vulnerable HML to exploit a device.

Automated Vulnerable Candidate Selection. Our attack model requires us to find vulnerabilities that are dependent on malicious user input, and that can affect the state of the system during its execution. Unfortunately, we are faced with the challenge of performing our analysis without access to a real execution environment for the firmware images (as we do not have the actual devices), and without reliable information about the true sources of external user input. To address this challenge, we developed an automated static technique, based on reaching definitions, that identifies whether an MMIO function has a static path to a call to `malloc` with a variable size or a call to `free`. Furthermore, in case of `malloc`, we check whether the definition it produces is used by a *basic function* (e.g., `memset` or `memcpy`) with variable size writing capabilities. This analysis aims to select blobs that are most accessible for further manual investigation. This approach identified 54 blobs (among all samples) that we used as the starting point for further manual analysis.

Manual Investigation. We carried out a best-effort manual search for vulnerabilities in the 54 blobs. This task required a week of work from two senior security researchers. During our manual investigation, we identified four firmware images that can be affected by an integer overflow bug. In particular, this vulnerability could lead to a small heap allocation being overflown by a subsequent *memcpy* (i.e., heap overflow). The discovery of these heap exploitation primitives makes the respective firmware images meet all the conditions we need for our threat model. However, while the discovery of these vulnerabilities provides some indication that the firmware image could be exploited in practice, a full re-hosting solution would be necessary to confirm our findings. For this reason, in the next section, we demonstrate how we confirmed vulnerabilities discovered by HEAPSTER on a real device.

Hardware Example. To validate HEAPSTER in a real-world scenario, we use an *STM32-NucleoF401RE* board expanded with an *X-NUCLEO-IDW01M1* Wi-Fi Module, as shown in Figure 5 of Appendix E. This hardware satisfies the prerequisites of our threat model, since an attacker can interact with the firmware code via the Wi-Fi module. We flashed on the board a firmware application that sends and receives packets over WiFi. To implement the application, we used the Mbed Studio IDE [10], which automatically selects the HML implemented in the *mbed library* version 172 [9]. We then run HEAPSTER on the resulting firmware image. The system correctly identified the generic HML chosen by the IDE in the blob. Moreover, it determined that the HML is vulnerable to OC, NHA, RW. We implemented an end-to-end exploit for the hardware device itself and confirmed the HML to be vulnerable to all discovered attacks.

VI. DISCUSSION AND LIMITATIONS

Loading Firmware. The results of our analyses depend on the precision of several data structures recovered from the firmware image (i.e., CFG, function boundaries, and the callgraph). In particular, function boundaries and the callgraph are heavily dependent on the quality of the CFG analysis. To extract this information we rely on the algorithms implemented in the `angr` framework. However, building precise CFGs for firmware blobs is a hard problem that presents numerous engineering challenges [4], [5].

Firmware Emulation. Inspired by previous work in monolithic firmware analyses [22], [29], [33] our approach internally uses an emulator to lift the assembly into an IR, to eventually execute the code. Consequently, the quality of our results depends on the precision and correctness of the emulator mimicking the concrete semantics of the program execution on a bare-metal device.

Basic Function Identification. During our large-scale evaluation, we found at least one *basic function* in 762 monolithic firmware images. However, *basic functions* can be inlined, not present at all, or not identifiable. To mitigate this issue, we could define addresses of instructions that perform memory writes as starting points of our *pointer sources* identification. Indeed, this would increase the opportunities to detect new *pointer sources*, and therefore, `malloc`. However, this would also directly largely affect the performance of the *pointer sources* identification analysis, thus we limit our starting points to registers supplied as arguments to *basic functions*.

Pointer Source Identification. As described in Subsection III-B the *pointer sources* identification analysis uses `angr`'s reaching definitions (RD) framework. However, RD has no support for stores done in memory locations where the address is not known statically (i.e., dynamic memory locations). Therefore, when the value returned by `malloc` is stored at a dynamic location, RD cannot keep track of the definition of that memory. Hence, when the pointer is later used in a *basic function*, we will miss this dependency.

Heap Initialization. When our approach to initialize the firmware memory fails (i.e., not all the memory is correctly initialized), we potentially end up with uninitialized heap global variables being accessed during the execution of `malloc` and `free`. To address this issue, we attempt to initialize the uninitialized memory with zero. If this is not fatal, and `malloc` is returning valid heap memory addresses, we proceed with our analyses. However, this can lead to false positives and false negatives that cannot be easily resolved automatically by HEAPSTER, but require either a test on the real hardware or a manual check of the generated PoV by a human analyst.

Exploitation Evaluation. When evaluating the 54 blobs selected by our analysis in Section V-D, we consider *all* the functions that access MMIO data as possible entry points for the attacker. However, if an MMIO function is not reading data from a peripheral that is receiving inputs from users, this analysis can generate false positives. Moreover, the mere presence of a static path cannot say anything regarding the feasibility of this path (that connects an MMIO function to a `malloc`/`free` callsite) during runtime. Thus, our manual review can only provide limited guarantees about the discovered threats. We consider building a robust system to detect heap exploitation primitives in applications as future work.

VII. RELATED WORK

A. Heapster vs. Membrush

The HML identification analyses implemented in this paper share with `Membrush` [19] the goal of detecting custom allocators in binaries. `Membrush` leverages binary instrumentation and full program execution to identify the allocator embedded in a desktop binary. In a nutshell, `Membrush` runs a target program on a native system — using as input the test cases of the target — and leverages Intel Pin [40] to instrument the program and analyze its dynamic behavior. The key difference between HEAPSTER and `Membrush`, is that our tool focuses on monolithic firmware images, while `Membrush` targets desktop binaries. While at first glance this difference might seem simple to overcome, the implications are far-reaching. First of all, since HEAPSTER is working with monolithic firmware images, it cannot rely on test cases to execute the firmware code, because they are rarely, if ever, available. A solution to this problem could be to automatically generate the inputs leveraging a fuzzer, but this would also require additional research to understand how to inject the inputs without a clear I/O interface defined in the firmware. Additionally, this would require generating good test cases that cover enough code to eventually identify the allocator. Finally, full firmware code execution would also require a *complete* re-hosting solution, which not only would add engineering complexity to the system, but is still an open problem when trying to analyze a large collection of blobs [28].

B. Heap Management Library Security

Different approaches have been proposed to analyze the security of heap allocators. In this paper, we leveraged and modified the work done by Eckert et al. [24] to symbolically execute the heap primitives identified inside a firmware blob. In particular, we use the same bounded model checking technique to discover whether a memory allocator is vulnerable to a class of known heap vulnerabilities. Recently, Yun et al. [63] proposed a system to discover *new* heap exploitation techniques that leveraged a fuzzer to bring the heap into new corrupted states. Heelan et al. proposed Gollum [34], a system that can perform automatic heap layout manipulation and exploitation of interpreter programs. In a similar vein, Zhao et al. [64] proposed an automatic framework to guide the exploitation of heap vulnerabilities. Finally, more recently, Wang et al. proposed Maze [62], a system that leverages symbolic execution and Linear Diophantine Equations to re-create

the heap layout required to exploit a heap vulnerability. Researchers have also focused on the automatic exploitation of the kernel heap. In particular, FUZE [61] provides a system to facilitate exploits generation for kernel use-after-free vulnerabilities, while Slake [20] facilitates the manipulation of a kernel-specific allocator (i.e., the slab allocator [13]) to eventually exploit vulnerabilities in the Linux kernel. However, the state-of-the-art research about the security of dynamic memory allocators has focused on traditional systems (i.e., desktop computers and servers). To the best of our knowledge, this is the first work to identify, and test, the security of memory allocators in the embedded world, and, in particular, in monolithic firmware.

C. IoT Vulnerabilities

The security testing of IoT devices and their firmware (Linux-based and not), has been the target of numerous research works in the past few years. Muench et al. [43] elaborated on the challenges introduced by firmware when applying traditional fuzzing techniques to this domain. In particular, the authors demonstrated that the side effects of a memory corruption inside a firmware sample are different from the ones observed over classic binaries, and, therefore, the effectiveness of classic tools is drastically reduced. Chen et al. [18] proposed IoTFuzzer, a fuzzing methodology aimed at discovering memory corruption bugs using the companion android app of IoT devices. Similarly, Redini et al. proposed Diane [47], a system to generate under-constrained inputs for embedded devices that consume data from companion applications. More recently, Wen et al. proposed FirmXRay [59], a system that uses static analysis to discover Bluetooth link layer vulnerabilities in a large number of monolithic firmware images. Ruge et al. [50] have also focused on Bluetooth devices, and proposed a fuzzing system based on firmware emulation of a specific Bluetooth board to uncover memory corruption vulnerabilities in the Bluetooth stack. Finally, Feng et al. proposed P^{2}IM [29], a hardware-independent firmware testing framework that uses an external fuzzer to provide inputs inside the target. The aforementioned systems cover specific kinds of bug classes, but none of them focus on understanding the weaknesses that affect heap management libraries employed by monolithic firmware images.

D. Re-hosting solutions

The analyses presented in this paper rely on emulating and symbolically exploring parts of the firmware code. While HEAPSTER *does not need* a full re-hosting system as it performs targeted execution of selected functions, a re-hosting solution would have certainly benefited our analyses, and removed the necessity of execution models based on empirical configurations. However, re-hosting and precise code execution of firmware code is still an open problem [60]. Gustafson et al. [33], proposed a system to perform automatic re-hosting of monolithic firmware by using information recovered from real interactions between the firmware and the peripherals. Muench et al. proposed Avatar^2 [42], a system that leverages "hardware in the loop" to forward I/O interactions with unsupported peripherals to the real device. More recently, Clements et al. [22] proposed a re-hosting solution based on replacing the functions in the high-level hardware abstraction layer (HAL) with generic implementations inside a full-system emulator.

To tackle the problem of peripheral interactions, Cao et al. proposed Laelaps [15], an MCU agnostic system that leverages concolic execution to execute firmware code. Similarly, a recent work proposed Jetset [26], a system based on symbolic execution that infers what behavior firmware expects from a target device, and synthesizes peripherals models that can later be imported into an emulator. Zhou et al. proposed µEmu [65], a system that leverages symbolic execution to infer how to respond to unknown peripheral accesses at individual access points. Finally, Fuzzware [51] proposes a fine-grained MMIO modeling approach that leverages a coverage-guided fuzzer to test unmodified firmware in a scalable way.

VIII. CONCLUSIONS

In this paper, we present HEAPSTER, a system that automatically identifies heap management libraries (HMLs) contained in monolithic firmware images and tests their security. This work represents the first attempt at analyzing the security of dynamic memory allocators used in monolithic firmware. We show that identifying the heap management library inside a firmware blob — with no symbols and limited support for dynamic execution — is a *very* challenging task. In particular, we leverage different heuristics and domain-specific intuitions to address the immense heterogeneity of the firmware domain. Our evaluation demonstrates that it is possible to identify the heap management primitives and to precisely execute them to test for the presence of critical heap vulnerabilities. We use HEAPSTER to identify the HML inside 819 real-world monolithic firmware images (considering both the *ground-truth* and the *wild* datasets). To the best of our knowledge, this is the biggest evaluation of a technical work that uses a combination of static and dynamic program analysis techniques to analyze monolithic firmware images. In particular, inside the 340 blobs for which we detected a dynamic memory allocator, we discovered *11* different heap implementation families, and a total of *48* different heap implementation variations. To test the security of the identified HMLs, we leverage a modified version of HeapHopper, and we show that these libraries are *all* affected by multiple classes of heap vulnerabilities. Our work sheds some light on the state of the security of embedded allocators, which has been largely ignored by the security community, and shows a quite worrisome picture. As IoT becomes increasingly ubiquitous, and firmware analysis tools are getting more powerful, we hope vendors and embedded system developers will start to provide guidelines, and security-vetted libraries, to bring safer products into our society.

IX. ACKNOWLEDGEMENTS

This material is based upon work supported all or in part by Office of Naval Research (ONR) under awards N00014-17-1-2011, N00014-20-1-2632, N00014-17-1-2897, by DHS award FA8750-19-2-0005, and by DARPA award HR001118C0060. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the US Government. We would like to thank the anonymous reviewers for their valuable feedback, and Haohuang Wen for the support regarding the dataset collected in FirmXRay [59].

References

[1] 32bitmicro. libc malloc. https://github.com/32bitmicro/newlib-nano-1.0/blob/master/newlib/libc/sys/linux/malloc.c#L2815, 2012.

[2] 32bitmicro. nanomalloc. https://github.com/32bitmicro/newlib-nano-1.0/blob/master/newlib/libc/stdlib/mallocr.c#L192, 2012.

[3] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *25th USENIX Security Symposium*, pages 583–600, 2016.

[4] angr. The Great ARM CFG Challenge 1. https://github.com/angr/angr/pull/1668, 2020.

[5] angr. The Great ARM CFG Challenge 2. https://github.com/angr/angr/pull/2075, 2020.

[6] Orlando Arias, Dean Sullivan, and Yier Jin. Ha2lloc: Hardware-assisted secure allocator. In *Proceedings of the Hardware and Architectural Support for Security and Privacy*, pages 1–7. 2017.

[7] ARM. Cortex-m3 embedded software development. https://www.eecs.umich.edu/courses/eecs373/readings/ARM_Cortex_AppNote179.pdf, 2007.

[8] ARM. Cortex-m3 vector table. https://developer.arm.com/documentation/dui0552/latest/the-cortex-m3-processor/exception-model/vector-table, 2007.

[9] ARM. Default hml used by mbed ide. https://os.mbed.com/users/mbed_official/code/mbed//rev/65be27845400/, 2021.

[10] ARM. mbed ide. ide.mbed.com, 2021.

[11] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium USENIX Security 14)*, pages 845–860, 2014.

[12] Michael Barr. Firmware-specific bug #5: Heap fragmentation. https://embeddedgurus.com/barr-code/2010/03/firmware-specific-bug-5-heap-fragmentation/, 2010.

[13] Jeff Bonwick et al. The slab allocator: An object-caching kernel memory allocator. In *USENIX summer*, volume 16. Boston, MA, USA, 1994.

[14] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[15] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation. In *Annual Computer Security Applications Conference*, pages 746–759, 2020.

[16] CEA IT Security (IT Security at the French Alternative Energies and Atomic Energy Commission). Sibyl: A miasm2 based function divination. https://github.com/cea-sec/Sibyl, 2019.

[17] Check Point. Safe-linking - eliminating a 20 year-old malloc() exploit primitive. https://research.checkpoint.com/2020/safe-linking-eliminating-a-20-year-old-malloc-exploit-primitive/, 2020.

[18] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*, 2018.

[19] Xi Chen, Asia Slowinska, and Herbert Bos. Membrush: A practical tool to detect custom memory allocators in c binaries. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 477–478. IEEE, 2013.

[20] Yueqi Chen and Xinyu Xing. Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1707–1722, 2019.

[21] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. Halucinator: Firmware re-hosting through abstraction layer emulation. In *29th USENIX Security Symposium USENIX Security 20)*, pages 1201–1218, 2020.

[22] Clements, Abraham and Gustafson, Eric and Scharnowski, Tobias and Grosen, Paul and Fritz, David and Kruegel, Christopher and Vigna, Giovanni and Bagchi, Saurabh and Payer, Mathias. HALucinator: Firmware Re-hosting through Abstraction Layer Emulation. In *USENIX Security Symposium*, 2020.

[23] Moritz Eckert. Security implications of tcache. https://sourceware.org/legacy-ml/libc-alpha/2018-02/msg00298.html, 2018.

[24] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Heaphopper: Bringing bounded model checking to heap implementation security. In *27th USENIX Security Symposium USENIX Security 18)*, pages 99–116, 2018.

[25] Embeddedinsights. Question of the week: Do you use or allow dynamic memory allocation in your embedded design? http://www.embeddedinsights.com/channels/2010/03/24/question-of-the-week-do-you-use-or-allow-dynamic-memory-allocation-in-your-embedded-design/, 2012.

[26] Johnson Evan, Bland Maxwell, Zhu YiFei, Mason Joshua, Checkoway Stephen, Savage Stefan, and Levchenko Kirill. Jetset: Targeted firmware rehosting for embedded systems. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021.

[27] Chris Evans. glibc patch. https://sourceware.org/git/?p=glibc.git;a=commit;h=17f487b7afa7cd6c316040f3e6c86dc96b2eec30, 2017.

[28] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, et al. Sok: Enabling security analyses of embedded systems via rehosting. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2021.

[29] Bo Feng, Alejandro Mera, and Long Lu. P 2 im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[30] Fitbit. Fitbit. https://www.fitbit.com/global/us/home, 2022.

[31] Zynamics GmbH. Bindiff manual. https://www.zynamics.com/bindiff/manual/#chapUnderstanding, 2020.

[32] GNU.org. The gnu c library (glibc). https://www.gnu.org/software/libc/, 2021.

[33] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, et al. Toward the analysis of embedded firmware through automated re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, pages 135–150, 2019.

[34] Sean Heelan, Tom Melham, and Daniel Kroening. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1689–1706, 2019.

[35] Grant Hernandez, Farhaan Fowze, Dave Jing Tang, Tuba Yavuz, Patrick Traynor, and Kevin RB Butler. Toward automated firmware analysis in the iot era. *IEEE Security & Privacy*, 17(5):38–46, 2019.

[36] Muhui Jiang, Yajin Zhou, Xiapu Luo, Ruoyu Wang, Yang Liu, and Kui Ren. An empirical study on arm disassembly tools. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 401–414, 2020.

[37] Joseph, Yiu and Andrew, Frame. Cortex-M Processors and the Internet of Things (IoT). https://community.arm.com/cfs-file/__key/communityserver-blogs-components-weblogfiles/00-00-00-21-42/White-Paper_2D00_Cortex_2D00_M-Processors-_2600_-the-IoT.pdf, 2013.

[38] M Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. Speculative disassembly of binary code. In *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*, pages 1–10. IEEE, 2016.

[39] Beichen Liu, Pierre Olivier, and Binoy Ravindran. Slimguard: A secure and memory-efficient heap allocator. In *Proceedings of the 20th International Middleware Conference*, pages 1–13, 2019.

[40] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

[41] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. Probabilistic disassembly. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1187–1198. IEEE, 2019.

[42] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar2: A multi-target orchestration platform. In *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, volume 18, pages 1–11, 2018.

[43] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS*, 2018.

[44] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[45] Gene Novark and Emery D Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 573–584, 2010.

[46] Carlos ODonell. Security implications of tcache. https://sourceware.org/legacy-ml/libc-alpha/2018-02/msg00313.html, 2018.

[47] Nilo Redini, Andrea Continella, Dipanjan Das, Giulio De Pasquale, Noah Spahn, Aravind Machiry, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for iot devices. In *In Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, May 2021.

[48] Chris Rohlf. Isoalloc. https://struct.github.io/iso_alloc.html, 2020.

[49] RT-Thread. lwip malloc. https://download.savannah.nongnu.org/releases/lwip/lwip-1.4.0.zip, 2013.

[50] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets. In *29th USENIX Security Symposium USENIX Security 20)*, pages 19–36, 2020.
[51] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using precise mmio modeling for effective firmware fuzzing.
[52] Secure Mobile Networking Lab (Seemoo-lab). Collection of fitness firmware. https://github.com/seemoo-lab/fitness-firmware/tree/master/firmwares, 2021.
[53] Shellphish. Educational heap exploitation. https://github.com/shellphish/how2heap, 2020.
[54] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.
[55] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. Freeguard: A faster secure heap allocator. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2389–2403, 2017.
[56] Sam Silvestro, Hongyu Liu, Tianyi Liu, Zhiqiang Lin, and Tongping Liu. Guarder: A tunable secure allocator. In *27th USENIX Security Symposium USENIX Security 18)*, pages 117–133, 2018.
[57] TrustworthyComputing. CSAW Embedded Security Challenge. https://github.com/TrustworthyComputing/csaw_esc_2019, 2019.
[58] Richard Wartell, Yan Zhou, Kevin W Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating code from data in x86 binaries. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 522–536. Springer, 2011.
[59] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. Firmxray: Detecting bluetooth link layer vulnerabilities from bare-metal firmware. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 167–180, 2020.
[60] Christopher Wright, William A Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A Clements. Challenges in firmware re-hosting, emulation, and analysis. *ACM Computing Surveys (CSUR)*, 54(1):1–36, 2021.
[61] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *27th USENIX Security Symposium USENIX Security 18)*, pages 781–797, 2018.
[62] Wang Yan, Zhang Chao, Zhao Zixuan, Zhang Bolun, Gong Xiaorui, Zou Wei, and Levchenko Kirill. MAZE: Towards automated heap feng shui. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021.
[63] Insu Yun, Dhaval Kapil, and Taesoo Kim. Automatic techniques to systematically discover new heap exploitation primitives. In *29th USENIX Security Symposium USENIX Security 20)*, 2020.
[64] Zixuan Zhao, Yan Wang, and Xiaorui Gong. Haepg: An automatic multi-hop exploitation generation framework. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 89–109. Springer, 2020.
[65] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. Automatic firmware emulation through invalidity-guided knowledge inference. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021.
[66] Zynamics. Zynamics bindiff. https://www.zynamics.com/software.html, 2020.

APPENDIX

A. Function Execution Models

In this Appendix, we provide technical details about the execution models used to emulate different functions in the firmware image.

Basic Function. When looking for *basic functions* (as explained in Section III-B), we simply setup the arguments of the procedure with values compatible with the prototype of the *basic function* we are trying to identify. After that, we simply run the function with a timeout of 10 seconds.

ResetHandler. When executing the `ResetHandler`, we focus the execution to target the loops responsible for the unpacking of the firmware's global variables (as discussed in Section III-C). In particular:

1) We terminate every loop with a symbolic guard. This is based on the intuition that any loop depending on symbolic data (during the firmware bootstrap), must depend on peripherals data, and, therefore, does not implement a compiler-injected stub responsible for unpacking firmware's global variables.
2) We stop the execution whenever an `angr`'s unsupported ARM SuperVisor Call instruction (SVC) is being executed.
3) We force a return to the caller whenever the RIP register contains an address out of the main binary's ROM address space.
4) We do not follow any function call during the execution of the `ResetHandler`.
5) We stop the execution whenever we hit the first basic block of a potentially infinite loop detected by purely static analysis in `angr`.
6) We return a fresh symbolic variable every time an access to the MMIO region is detected. This is necessary to overcome issues related to *time-dependent memory locations*. More specifically, as shown in Figure 3, the content of the memory location at `0x40064006` is expected to change during the peripherals initialization. When returning a fresh symbolic variable at every access, `angr` has the opportunity to concretize its value to the one necessary to break out of the loop, and therefore to advance the execution.
7) We stop the execution when leaving the `ResetHandler` function with a *callout* (i.e., a jump to another function that never returns to the caller). In fact, we assume for simplicity that a *callout* corresponds to the transition from the `ResetHandler` to the firmware's `main` function.

```
1  while ( (MEMORY[0x40064006] & 2) == 0 );
2  while ( (MEMORY[0x40064006] & 0x10) != 0 );
3  while ( (MEMORY[0x40064006] & 0xC) != 8 );
```

Fig. 3: Time-dependent memory location.

Heap Initializers. Even if *heap initializers* are commonly very simple functions per se, they can sometimes be part of a bigger library initialization procedure, which, in turn, can call multiple other functions before reaching the code responsible for writing the heap global variables. For this reason, we need to use a less strict execution model that forces the execution to make progress, without terminating it prior to the heap initialization. In particular, to execute the *heap initializers* we follow strategy steps 1)-5) defined for `ResetHandler`, plus:

6) We stop the execution whenever we reach the address of a *pointer source* because we assume that when calling a *pointer source* the heap has already been initialized.
7) We follow function calls only if the arguments are concrete.
8) We expect *heap initializers* to only write concrete data in memory, and to be executed at the bootstrap of the firmware. Therefore, we do not execute functions with symbolic arguments.
9) We stop the execution whenever we detect a read/write memory operation over a symbolic address.
10) We timeout the execution of a function after 30 minutes.

Pointer Sources & De-allocator Candidate Execution. During the analyses presented in Sections III-C and III-D, we execute the

functions using steps 1)-3) of the *ResetHandler*'s execution model, and:

4) We limit concrete loops iterations to 100.
5) We stop the execution of a function after 15 seconds. In fact, we expect the execution of HML functions to be very quick as these procedures must have high performances.

B. Firmware HML Usage Categorization

Table III presents the distribution of firmware images in our dataset across different categories. For each category, we also report how many images contain an HML library according to our evaluation in Section V. The main observation is that HMLs are rather widespread and used in all identified categories.

Category	Blobs with heap	Tot. Blobs
Wearable	83	209
Generic Upgrade Tool	50	51
Others	34	53
Sensor	24	67
Medical Devices	22	41
Bike Accessory	19	40
Smart Eyeglasses	19	19
Tracker	16	58
Switch	14	20
Car Accessory	9	25
Robot	9	41
Smart Lock	7	15
Smart Light	7	21
Battery	6	9
Smart Home	5	20
Game Accessory	4	9
Agricultural Equipment	3	10
Thermometer	2	16
Beacon	2	12
Firearm Accessory	2	11
Headphone	2	2
Alarm	1	2
Total	340	799

TABLE III: Categories breakdown for firmware blobs used in the evaluation.

C. Wild Dataset Reports

For each of the blobs from the *wild* dataset discussed in Subsection V-B, Table 4 presents their (sub)cluster, total number of functions, number of *basic functions* and *pointer sources*. Also, we report whether the HML required patching. Figure 4 reports the number of blobs and variants per coarse cluster. Interestingly, we observe a big representation for clusters D, E, and F with a rather small variance. Most likely, this indicates that these blobs use standardized HML coming from vendors/IDEs.

D. Resource Usage Statistics during HML Identification

In Table V we report the average memory consumption and median memory consumption in Megabytes (MB) for each stage of the analysis detailed in Section III. Reported metrics were consistently bound by 4.5GB across diffident analysis stages and datasets allowing our HML analysis to be performed either on a general-purpose machine or on an upscale one with several instances run in parallel. Also, we present the average and median time (in seconds) spent in each analysis step in Table V. The

Sample		# Functions			HML	
Name	Size (KB)	Tot.	B	S	C	P
AC603_0101_V0.9.18_191114_1131.bin@2d2b	101	843	3	11	D_0	✗
BSW20204006.bin@be06	184	1385	4	14	D_1	✗
AC603_VIITA_BT_GS_V058_180414_1610.bin@a851	87	715	3	8	D_2	✗
Exakt_Pedal_Radio_Firmware.bin@33d1	83	681	4	3	E_0	✗
BSRLWK_h10_s9_20191124.bin@caa9	121	520	5	17	E_1	✗
nrf52832_xxaa.bin@000d	90	454	4	10	E_{10}	✗
app_fw_RELEASE.bin@a4fa	53	439	3	7	E_{11}	✗
BME-100.bin@57a8	119	621	4	35	E_2	✗
bsafebeacon-S110.bin@e348	47	267	4	4	E_3	✗
ICP_NRF52.bin@6b7c	117	1065	6	12	E_4	✗
BP_application.bin@1175	100	645	6	8	E_5	✗
Hoot_release_pca10040.bin@87ce	121	560	6	11	E_6	✗
nrf52832_xxaa_s132.bin@7370	117	382	4	20	E_7	✗
dddock_app_dock.bin@d2f5	63	490	5	14	E_8	✗
BLERemote.bin@c7bb	43	464	5	14	E_9	✗
trigno_update_v040.024_T014.bin@12f3	283	903	3	2	F_0	✓
plugin_.bin@e6f4	125	1172	4	22	G_0	✗
LRIP_nRF52_release.bin@9c15	167	1470	5	12	G_1	✗
qtBrainoad_Car_Release_oad.bin	46	406	5	3	H_0	✗
ble5_project_zero_cc13x2r1lp_a	167	1170	5	22	H_1	✗
new_bin.bin	42	480	3	9	H_2	✗
nrf52832_xxaa.bin@c376	168	1437	6	44	I_0	✗
w-qcpr-sensor-mk3_release_0.45.1.69.bin@eece	385	2203	12	25	I_1	✗
sma10b_firmware.bin@4e84	284	1437	5	63	I_2	✗
pavlok_2_2008_0930.bin@739e	98	1037	4	6	I_3	✗
nrf52832_xxaa.bin@589c	173	829	4	38	J_0	✗
Exakt_Pedal_Radio_Firmware.bin@3dd4	90	673	3	12	J_1	✗
plot.bin@1050	81	574	4	14	J_2	✗
Bond_Gen2.bin@5ff8	52	790	4	2	K_0	✗
LinOn_Pro_RC18.bin@94ee	68	907	5	7	K_1	✗
bicult_ble_sdk15_sd_132v6.bin@8e6c	112	1047	5	21	L_0	✗
tag-firmware.bin@4acb	135	1062	5	11	M_0	✗

TABLE IV: Firmware blobs in the wild dataset that have been tested with HeapHopper. Column Tot. show the total number of functions, B the number of identified *basic functions*, S the number of *pointer sources*, C identified *cluster*, and P whether is the HML needed to be patched or not.

average stage execution time topped at 3 hours 50 minutes, median at 2 hours, which demonstrates that our analysis required less than 2 hours for more than half of the samples in both datasets. Although both numbers (2 and 4 hours) constitute significant computation time, we argue that the HML needs to be identified only once per each sample, facilitating any subsequent (security) analysis.

E. Attacks on Example Board

As discussed in Subsection V-D, to exemplify how one can use HEAPSTER to discover attacks on a real device, we leveraged the STM32-NucleoF401RE board expanded with an X-NUCLEO-IDW01M1 Wi-Fi Module, as depicted in Figure 5 and used HEAPSTER to find, first, the HML and, next, 3 possible attacks against the HML that successfully trigger Overlapping Chunks and Non-Heap Allocation. After that, we run the attacks on the real board, confirming all of them.

F. Firmware Clustering

Figure 6 depicts the results of our clustering algorithm discussed in Section V applied to both *ground-truth* and *wild* datasets. These results allowed us to establish a relationship between the two datasets in our evaluation, perform a deeper investigation of false negatives presence among the blobs with unidentified HML, and, re-use HML identification and security evaluation results for the new firmware samples if they can be assigned to an existing cluster.

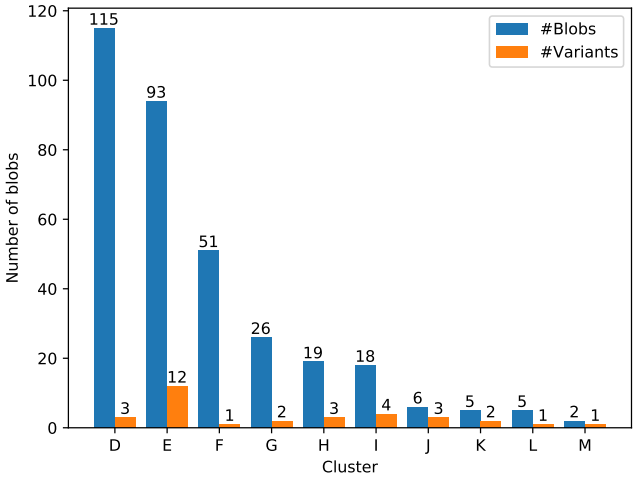

Fig. 4: In blue, number of blobs in each identified coarse-grained cluster in the *wild-dataset*. In orange, number of variants (i.e., sub-clusters) per identified coarse-grained cluster in the *wild-dataset*.

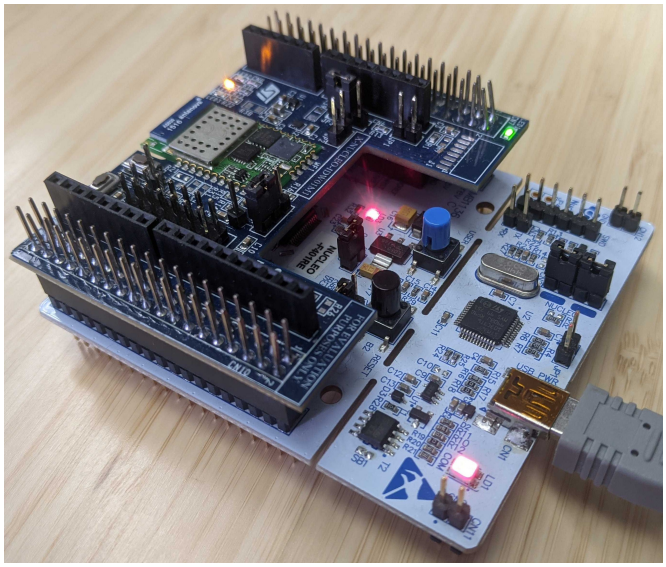

Fig. 5: The STM32-NucleoF401RE board (expanded with an X-NUCLEO-IDW01M1 Wi-Fi Module) used in our experiments.

	Stage	AT(sec.)	MT(sec.)	AM(MB)	MM(MB)
Ground Truth Dataset	①	13	10	219	207
	②	766	675	1482	1277
	③	2333	1928	3497	3031
	④	3526	3717	2206	2020
	⑤	12565	7233	1751	1555
	⑥	1858	405	2380	2412
	⑧	105	94	1456	1257
Wild Dataset	①	21	19	263	247
	②	2479	2003	2155	1719
	③	999	141	2873	1823
	④	8679	3627	4474	2224
	⑤	5808	136	2045	1602
	⑥	2999	89	2307	1629
	⑦	145	118	2186	1741
	⑧	121	95	2146	1555

TABLE V: Statistics for the HML identification. *Stage* corresponds to each analysis part number as described in Section III. *AT*/*AM* report average time/memory. *MT*/*MM* report median time/memory required for each stage.

G. Bounded Model Checking Configuration

When tracing a PoC with HeapHopper (as explained in Sections III-F and III-G), we use different parameters to bound the analysis. Without these limits, the analysis would quickly incur in state explosion, and, therefore, would not be able to provide any result. These are the main parameters used to configure the symbolic execution of the PoC:

Malloc sizes We use a limited number of default values (i.e., 8, 10, 20) as requested sizes when calling `malloc`.

Overflow sizes When modeling a heap overflow exploitation primitive, we specify the maximum amount of bytes that can be overflown. In our configuration, we use a value of 8 bytes.

Chunk header size This parameter is extracted by the analysis discussed in Section III-E. Practically, this indicates how many bytes of inline metadata are associated with each heap chunk.

Write target size When symbolically exploring the PoC, we handle attacker-controlled symbolic writes by concretizing addresses within a specific memory region we called "write target". The larger the memory area dedicated to the "write target" is, the more opportunities we have to concretize the values useful to trigger a heap vulnerability. However, the "write target" size increases together with the complexity (and number) of path constraints, drastically impacting the performance of the analysis. For our analysis, we use a value of 32 bytes.

Fake free chunk size When using fake-free as exploitation primitive, we execute `free` on a memory region filled with a configurable number of unconstrained symbolic variables. The bigger this memory region is, the more flexibility HeapHopper has to manipulate memory to trigger a vulnerability. However, similarly to the write target size case, this drastically impacts the scalability of the analysis. For this reason, in our evaluation, we limit the size of this memory region to 64 bytes.

Loop iterations When symbolically tracing a PoC, we set a limit on the number of iterations performed by every concrete loop. For our security evaluation, we set this value to 1000.

HML extra arguments As discussed in Section III-E, the prototypes of `malloc` and `free` may have extra arguments. In these cases, we constrain the arguments that are representing neither the requested size nor the pointer to free, to a list of possible concrete values. This effectively avoids the generation of false positives related to an unconstrained argument, but also ensures a precise execution of the allocator/de-allocator code.

Concretization strategies limits Numbers of possible solutions used when the HeapHopper's analysis concretizes a symbolic memory operation. We set this value to a maximum of 100.

Timeout We use this parameter to stop a symbolic analysis that becomes too expensive to trace. We set this value to 10 minutes per PoC.

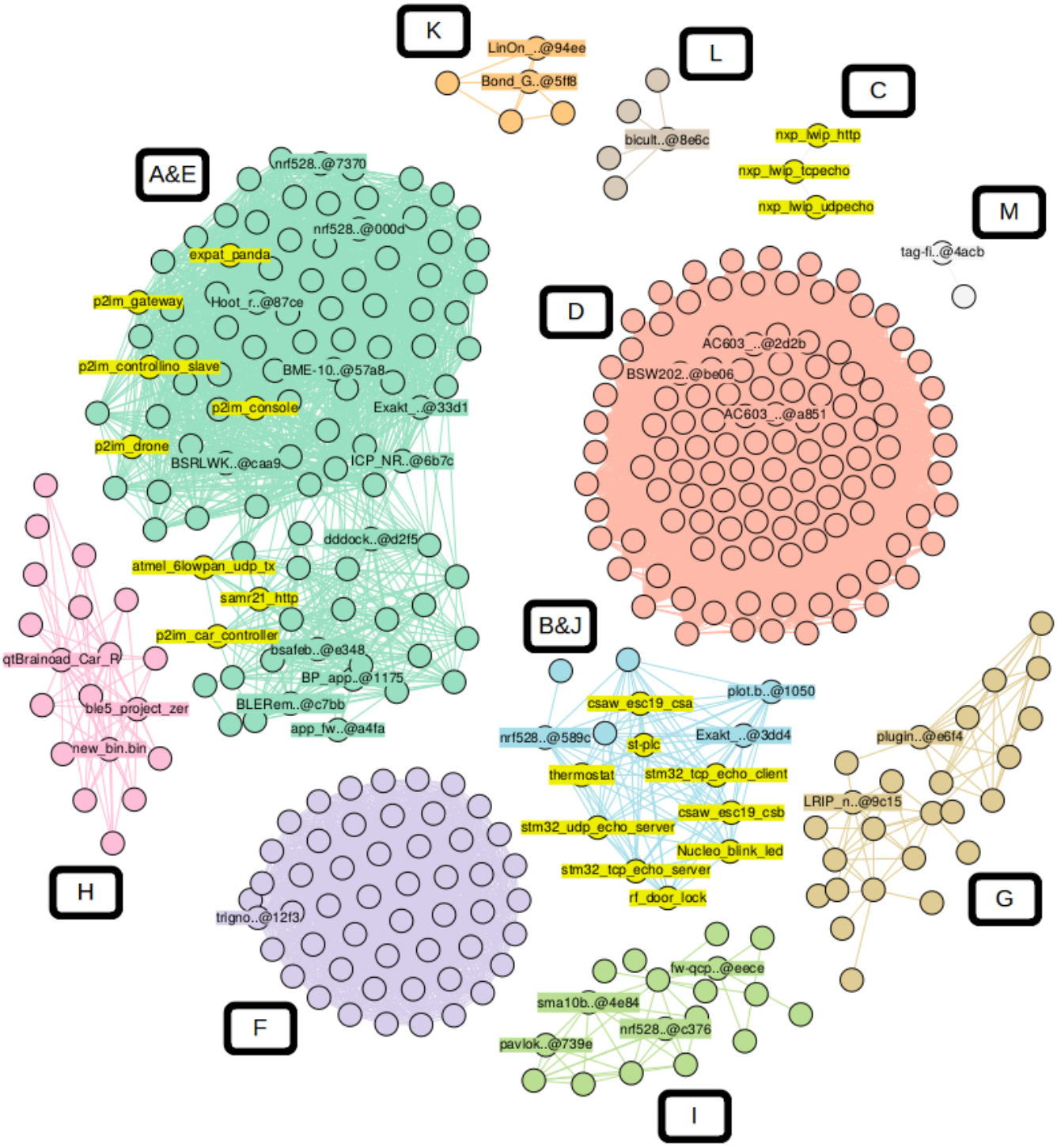

Fig. 6: Graph representing the similarity between the allocators discovered inside firmware samples in *ground-truth* and *wild* dataset. We show an edge between nodes only if the BinDiff similarity and confidence scores between the bodies of the respective `malloc` have values $\geq$0.7. Yellow nodes represent firmware of the *ground-truth* dataset. Nodes with labels represent the firmware blobs we tested with HeapHopper for the security evaluation (20 from the *ground-truth* dataset and 32 from the *wild* dataset).