An Automated Approach to Re-Hosting Embedded Firmware by Removing Hardware Dependencies

Austin Ketterer*ψ, Asha Shekar*†, Edgardo Barsallo Yi†, Saurabh Bagchi†, Abraham Clements‡
ψAMD, Fort Collins, USA. austinketterer@gmail.com
†Purdue University. West Lafayette, USA. {shekara, ebarsall, sbagchi}@purdue.edu
‡Sandia National Labs*. Alburquerque, USA. aacleme@sandia.gov

***Abstract*—Firmware emulation is useful for finding vulnerabilities, performing debugging, and testing functionalities. However, the process of enabling firmware to execute in an emulator (i.e., re-hosting) is difficult. Each piece of the firmware may depend on hardware peripherals outside the microcontroller that are inaccessible during emulation. Current practices involve painstakingly disentangling these dependencies or replacing them with developed models that emulate functions interacting with hardware. Unfortunately, both are highly manual and error-prone. In this paper, we introduce a systematic graph-based approach to analyze firmware binaries and determine which functions need to be replaced. Our approach is customizable to balance the fidelity of the emulation and the amount of effort it would take to achieve the emulation by modeling functions. We run our algorithm across a number of firmware binaries and show its ability to capture and remove a large majority of hardware dependencies.**

***Index Terms*—emulation, firmware, static analysis**

I. Introduction

In today's world, the utilization of embedded systems is almost guaranteed. As they become more connected, they are exposed to cyber-security concerns such as remote-exploitation. However, ensuring the correct operation and security of embedded systems is complicated by the tight coupling of the firmware with its hardware. Re-hosting firmware in an emulator enables deep inspection of a firmware's execution because every element of its execution can be monitored. Thus, firmware emulation can facilitate vulnerability detection and mitigation, embedded software testing, verification of safe behavior, and firmware analysis beyond the capabilities of physical hardware [15].

HALucinator [5] is a promising approach to embedded firmware re-hosting that leverages *high level emulation*. It provides hand-written proxy functions to replace functions used in embedded firmware that rely on non-standard hardware. Thus, the firmware can be run in an emulated environment and be inspected for vulnerabilities. However, this approach involves a considerable manual effort from the analyst and relies on their familiarity of the firmware to identify the parts that need to be modeled to decouple the firmware from the hardware and to create the proxy functions. This paper puts forth an automated approach for decoupling firmware from hardware when using high level emulation for re-hosting. This process is meant to inform emulation carried out by high-level emulators like HALucinator and QEMU. Interaction with hardware, for example, may cause trouble when the firmware is waiting on a hardware action to continue. With no hardware, the firmware may get stuck waiting forever. Decoupling a firmware from hardware involves identifying which of its functions cannot execute correctly without hardware. To date, this has been a tedious, manual, and iterative process — identify and replace a function with a stub and run the firmware to see if it exhibits the expected behavior; if not, replace some other function. We aim to change the process of identifying the functions to be modeled from an art to an automated solution that leverages dependency information to identify functions that should be modeled, removing a firmware's dependency on its hardware while balancing the effort required to model the functions.

Our solution is based on two algorithms for the firmware analysis: *Min-Cut* and *Simulated Annealing*. Both algorithms rely on the idea that firmware, at the function level, can be **partitioned** into a **retained** set and a **removed** set. The retained set contains the firmware's functions that would be retained in their original form for emulation, while the removed set contains the firmware's functions that should be removed entirely or replaced with models. A subset of the retained set is called the **must retain** set. These are nodes the user has required to be kept in the firmware. There is also a subset of the removed set called the **must remove** set. Functions in the must remove subset are required to be removed from execution as they include hardware dependent functions. The remaining functions can either be retained or removed depending on the results of our algorithms, placed in either the **suggested retain** subset or **suggested remove** subset. The output of the algorithms is a list of the removed functions and another list of the retained functions. The former list, called **removed set**, includes the functions that need to be modeled and replaced with proxy functions prior to emulation. While the task of creating models for the removed functions is still required, the manual task of iteratively running the emulation to identify where the firmware crashes is replaced with our algorithm that identifies the set of functions that needs modeled to satisfy the dependencies of the retained executable portions of firmware.

In summary, our contributions are as follows: 1) We provide an algorithmic technique to aid high level emulation in deciding which functions to model. This enables one to partition a firmware such that it is decoupled from its hardware

*These authors contributed equally.

dependencies yet retains its core functionality 2) We provide a policy-driven approach for users to indicate what part of the emulation needs to be performed with fidelity and what parts can be skipped. 3) We perform binary analysis on the firmware to determine the types of dependencies present and decide which need to be removed. 4) We show that our approach enables high level emulation of various firmware from Atmel and STMicroelectronics using HALucinator.

II. Background

Our approach relies on function call graphs (CFGs) and Data Dependency Graphs (DDGs) to represent the information recovered from the binary analysis. A CFG represents the known calls possible within a program. Each of the nodes are functions, and the edges represent a call from the source to the destination. Nodes and edges contain data about the functions and their relations to one another. We apply various weights to the nodes and edges to capture information about how difficult modeling a particular function would be and how removing it would impact the rest of the program.

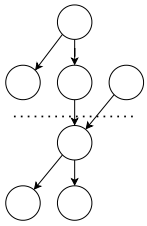

Fig. 1. Example CFG

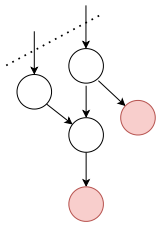

Fig. 2. Example of Utility Issue

Figure 1 shows the CFG of a section of a program. Nodes represent functions, and the arrows represent calls. The dashed line represents a cut being made. Our algorithms seeks to identify a cut that is the border where functions in the removed set will be replaced with models during emulation. In the example, we can see that four nodes remain on the top retained half, and three nodes are in the bottom removed half. The nodes that are called across the line are **border nodes**. By this cut, C and D will not call E but call a model built on HALucinator that will mimic E's behavior. Although three nodes are removed by the cut, E is the only one that has to be modeled as it is the only one called from the retained set. Therefore, when considering the effort to implement this cut in HALucinator, it is important to look at how many removed nodes are on the border rather than the total number of removed nodes. We now introduce different possible scenarios which were considered when designing both algorithms.

Trivial Case. The trivial case is to remove all nodes interacting with hardware (i.e., all functions that read/write hardware registers are put in the must remove set). This ensures the program is decoupled from hardware so that we may emulate it. Therefore, if G was the only node in the graph we knew interacted with hardware, we could cut to just remove G.

Loss of Functionality. In the previous scenario, we simply cut out any node that interacts with hardware. If we are willing to sacrifice functionality, however, there are times when we can make cuts that remove more nodes but reduce the number of nodes on the border. This results in fewer functions to model, hopefully leading to less implementation work.

Global Write. Let us say we have the scenario where we want to cut E, G, and H as we decided to sacrifice E functionality. In this case, a simple model for E could serve as a fine replacement. However, if E makes changes that affect the program outside the scope (e.g., E writes to global memory that is used in another part of the program), this may have major repercussions on the flow of the program. If the global write is read from another function, say B, in the retained set, we need to ensure the model for E properly initializes the global structure (i.e., increasing the complexity of the model) or to remove B from the emulation. Thus, our models become simpler if all functions dependent on external data are retained or removed. We seek to capture this dynamic in our algorithms.

Pointer Write. Finally, many functions take a pointer and write to the address to which it points. Determining this address can be difficult; however, we can follow a similar approach to the previous scenario for constant pointers. If E receives a pointer to data in B, we can remove the dependency by removing both. Here, we want to consider E's writes while also monitoring if E calls any other functions and passes on the pointer where it could be written there.

III. Design & Implementation

We now explain how our algorithms partition a firmware into the retained set and the removed set in order to facilitate successful firmware emulation.

A. Cut-Sets

We classify the part of the program that must be removed as the **must remove** set. These are the set of nodes that interact with hardware and are likely to cause issues with emulation. Although breaking all hardware dependencies may not be needed for firmware re-hosting (e.g., a read value is not used), our algorithms target the removal of all known hardware dependencies as this will ensure reliable behavior during emulation.

Similarly, there is also a **must retain** set. While not required, our algorithms allow the user to place functions in this set to specify that they must be retained in all generated solutions. However, to guarantee the **must retain** set is kept, it must be reachable from an entry point through a path of **must retain** nodes. Suppose a node is in **must retain** but is only reachable through nodes not in must retain, then it may be removed as a result of those nodes' removal. The Min-Cut algorithm may generate a solution that removes all the callers of the must retain nodes; thus, while the nodes remain in the retain set, they will be unreachable during execution. To prevent this, a shortest path between the entry point and **must retain** nodes not otherwise connected is added to the must retain set to ensure they are not indirectly cut.

All nodes not included in a "must" set are then classified by the algorithm as either **suggested remove** or **suggested retain**. Different trade-offs come with deciding to optionally remove a node. One situation in which removing a node makes the program easier to emulate is when it reduces the total number

of functions that need to be replaced. For example, there may be one function that is not hardware dependent (e.g., not in must remove set) that calls three separate hardware dependent functions. If we decide to remove hardware-dependent nodes, we would need to replace three functions. Instead, cutting one level above requires only replacing a single function and removes the same dependencies. Another situation where an optional cut is beneficial is if it removes a global dependency. In these cases, eliminating the function may reduce the risks of having problems due to improper model implementations during the emulation. An engineer may have two major goals in trying to emulate firmware: to have as much functionality retained in the original firmware as possible and to get this emulation done promptly. These two goals are at odds as more accurate emulation generally requires more time. These are the two metrics that our algorithms attempt to optimize.

B. Preprocessing

To analyze the firmware, we use the reverse engineering software Ghidra. From this analysis, we obtain information about a function's parameters, calls, writes and reads. The calls are used to establish a call graph, while the writes and reads are used to build a DDG. However, some issues may arise due to the challenges of alias analysis and errors in Ghidra. For example, Ghidra may miss identified targets of calls due to the handling of ARM Thumb instructions. In Thumb instructions, calls are made with odd addresses as the least significant bit indicates to execute the calling function in Thumb mode. Thus, Ghidra does not realize that a call to a `function pointer+1` is not a call to a separate function but a call to the function at function pointer. Therefore, when a `pointer+1` is stored in a data structure and then called later, Ghidra did not identify a call for this. To fix this issue, we develop a script to establish an edge between all functions that use that portion of the data structure to the function stored within. This may be an overestimate, but a few major connections in the graph were missing without this analysis.

Hardware Identification. The DDG provides information regarding the list of addresses, their lengths, and the functions that have *read from* or *written to* that location. Assuming the firmware's intended board is known, the information about the memory map can be found. From this information, hardware write and reads, as well as global writes and reads, can be identified by their address.

Escape Analysis. Another valuable input is knowing the dynamic scope of the functions, i.e., the function either generates or uses data that persists after the function returns. Telling where all possible memory locations to which a pointer could point is difficult. We use Ghidra to determine if a write is made to pointers that are not in the current function's context. We also look for pointers that are passed in and subsequently passed as arguments to functions.

C. Min-Cut Based Algorithm

Based on the Edmonds-Karp Min-Cut algorithm [8], this algorithm is able to separate two nodes by removing the directed weighted edges that connect them with minimal cost. In this particular use case, the **source** node points to all entry points and interrupts of the program, and the **sink** node is pointed to by all hardware interactions which need to be removed. The intuition behind this algorithm is that the min-cut algorithm could provide the optimal solution by weighting the functions based on how difficult they would be to remove and how much benefit they provide by being retained. The difficulty of replacing a function is only based on the effort to implement the script itself (that will emulate the function) and is not affected by the number of invocations. However, the algorithm outputs the edges, not nodes, that need to be removed. Hence, to overcome this, we assign each call edge an infinite weight and turn each node into a pair of nodes with a weigthed edge between them. This edge corresponds to the weight of the function, and when it is cut, it suggests that the given function should be removed. The formula for establishing a node's weight is as follows:

$$weight = \mu \times effort + (1 - \mu) \times utility \tag{1}$$

The **effort** is established by looking at the difficulty of the function's implementation. This difficulty stems from how the function could escape its local scope. If a function is self contained, it is much easier to cut than one which affects the rest of the program. Escaping the local scope can be achieved by making a write to the global memory space or to a pointer. If either of these two situations occur, the node's weight is increased. The specific values to be used for any of these weights is up to the user, as the difficulty to overcome this issue is subjective. However, this metric is flawed in Min-Cut since each node needs a set weight before the cut determination begins. Further, how the effort should be weighted depends on the cut. An example is if function A writes to a global that is only read by function B. If only A is cut, we are left with a now missing global dependency. If A and B are cut, there is no global dependency that needs to be fixed. Therefore, knowing the cut is essential to properly weighting A, but the weights cannot be established after the cut in Min-Cut.

Based on the description of **utility** (i.e., valuing how much of the program is kept) the nodes should have a weight that describes how many nodes will be lost if a cut is made at that location. The following examples are based on Figure 2. If just B needs to be cut, then B should have a weight of 2, accounting for itself and the hardware (HW) node to which only it points. If just A should need to be cut, it should have a weight of 1 just accounting for itself. If A and B are cut, the result would be 3 but with them both gone a total of 5 nodes are lost. Due to these situations in which the proper weight is only known after the cut, an estimate of utility can be made but it cannot be done perfectly.

To serve as an estimate for utility, a metric calculates roughly how many nodes are between the given node and the sink node. To do this a depth-first search is done, and then the nodes are iterated through in a post-order traversal. Each node that points at the sink directly is given a utility of 1. For nodes that do not point at the sink, their utility value is

the added utility of the nodes to which that node points. For example, if a node had two outward edges, one to a node with utility 2 and one to a node with utility 1, its own utility would be 3. As stated earlier, a perfect calculation is impossible for min-cut, but this gives an estimate of each node's utility.

These weights are then added together, and, as with both factors, an increase in value implies it would be preferable not to cut at that location. If a function takes more effort to remove or has a great benefit in being retained, it would be best not to cut it. The μ constant, a value between 0 and 1, allows for bias towards one of the factors. Adjusting this allows to focus on achieving the most easily implementable solution or the solution which retains the most functionality while still removing all directly hardware dependent functions.

D. Simulated Annealing (SA)

Simulated Annealing, inspired by the annealing procedure of metal working [13], was devised as an effort to improve upon the combinatorial optimization heuristic of iterative improvement. SA starts with an initial state as the current state and as the best state and an initial maximum temperature as the current temperature. While the current temperature is greater than 0, the following process occurs. A potential neighboring state (where a border node is moved across to the other set) is chosen based on a **move function**. The energy (or cost) of that neighboring state is calculated using the **objective function**. The difference ($\triangle E$) between the energy of the neighboring state and the energy of the current state is found. If $\triangle E \leq 0$ (the chosen neighboring state is better than or the same as the current state), the neighboring state is set as the current state and best state. Else if $\triangle E > 0$ (the chosen neighboring state is worse than the current state), then the probability that the neighboring state is accepted is P($\triangle E$) = $e^{\frac{-\triangle E}{T}}$, where T is the current temperature. A random number r is chosen from the interval (0, 1). If $r < P(\triangle E)$, then the neighboring state is set as the current state. If $r \geq P(\triangle E)$, then the current state is not changed. The current temperature is updated based on the **cooling schedule**. Once the current temperature reaches 0, the system "freezes," and the best state found is the solution output. Our implementation of SA involves defining how a state represents the program, how the optimized parameter is calculated, how neighboring states of a current state are determined, and how current temperature T changes as the algorithm progresses.

States represent a certain configuration of the whole firmware. Thus, a state is defined by how we have divided the firmware's functions: the sets must-retain, must-remove, suggested-retain, and suggested-remove. Functions in the suggested-retain set have the potential to be moved to the suggested-remove set and vice versa. Our initial state is the output of the Min-Cut algorithm, with the goal of determining if there are potential "better" solutions near the solution provided by the Min-Cut algorithm.

An **objective function** is used to calculate the energy (E), the optimized parameter, of a state. E is calculated as the weighted difference between the total effort and total utility of a given state. The total utility is calculated as the number of functions that are retained, as that best represents the utility of the state relative to the full utility offered by the entire firmware. The total effort represents how difficult it would be to account for the removed functions to make the firmware runnable. The total effort is calculated as the sum of the number of functions in the removed set that are called from any function in the retained set and the number of writes to a global variable in the removed set that are read by the retained set. A user-determined weight is applied to the effort and utility. $w_e + w_u$ = 1. Therefore, if preference wants to be given to effort required to make the firmware runnable, then $w_e > w_u$, whereas $w_u > w_e$ if preference to utility of the firmware is desired. The objective function is as follows:

$$(w_e \times effort) - (w_u \times utility) \tag{2}$$

IV. Evaluation

Methodology. We evaluated our approach over a series of firmwares sourced from different vendors: a) STM32-Cube HAL library for STM32F4 boards ("STM-") [2]; b) a selection of programs for the STM Nucleo chosen to use in Pretender [10]; c) Atmel's Advanced Software Framework for SAM R21 boards("Atmel-") [1]. Through our evaluation, we show that as we vary the weighting between utility and effort, our approach comes up with viable solutions that scale as planned—which also meet the basic requirement of removing all **must remove** nodes and retaining all **must retain** nodes. The complete set of results can be found at https://github.com/purdue-dcsl.

Variation of Utility Weight. Figure 3 shows how the Min-Cut algorithm performed over variation of the utility weight. Only the utility weight is shown as the effort weight is implied to be $1 - utility$. The important factors to look at are the performance of the cut at 0 and 1. A cut at 0 represents a graph purely weighted by effort, while a cut at 1 is purely weighted by the utility. As shown in the figure, as the weight utility get closer to 1, more nodes are retained. Note that the percentage is shown as the number of suggested nodes retained. These nodes can be either removed or retained. Therefore, the value of the utility increases as the number of nodes retained.

Although the trend in Figure 4 is that the number of border nodes increases as weight utility, this does not necessarily need to be the case. As the number of border nodes does not purely calculate weight effort, it can vary in either direction as weight utility increases. A few nodes that do not escape their context may be less work to implement than single nodes that write to many globals. For example, Atlem-SD-FatFS begins with 10 border nodes in a pure effort weighting and then drops to 8 border nodes when the utility increases.

Pointer Enhancement. Fixing the Thumb mode pointer issue did not significantly increase the number of reachable functions in the call graph as shown in Table I. The labels in the table are Total Fxns, the total number of functions in the binary, Reachable, for all functions that were found to be statically reachable from the initial Ghidra analysis, and Enhancement, for all nodes that were found to be reachable after

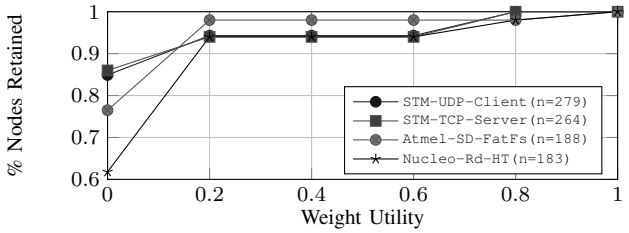

Fig. 3. Effect of W_u on % Nodes Retained (Min-Cut)

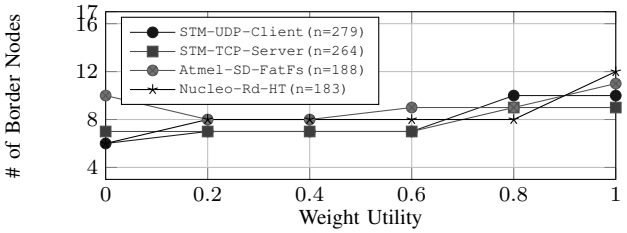

Fig. 4. Effect of W_u on # of Border Nodes (Min-Cut)

the enhanced pointer analysis. A few binaries did see sections that were missing in the call graph get attached through new edges, but many functions are still never reachable. This can be attributed to two possible causes: 1) the dynamically computed calls are still missing from the call graph; 2) as these binaries were compiled, library functions that are unused were included (e.g., dead code). If the compiler chose not to optimize them out, then they may be included in the binary even if they are unreachable in the normal flow of the program. To test how compilation affected the results, The FatFS file was tested at O0 – which was used for all other examples as well – and also O2 as a comparison. When increasing the optimization level, the number of both functions reachable and unreachable both dropped. This can be explained by potential for functions to be inlined and more aggressive removal of dead code. However, it is possible both dynamic calls are used which could not be identified statically are still present and also some dead code is still present.

	Total Fxns	Reachable	Enhancement
STM-UART	71	57	57
STM-FatFS O0	277	157	157
STM-FatFS O2	222	122	122
STM-UDP-echo-client	464	257	279
STM-UDP-echo-server	459	247	269
STM-TCP-echo-client	473	259	267
STM-TCP-echo-server	474	256	264
STM-plc	994	310	338
Atmel-SD-FatFS	47	40	40
Atmel-LwIP-HTTP	223	188	188
Atmel-6LoWPAN-Sender	483	155	168
Nucleo-blink-led	189	142	142
Nucleo-read-hyperterminal	304	175	183
Nucleo-rf-door-lock	335	185	193
Nucleo-thermostat	304	175	183

TABLE I
POINTER ENHANCEMENT BENEFIT (# OF FUNCTIONS)

Simulated Annealing. To compare the benefits of SA, we use the results from the Min-Cut as a baseline. Table II shows which solutions turned out to be better or equal after running SA. Only in a handful of situations, SA was able to find a better solution, showing that min-cut can be improved. Some reasons that explain this include: a) the Min-Cut was able to find the best solution instantly most of the time; b) SA was not given enough time to run and explore. Table III shows the runtime in seconds of Min-Cut and of SA at different # iterations ran. Min-Cut is a much simpler algorithm that runs faster than even SA at its lowest # iterations. As SA iterates, its runtime grows linearly. Given that the total number of combinations that n nodes could reach between having sets of retained and removed nodes is 2^n, it is possible that the # of iterations was not high enough, and better solutions may appear at higher # iterations. We can conclude that from 10K to 100K iterations, SA could not find a better, less costly solution.

	0	0.2	0.4	0.6	0.8	1
STM-UART	✓	-	-	-	-	-
Atmel-UART	-	✓	✓	-	-	-
Atmel-SD-FatFs	-	-	✓	-	-	-
Nucleo-blink-led	✓	-	-	-	-	-
Nucleo-Rd-HT	✓	-	-	-	-	-
Nucleo-rf-door-lock	-	-	-	✓	-	-
Nucleo-thermostat	✓	-	-	-	-	-

TABLE II
SA GENERATED BETTER SOLUTIONS

	RT
Min-cut	0.19
1000	1
10000	9
100000	98

TABLE III
RUNTIME OF MIN-CUT VS SA VARYING # ITERATIONS

Basic Block Coverage. Next, we look at the total number of basic blocks that are reachable from our generated solutions in Table IV. This gives an understanding of how much of the original firmware we are removing with our cuts and how much is retained. The table shows: **TTL** or total # of basic blocks of the binary; **RCH** or the # nodes that appear to be reachable from a static analysis; **PRTN** or the # of basic blocks that PRETENDER was able to reach in its runs; **IDL** or the ideal # of basic blocks for the best cut solution (fully effort weighted/fully utility weighted); and **DYN** or dynamic analysis. Regarding **PRTN**, recall that PRETENDER uses QEMU to determine basic block counts, so it is possible that Ghidra and QEMU define basic blocks differently. Further, PRETENDER also dynamically measured their basic blocks; thus, their number is highly dependent on the inputs given to the system. Finally, the dynamic analysis has only been done for one basic program. Although it was originally intended for all the programs, we revised this decision. Making ideal models and input for these systems is time intensive, and the ideal is already known. Further, it is also not fully known how many should be able to be hit if the firmware is being used as intended. A HAL function, for example, could support five different ways of using it, but if it is only being used for part of its functionality, it will likely never hit certain blocks.

	TTL	RCH	PRTN	IDL	DYN
STM-UART	497	469	✗	246/256	66
STM-FatFS	2296	1687	✗	1416/1416	✗
STM-UDP-echo-client	5682	3935	✗	2110/3600	✗
STM-UDP-echo-server	5240	3436	✗	2046/3111	✗
STM-TCP-echo-client	5905	3444	✗	2041/3106	✗
STM-TCP-echo-server	5495	3398	✗	2008/3073	✗
STM-plc	9371	4080	✗	3566/3701	✗
Nucleo-blink-led	2165	1916	218	573/1580	✗
Nucleo-read-hyperterminal	2781	545	828	658/1847	✗
Nucleo-rf-door-lock	2840	1992	665	1615/1770	✗
Nucleo-thermostat	2781	2183	1263	658/1847	✗

TABLE IV
BASIC BLOCK COVERAGE OF SOLUTIONS

Affecting Resulting Outcome. Here we present an example

on the UDP-echo-client firmware to show how manually putting nodes in the must retain and must remove sets can affect the overall outcome. The results are shared for the Min-Cut and SA algorithms. First, at a utility weight of 0.2, the initial cut suggested by the algorithms was to cut at a border comprised of 6 **must remove** nodes and a single **suggested remove** node, being the BSP_Config function. Then, we added the BSP_Config function to the must retain set. After doing so, the cut comes out to a border of 10 **must remove** nodes. This shows that the suggested cut of removing the BSP_Convig function saved the work of implementing one node on the border instead of 4. To test manually adding nodes to must remove, a random function, User_Notification was added. Previously it was not included in the cut, but after being put in must remove, it was then included as a suggested border node to be cut.

V. Related Works

Avatar2 [11], PROSPECT [12], and SURROGATES [14] employ HITL (Hardware-in-the-Loop) to emulate system execution. This method is beneficial since it makes emulating peripherals unnecessary, thus overcoming the challenge with high confidence. However, the drawback of HITL is that requiring physical hardware during emulation execution, or even worse, requiring specialized embedded hardware, limits scalability and increases the emulation's effort, ultimately limiting its potential for application. Meanwhile, Pretender [10] uses a ML-based technique, where physical hardware is used to train peripheral models during emulation setup. Although this technique takes less effort, the accuracy of execution and data fidelity suffers as it is hardware dependent.

Costin Firmware Analysis [6], [7], Firmadyne [4], P2IM [9], and HALucinator [5] are some of the emulators that do not require the use of physical hardware during the process of attaining emulation. Not requiring specialized hardware allows the techniques to be scalable and reduces the effort of having to configure the technique to each specific hardware. However, not being able to rely on the physical hardware for emulation implies that the firmware needs to be patched up to handle any hardware accesses. All four of these techniques use QEMU [3] as its base emulator, which does not handle peripherals being accessed by firmware. The methods among these techniques for handling peripherals vary, but the common aspect is that patching up the firmware to handle hardware access is a manual process that takes time and effort. Contrary, our proposed approach aids to reduce the effort invested by the developer patching the firmware.

VI. Limitations

Our approach is based on the information that Ghidra is able to provide about the control flow and data dependencies. This base is, as of now, imperfect. Any improvements to this process would help create a more accurate result. Furthermore, the Min-Cut algorithm has limitations in that it can only estimate the level of effort and utility and cannot give a true measurement. SA is much more capable but is limited by the amount of space it can search, so it has no guarantee of finding the best solution in the entire search space.

VII. Conclusion

Through this effort, a formal approach to removing dependencies was started, creating a guided method through which hardware dependencies can be removed more easily to facilitate firmware re-hosting. Through binary analysis, users are able to know which functions to remove to eliminate hardware dependencies. They can tailor the algorithm through use of retained and removed sets and a variable that instantiates the trade-off between max utility and min effort. This method has been shown to work for a few micro-controller firmwares. These solutions have been shown to provide an implementation that will allow for a large amount of the original binary to be analyzed in a completely virtual environment.

References

[1] "Atmel advanced software framework," 2021. [Online]. Available: http://asf.atmel.com/docs/latest/architecture.html

[2] "STM32Cube MCU packages," 2021. [Online]. Available: https://www.st.com/en/embedded-software/stm32cube-mcu-packages.html

[3] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conference)*, Apr. 2005.

[4] D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware." in *NDSS*, 2016, pp. 1–16.

[5] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "HALucinator: Firmware re-hosting through abstraction layer emulation," in *29th USENIX Security Symposium*, 2020, pp. 1201–1218.

[6] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A Large-Scale analysis of the security of embedded firmwares," in *23rd USENIX Security Symposium*, 2014, pp. 95–110.

[7] A. Costin, A. Zarras, and A. Francillon, "Automated dynamic firmware analysis at scale: A case study on embedded web interfaces," in *ACM ASIA CCS*, 2016, p. 437–448.

[8] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *J. ACM*, vol. 19, no. 2, p. 248–264, apr 1972.

[9] B. Feng, A. Mera, and L. Lu, "P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *29th USENIX Security Symposium*, 2020, pp. 1237–1254.

[10] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. Vigna, "Toward the analysis of embedded firmware through automated re-hosting," in *RAID*, 2019, pp. 135–150.

[11] Z. Hu and B. Dolan-Gavitt, "Irqdebloat: Reducing driver attack surface in embedded devices," in *IEEE Symposium on Security and Privacy*, 2022, pp. 1465–1479.

[12] M. Kammerstetter, C. Platzer, and W. Kastner, "Prospect: Peripheral proxying supported embedded code testing," in *ACM ASIA CCS*, 2014, p. 329–340.

[13] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[14] K. Koscher, T. Kohno, and D. Molnar, "SURROGATES: Enabling Near-Real-Time dynamic analyses of embedded systems," in *9th USENIX Workshop on Offensive Technologies*, 2015.

[15] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements, "Challenges in firmware re-hosting, emulation, and analysis," *ACM Comput. Surv.*, vol. 54, no. 1, jan 2021.