Reconfiguration of Legacy Software Artifacts on Resource Constraint Smart Cards

Daniel Baldin, Stefan Grösbrink, Simon Oberthür

Design of Distributed Embedded Systems
Heinz Nixdorf Institute, University of Paderborn
Fuerstenallee 11, D-33102 Paderborn, Germany
dbaldin@upb.de, stefan.groesbrink@hni.upb.de, oberthuer@upb.de

***Abstract*—Today's adaptable architectures require the support of configurability and adaptability at design level. However, modern software products are often constructed out of reusable but non-adaptable legacy software artifacts (e.g., libraries) to meet early time-to-market requirements. Thus, modern adaptable architectures are rarely used in commercial applications, because the effort to add adaptability to the reused software artifacts is just too high. In this paper, we describe a methodology to semi-automatically use existing binaries in a reconfigurable manner. It is based on building the annotated control flow graph to identify and extract code on static basic block level depending on different execution requirements given as a set of constraints. This allows for adaptation of binaries after compile time without the use of the corresponding source code. We propose a way of adding additional reconfiguration support to these binary objects. With this approach, reconfiguration can be added with a low effort to non-adaptive software.**

Keywords*—*Reconfiguration; Legacy Software; Smart Cards

I. INTRODUCTION

Software developers often use existing pre-compiled software libraries for various reasons. One reason may be the reduced development time by using third party libraries. Sometimes the use of third party hardware components may also require the use of so called board support packages. In other cases the reason for using pre-compiled libraries may even be as simple as missing source code or documentation. While using these libraries greatly eases the development of new software products, they may also be a source of problems in very resource constraint embedded systems.

Runtime reconfiguration can be the enabling technology for these kind of embedded systems such as Smart Cards by allowing temporarily unused functionalities to be replaced by currently needed functionalities. However the use of pre-compiled third party libraries limits the reconfigurability of the system. State of the art approaches try to solve this problem by wrapping the whole legacy library into a reconfiguration component, leading to a huge waste of memory. Thus, if we want to efficiently use existing libraries inside a reconfigurable system, which cannot be modified at source code level and contain huge amounts of unused or rarely used code, a new approach is required.

In this paper, we introduce a methodology which semi-automatically adds reconfigurability to binary objects using a set of constraints which specify reconfiguration points by high level expressions. The approach is based on creating an annotated control flow graph of the binary on static basic block level and requires only minimal source code information. Specifically, we analyze method signatures to identify higher level expressions that are used for the identification of reconfiguration entry points of the software. The availability of method signatures is only a small restriction since even proprietary libraries include header files containing structure and method signatures describing the Application Program Interface (API) of the library. If this is not be the case, the entire library would not be usable by any higher level programming language as the interfaces would be unknown.

The remaining paper covers the overall methodology of our reconfiguration framework implemented for the ARMv4 Instruction Set Architecture (ISA) in detail, starting with the basic techniques used, followed by the component model, the identification of components, optimizations and concluding with an explanation on the modifications of the original system. Our case study, the evaluation section is based on, focuses on an Internet-Protocol Stack library for an ARM powered Smart Card containing protocol implementations for IPv4 [1], IPv6 [2], TCP [3], UDP [4] and TLS [5]. The scenario contains a web-server which offers communication ports using all of these protocols of the library. However, at runtime not all protocols are used at the same time, which makes it interesting to use the corresponding protocols as reconfiguration components. The paper concludes with related work and outlook.

II. METHODOLOGY

Our approach allows the use of code from legacy libraries as well as fine-granular reconfiguration without the drawbacks of current state of the art approaches. Common approaches either do not allow legacy libraries to be used or simply wrap the complete legacy library into one huge component. This however is not practical for very resource constraint systems. Libraries are typically not given as high level code which might be rewritten for reconfiguration support. Thus, a low level method to extract components out of these libraries and to add reconfiguration support to them is needed. Forcing the

 ISBN: 978-1-61208-229-5

user to do this manually is something that is highly undesirable as well as often impractical as the expert knowledge required to do this cannot be assumed to be available. With this in mind the approach proposed in this paper focuses on the following requirements:

- Usability: Converting parts of the legacy code into reconfigurable components shall be supported by an automated tool that supports to configure the system parameters.
- Run-Time Efficiency: Component loading and replacement shall be as simple as possible without any need of linking the components at run-time. The execution overhead at runtime shall be kept as small as possible.
- Correctness: The semantics of the legacy code must not be changed.

All of these requirements are covered by the approach described in the next sections. The usability is improved by the use of an automatic binary analysis step in combination with the possibility of allowing the user to specify components with high level constraints. Run-time efficiency is achieved by minimizing the overhead of the run-time reconfiguration approach by statically resolving dependencies and by optimizing the components based on parameters as memory and binary overhead, as well as the worst case number of reconfigurations at runtime. The correctness is ensured by the use of instrumentation code which does not change the context of the application. The overall approach is depicted in Figure 1. The approach uses the binary objects, a reconfiguration manager including a replacement policy and a configuration file as the input. The first step is the *binary analysis* of the legacy objects which is covered in the next section. Some of the steps of the approach are ISA specific. In this paper, we will focus on the ARMv4 ISA as our evaluation platform is an ARMv4 powered Smart Card.

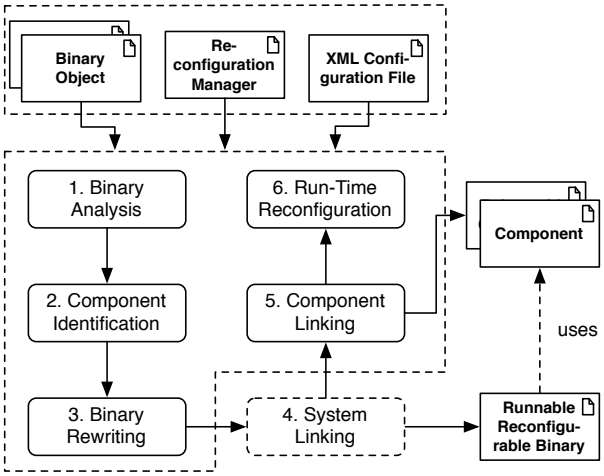

Fig. 1. Steps of the Reconfiguration Approach

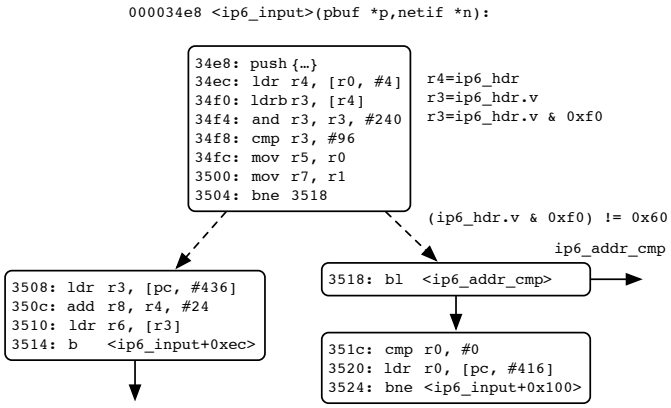

Fig. 2. Parts of the annotated control flow graph of the ip6_input method.

A. Binary Analysis

By disassembling the binary code the static basic blocks and the control flow between these blocks of the program are identified. A static basic block is a sequence of instructions that has exactly one entry point and one exit point. We use the basic block as the smallest representation unit since it describes a linear flow of instructions. A non-linear control flow appears only at the end of a basic block. Each instruction that is a target of a branch instruction defines a new basic block. In general, every program can be uniquely partitioned into a set of non-overlapping static basic blocks.

Figure 2 depicts the first four basic blocks of the disassembled `ip6_input` method. Using these blocks a graph representing the possible control flow of the processor as seen in the Figure is derived. This graph is called the Control Flow Graph (CFG). Each node defines a basic block and the edges represent conditional control flow (dashed edges) and unconditional control flow (solid egdes) between these blocks. Each control flow edge models a dependency between the basic blocks, as reaching one basic block means that we may also reach the successors of it.

The analysis of binary code is a non-trivial task. While disassembling and interpreting binary files, one may encounter several problems as, e.g., the Code Discovery Problem. Many ISAs allow binary data to be mixed up with executable instructions and vice versa. Not being able to distinguish between instructions and data may invalidate the extraction process since some control flows may not be discovered or data may be misinterpreted. However, for our evaluation platform this problem does not exist, since the ARM Embedded Applications Binary Interface (EABI) forces all EABI conform Embedded Linker File (ELF) object files to provide information on all occurrences of data and instruction blocks by special mapping symbols inside the symbol table (see Section 4.6.5 in [6] for the symbol definition).

Another problem with control flow detection arises if indirect control flow instructions are used inside the binary. Most of the indirect control flows are due to jump tables that are generated by the compiler to speed up switch/case statements. The targets of these jumps can be computed with high precision as it was shown by Cifuentes et al. [7]. Other sources of indirect control flows are method pointers, available in most high-level languages, e.g., to implement inheritance

 ISBN: 978-1-61208-229-5

or to realize dynamic program behavior. The targets of these kind of indirect control flows are very hard to compute and to the best of our knowledge no approach exists which can guarantee the precise detection of all targets. However, using the approach proposed by B. Sutter et al. [8] we may overestimate the set of jump targets by introducing a so called *hell node*. The estimation uses the complete set of relocatable symbols, which is the union of all relocatable symbols of all object files, as the target for every indirect jump that can not be resolved. The result may not be as tight as possible but it ensures the correctness of the following reconfiguration process.

In the next step, the edges of the graph are annotated using the common approach of forward substitution. As described by previous work of Cifuentes et al. [9] [10], we derive complex expressions from low level expressions, which in our case are the assembler instructions of the ARM binary. For assembly code one can express the contents of a register r in terms of a set a_k at instruction i as $r = f_1(\{a_k\}, i)$. If the definition at instruction i is the unique definition of a register r that reaches an instruction j along all paths in the program, without any of the registers a_k being redefined, one can forward substitute the register definition at instruction j with $s = f_2(\{r\}, j)$, resulting in:

$$s = f_2(\{f_1(\{a_k\}, i)\}, j)$$

Using these expressions it is also possible to annotate the edges of the control flow graph with constraints that need to be fulfilled for the edge to be taken. However, these expressions consist only of very low level type of operations and resources as, e.g., binary operations and registers. In order to allow these expressions to be used by some developer, it is important to derive as many high level programming language expressions out of the low level expressions as possible. We developed a binary analysis framework [11] which utilizes the high level information stored inside header files to extract type information on the input parameters and global variables of the binary objects. Using global data flow analysis techniques it is possible to annotate parts of the control flow graph with high level constraints based on the input parameters of the binary objects. The result of this analysis has partially been annotated next to the corresponding instruction in Figure 2. However, detecting access to high level data structures is not trivial as an unlimited number of access possibilities to these data structures can be generated by a compiler. In consequence, an expression normalization step as described in [11] is mandatory to allow a meaningful and usable annotation of the binary code.

B. Component Model

After the binary analysis, the binary objects are represented by its CFG $G = (N, E)$, with N being the set of nodes (static basic blocks), E the set of edges and $S \subseteq N$ the set of start nodes (entry points). The function $c : E \rightarrow C$, with C being the set of all constraints, matches every edge to its specific constraint that has been calculated in the previous step. We now need to identify sets of basic blocks inside the CFG which we may use as components inside the reconfiguration process. With a specific input language, the user is able to specify constraints on variables or method parameters used inside the binary objects. An example for such an constraint is given in Listing 1, which has been used for our evaluation scenario. Constraints are either specified for API functions or globally visible symbols. The former ones can contain arbitrary binary operations as the constraints in line two, four and six of Listing 1. The latter ones directly define reconfiguration points as the constraint in line eight. The constraint set is part of the configuration XML file, which is given as an input parameter to the reconfiguration framework, as shown in Figure 1.

```
1 [ip4_input]
2   (ip4_header._ttl_proto & 0x00ff) != 0x06
3 [ip6_input]
4   ip6_hdr.nexthdr != 0x6
5 [ethernet_input]
6   eth_hdr.type != 0x86dd
7 [tls_input]
8   @tls_input
```

Listing 1. Constraints set masking the TCP, IPv4 and IPv6 support as components

For every edge $e \in E$ of the CFG, we then combine the edge constraint $c(e)$ and the corresponding constraint of the user by a logical *and* operation. Our framework uses a constraint solver, which tries to check the satisfiability of the expressions. The set of edges, for which the expression is unsatisfiable, defines the set $R \subseteq E$ that we call set of reconfiguration edges. For unsatisfiable expressions, there exists no assignment of values that satisfy the expression. Our framework implementation currently allows different constraint solvers to be used. For our evaluations, we utilized the STP Constraint Solver [12].

Using the reconfiguration edges of set R, it is now possible to define some important sets of basic blocks, which will be used for our component model throughout the rest of the paper.

Definition II.1 (Mandatory Set)**:**
We define the set M of nodes that can be reached from the start nodes S without taking any reconfiguration edge as $M = \{n \in N : \exists w = (w_1, w_2, ..., w_n), w_1 \in S \wedge (w_i, w_{i+1}) \in E \setminus R \wedge w_n = n\}$. This set defines the set of basic blocks that we call the Mandatory Set.

Definition II.2 (Intermediate Components)**:**
For every reconfiguration edge $r_i \in R$ with $r_i = (n_{i1}, n_{i2})$ we define the set N_{r_i} of nodes that can be reached over the reconfiguration edge r_i without reaching a node that is mandatory (inside set M): $N_{r_i} = \{k \in N : \exists w = (w_1, w_2, ..., w_n) \wedge w_1 = n_{i2} \wedge (w_j, w_{j+1}) \in E \wedge w_j \notin M \wedge w_n = k\}$. We call these sets Intermediate Components.

Both sets can easily be computed by using a depth first search starting at the start nodes S for finding the Mandatory Set, or at the nodes $\{n_{i2}\}$ with $r_i \in R, r_i = (n_{i1}, n_{i2})$ for finding the Intermediate Components respectively using

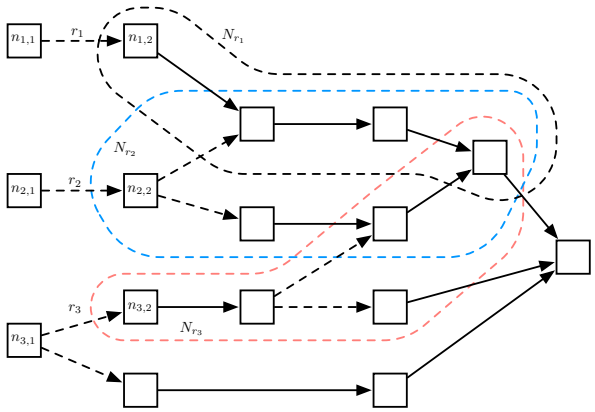

Fig. 3. Example Intermediate Component sets based on definition II.2

the restrictions inside the corresponding definition. As the name Intermediate Component suggests, these sets are used intermediately and form the basis for the final components used inside the reconfiguration process.

C. Component Identification

The initially computed sets $N_{r_1}, ..., N_{r_n}$ already describe which kind of functionality can be executed when the corresponding edge $r_i \in R$ is taken. These sets however may be ambiguous as seen in Figure 3. Using these sets of intermediate components without further refinement could create duplicate code segments, which is highly undesired. In order to resolve all ambiguities, Algorithm 1 can be used to generate distinct components that, while depending on each other, can be used efficiently inside the reconfiguration process. The basic idea is to generate all possible intersections as longs as there exist ambiguous sets of basic blocks. In each intersection iteration (see line five of Algorithm 1), all possible intersections of the current working set S are calculated. Redundant intersections or empty intersections are not stored. At the end of the intersection step all basic blocks contained inside any of the intersection sets K_i are removed from the configurations inside the working set S. The created sets K_i then define the working set for the next intersection step. The iteration ends if the working set contains only one or no set anymore as there exists no possible new intersection that may be computed. The iterations (given by the while loop in line five) of the algorithm on the example graph of Figure 3 can be seen in Figure 4 .

Using Algorithm 1, it is possible to split up the Intermediate Components into single distinct sets of basic blocks. However, this will introduce dependencies between each of the sets, which are defined by the control flow edges between them.

Definition II.3 (Dependency)**:**
Given two components S_i, S_j, if there exists an edge $e = (n_1, n_2)$ with $n_1 \in S_i, n_2 \notin S_i$ and $n_1 \notin S_j, n_2 \in S_j$ we say S_i *directly depends* on S_j, denoted by $S_i \rightarrow S_j$. If there exists a path $w = (w_1, ..., w_n)$ with $w_1 \in S_i, w_2, ..., w_{n-1} \in M, w_n \in S_j$ we say S_i *depends* on S_j, denoted by $S_i \rightsquigarrow S_j$.

The corresponding direct dependencies between components inside the example graph can be seen in Figure 4. The "direct" dependency graph of the initial components can never contain loops due to the construction of it. However, the dependency graph may contain loops as control flow from components may happen to the mandatory set and back. Using the dependency graph, it is possible to estimate the runtime overhead for different paths of the application if the reconfiguration time is known. The execution time of the code inside the components does not change as the application code is not changed. The only source of execution time changes inside the components may result from different caching and pipeline effects which we do not consider.

Algorithm 1 Component Identification

1: **procedure** GENERATECOMPS($N_{r_1}, ..., N_{r_n}$) ▷ Input: N_{r_i} of definition II.2
2: Set $S \leftarrow \{N_{r_1}, ..., N_{r_n}\}$
3: Set $K \leftarrow \{\}$
4: Set $R \leftarrow \{\}$ ▷ The set of output components
5: **while** $|S| > 1$ **do**
6: **for all** $S_i, S_j \in S, S_i \neq S_j$ **do**
7: $T \leftarrow S_i \cap S_j$
8: **if** $T \notin K \wedge T \neq \{\}$ **then**
9: $K \leftarrow K \cup \{T\}$ ▷ Add the set T to K
10: **end if**
11: **end for**
12: **for all** $S_i \in S$ **do**
13: $S_i \leftarrow S_i \setminus \left(\bigcup_{K_i \in K} K_i \right)$ ▷ Remove all sets in K from S_i
14: $R \leftarrow R \cup \{S_i\}$ ▷ Add a new layer of components
15: **end for**
16: $S \leftarrow K$
17: $K \leftarrow \{\}$
18: **end while**
19: **return** R
end procedure

D. Optimal Component Size

The extracted components may now be used for reconfiguration. However, using the components in this state may be far from optimal if we consider factors as runtime overhead, binary overhead or memory fragmentation. Especially in Smart Cards, innately being very resource constraint systems, these overheads need to be kept as small as possible. Many Smart Cards use flash memory for the non-volatile memory space. Most of them also execute applications directly out of flash memory. The use of flash memory inherently raises the demand of reducing the amount of flash page writes at runtime as the lifetime of the memory page is limited by a certain number of erase/write operations. One objective optimization function would thus try to minimize the number of such operations. We are currently only focusing on components which are

 ISBN: 978-1-61208-229-5

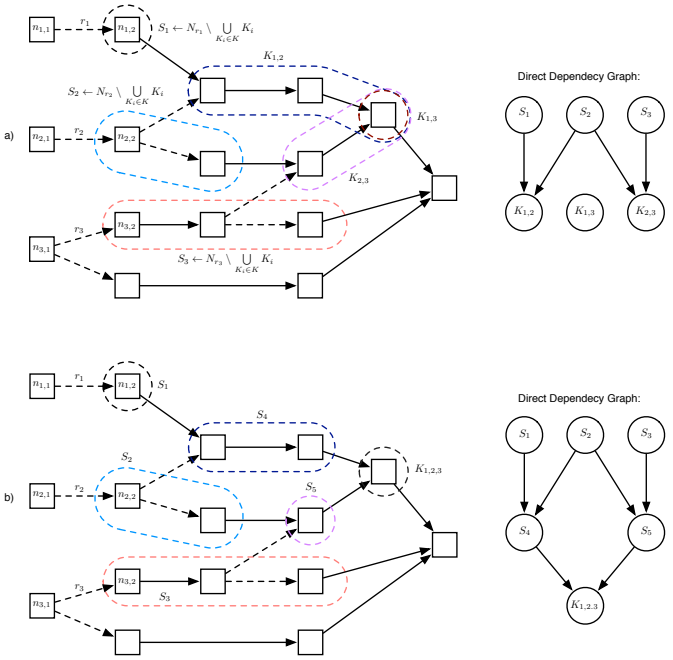

Fig. 4. The iterations of Algorithm 1 on the example graph.

bigger than the minimal flash page size. In order to reduce the number of write operations during reconfiguration, only one component may be placed in one flash page. In addition, the component size is kept as close as possible to multiples of the page size in order to reduce the memory fragmentation at runtime.

This problem is solved by splitting up the components $S_1, .., S_n$ into smaller components. The components $S_1, .., S_n$ with sizes $s_1, .., s_n$ exceeding a size s may be split up into multiple components of size s and one component of size s_i modulo s. This can be done in several ways. One may simply use the linear binary object layout and split the basic blocks at the corresponding positions or one may use a reordering approach which tries to place strongly connected basic blocks close together, just like a compiler would do.

On the one hand, splitting the components will introduce new dependencies resulting in a higher number of reconfiguration edges. On the other hand, this changes the memory fragmentation introduced by each component. Depending on the target system, these attributes will be more or less important. However, an optimal solution to this problem often does not exist because properties as runtime overhead and memory fragmentation are, typically, in contrast to each other and highly dependent on the application. In the next step we thus perform a design space exploration.

Definition II.4 (Design Points)**:**
We define the set of design Points $D_{(m,s)} = (D_B, D_M, D_R)$ as follows: for possible component sizes $s = x \cdot F_{min}$ (with F_{min} being the smallest page size) and total reconfiguration memory space $m = r \cdot s$, the values for the binary overhead D_B and memory fragmentation D_M as well as the worst case number of reconfigurations D_R are calculated.

Our approach of solving this problem is to do a multi-objective optimization by calculating the Pareto optimal points using the Greaf-Younes algorithm [13] with backward iteration over the set of all design points $D_{(m,s)} = (D_B, D_M, D_R)$. Given the total amount of reconfiguration space m it is possible to store r components of size s before runtime replacement (based on some function $f_{replace}$) starts. Using this definition the design space values are calculated in the following way:

- *Binary Overhead D_B:*
  The Binary Overhead is defined as the median percentage based increase of the component size due to added instrumentation code. If the number of jumps / references between components increases, the amount of instrumentation code may increase as well.
- *Memory Fragmentation D_M:*
  For every component size s we sum up the r highest fragmentations of components as this yields the worst case situation with the highest amount of wasted memory.
- *Worst Case Number of Reconfigurations D_R:*
  The worst case number of reconfigurations is the maximum number of reconfigurations needed to execute any possible path inside the context sensitive control flow graph. The path analysis is context-sensitive and done using a Depth First Search approach. Iterating over all possible context-sensitive paths is in general infeasible for even small graphs, however, we use a branch and bound based algorithm to avoid traversing any path which can not increase the worst case reconfiguration number. The value can be calculated more easily by restricting candidate paths to start at the incoming edges of components and to follow a path inside the dependency graph. The function $f_{replace}$ is used to simulate the runtime replacement. If a loop over more than r components is encountered the design point is removed from the design space as we are not able to estimate the number of reconfigurations needed to execute the loop at runtime without knowing loop boundaries.

The final component size s and reconfiguration space m may then be chosen from the set of Pareto optimal points either by the system designer or by using a user defined rating function. The evaluation section will cover an example for determining the optimal component size for our evaluation scenario. It is important to note that increasing the reconfiguration space m will not always lead to better design space points as discussed in the Section Evaluation (IV).

E. Binary Rewriting

As the components are given by the previous steps, we now have to add reconfiguration support to them. The extracted sets of basic blocks usually contain references to relocatable symbols, which would have been resolved at link time of the binary. Relocatable symbols may reference different kinds of application sections as, e.g., the executable code area of other

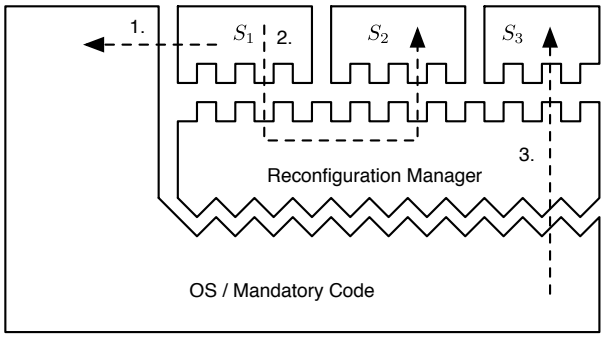

Fig. 5. The reconfiguration architecture and some possible control flows: (1.) control flow from a component to the mandatory set, (2.) control flow from between components, (3). control flow from the mandatory set to a component

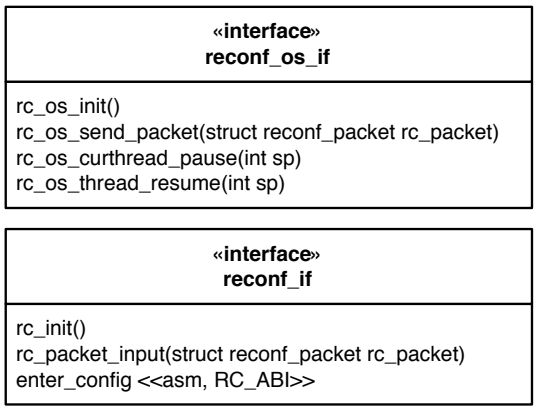

Fig. 6. The interface required/provided by the reconfiguration manager.

components, read-only data or the heap of the component. Relocatable symbols, which reference addresses inside the mandatory set M, are resolved after link time in step *component linking* (see Figure 1). References to other reconfiguration components are replaced by instrumentation code, which adds a call to the reconfiguration manager. Currently, unsupported are references to the data areas of components. The current solution places data areas of the reconfiguration component into the mandatory set, thus allowing the reference to be solved after the mandatory set has been linked.

The instrumentation code is added to the components and placed as close as possible to the corresponding reference in oder to avoid additional overhead of implementing long jumps. The framework also uses the live register information gained by the binary analysis step to use free registers to implement the call to the reconfiguration handler. If all registers are used the context is temporarily stored on the stack. The binary overhead introduced by the instrumentation code may thus vary between four and twenty bytes depending on the free register set and the instruction set (ARM features a 16 bit THUMB and a 32 bit ARM instruction mode). The overhead introduced for a realistic example is discussed in the evaluation section.

The components themselves also need to be rewritten. This is required as references to other basic blocks inside the components may be invalid due to the fact that basic blocks may have been added, removed or changed. Thus, all instructions containing references to other basic blocks inside the component are updated. The same holds true for symbol and relocation entries inside the binary which are used by the linker. The process of modifying these offsets is described inside [11].

III. Run-Time Reconfiguration

At runtime, transparently to the user, the components are exchanged on demand. The architecture depicted in Figure 5 describes the possible control flows and the interfaces involved can be seen in Figure 6. The reconfiguration manager is the central part of the reconfiguration process. In order to work properly, the interface `reconf_os_if` needs to be implemented by the operating system. Among others, it defines a method which is used to send reconfiguration packets to the reconfiguration server, which runs for example on a terminal connected to the smart card. In our example scenario we use a UDP based connection.

Basically, three types of control flows needs to be covered by the system. Control flow going from a component to the mandatory code / OS (see 1. in Figure 5) is already covered by the binary rewriting process. As the mandatory code is not moved inside the physical address space, the corresponding branches are handled by the instrumentation code. The runtime overhead of these control flows is static. Control flow occurring between components (see 2. in Figure 5) involves the reconfiguration manager. Let us consider the components S_1 and S_2 inside the figure. The component S_1 needs to execute some code in component S_2. As the physical memory location of the component changes and/or the component may not be loaded, the reconfiguration manager is called first. The reconfiguration manager provides an assembler routine named `enter_config`, which is automatically called by the instrumentation code added to the components. The assembler routine takes the ID and the offset of the code to be executed inside the component as parameters. If the component is loaded the call is forwarded and the code is executed. This takes around ten assembler instructions on the ARMv4 THUMB ISA. If the component is currently not loaded a reconfiguration request is issued which will use the interface to the operating system to reload the component. The issuing thread context is saved on the stack and stored by the reconfiguration manager to resume the thread upon completion of the reconfiguration. Control flow from the mandatory code to a component (see 3. in Figure 5) involves the same steps as the previous one. It is handled by the same assembler routine and the call to the reconfiguration manager is also automatically added to the mandatory code by the binary rewriting step.

The replacement strategy $f_{replace}$ is implemented inside the reconfiguration manager and used to determine which component will be replaced at runtime. The current implementation uses a *least frequently used* (LFU) replacement function. For

 ISBN: 978-1-61208-229-5

Design Flow Step	Execution Time
Header Analysis	7929 ms
CFG Generation	4988 ms
DF Analysis	33921 ms
Constraint Checking	264 ms
Component Identification	297 ms
Binary Rewriting	963 ms
Component Optimization	6560 ms

TABLE I
EXECUTION TIME OF THE DESIGN FLOW STEPS FOR THE EXAMPLE SCENARIO.

every reconfiguration edge taken a counter is increased for the corresponding component. If replacement takes place the component with the smallest counter is removed. The LFU algorithm was chosen as the implementation overhead of this algorithm is very small.

IV. EVALUATION

This section gives an evaluation of the binary reconfiguration approach.

A. Case Study

Our case study is an Internet-Protocol Stack library for an ARM powered SmartCard containing protocol implementations for the Internet Protocol Version 4 (IPv4), Version 6 (IPv6), the Transmission Control Protocol (TCP), User Datagram Protocol (UDP) and a Transport Layer Security (TLS) implementation. The Smart Card contains a USB interface, which we use to emulate an ethernet connection using the Ethernet Emulation Model (EEM). With this USB connection the Smart Card is connected to a Linux computer, which uses the Ethernet interface to transparently communicate with the device. The scenario contains a smart card web-server, which offers communication ports using all of these protocols of the library. However, at runtime not all protocols are used at the same time which makes it interesting to use the corresponding protocols as reconfiguration components. The complete binary size of all binary objects inside the case study consists of 44246 Bytes.

B. Design Time Overhead

Our binary reconfiguration framework has been implemented in Java and supports the ARMv4 ISA. However, the software architecture allows for the easy addition of support for other ISAs. The reconfiguration process was executed on a Linux computer with a single core 2,8 Ghz Pentium processor. The execution time of the design flow steps can be seen in Table I. The most time consuming part is the Data-Flow (DF) Analysis which annotates the CFG with high level constraints and resolves indirect branches. The Component Optimization step which includes the calculation of the worst case reconfiguration amount for all design points only took about seven seconds to complete. All together the complete execution time of the framework stayed under one minute which is a reasonable time frame.

Component	Component Size	Complete Size	Percentage
S_1 (TLS)	6948	10252	67,7 %
S_2 (IPv6)	1046	2024	51,6 %
S_3 (TCP)	4136	10468	39,5 %

TABLE II
EXTRACTED COMPONENT SIZES IN BYTES

C. Reconfiguration Manager Overhead

As the reconfiguration itself adds new executable code to the original binary, it is very important to keep this additional code as small as possible. The implementation of the reconfiguration manager including the interface implementation and the replacement function added an additional 680 Bytes of code to the application. Inside the example scenario the communication stack of the operating system could be reused resulting in a small reconfiguration manager.

D. Component Extraction

The XML configuration file contained the constraints shown in listing 1, which were passed to the constraint solver with the goal to extract the TCP, IPv6 and TLS components from the application in order to reuse these components inside the reconfiguration process. Line two and four describe a constraint to identify the control flow to the TCP component, line six specifies the control flow to the IPv6 component and the symbol constraint in line eight describes an entry point to the TLS component. Table II shows the size of the extracted components S_i after using Algorithm 1.

Using this simple constraint set, it was possible to extract 68% of the TLS implementation code to be used inside a reconfiguration component. The remaining bytes of the implementation may be extracted with a more sophisticated constraint set as not all control flows are covered by the set of Listing 1. A similar statement holds true for the TCP and IPv6 components in Table II for which the percentage is lower. This is due to the fact that the constraints only restricts control flow from the lower Ethernet packet layer. Control flow from higher layers, as, e.g., the application layer, was not considered by the constraint set. This is, however, possible without restriction.

E. Component Optimization

In the next step, the design points $D_{(m,s)} = (D_B, D_M, D_R)$ have been calculated. The basic blocks have not been reordered and were placed inside a component based on their linear order inside the object files they were taken from. For every combination of component size and reconfiguration space (m, s) the resulting worst case number of reconfigurations, binary overhead and memory fragmentation has been calculated. Table III gives an overview of some of the calculated points sorted by number of reconfigurations as this has been most interesting for our scenario. The Pareto optimal points have been highlighted. All of the highlighted design points are equally good with respect to the pareto optimality. However, depending on the target architecture and the execution scenario one may favor design points with a small numbers of reconfigurations or low fragmentation.

 ISBN: 978-1-61208-229-5

m	s	D_R	D_B	D_M
4096	256	17	28,06252967	530
...	...	...	...	...
4096	1024	15	19,07260033	1654
3072	512	12	21,420201	954
...	...	...	...	...
4096	512	10	21,420201	954
8192	1024	8	19,07260033	1654
4096	2048	7	14,56403317	2134
...	...	...	...	...
2048	2048	7	14,56403317	1372
5120	1280	7	16,03623833	2166
5120	5120	7	12,3342234	3834
6144	1536	6	16,813032	494
6144	2048	6	14,56403317	2694
7168	1792	5	15,73381667	2110
4096	4096	5	13,16168417	3336
6144	3072	5	15,5311505	3490
...	...	...	...	...
6912	6912	4	12,29584317	5626
7168	3584	3	14,45041367	3394
7168	7168	3	12,21908233	5882

TABLE III
CALCULATED DESIGN POINTS $D_{(m,s)} = (D_B, D_M, D_R)$. THE PARETO OPTIMAL ONES ARE HIGHLIGHTED.

The binary overhead for our evaluation example stayed between 30% and 12% (compare the values D_B of Table III), which is a reasonable increase in code size of the components. We used the design point $D_{(2048,2048)}$ for the evaluation, which resulted in a final component size $m = 2048$ bytes and reconfiguration memory size $s = 2048$ bytes. Using this design point limited the number of concurrent components on the Smart Card to one. Thus, each time a dependency edge is taken between two components a reconfiguration needs to take place. The worst case path consisted of seven reconfigurations. Using the UDP connection to the Linux computer running the reconfiguration server the maximum reconfiguration time for one component (2048 Bytes) took 153 ms. A connection request to the web-server was delayed by a median value of 920 ms. This demonstrates that it is possible to run the web-server application on the Smart Card with a memory requirements of 35 KB. This results in a memory saving of approximately 22%. However the memory saving is paired with a much higher run-time of the application.

Interesting to see is that simply increasing the amount of reconfiguration memory space does not always yield better design points. This can be seen by comparing $D_{(4864,4864)}$ with $D_{(2816,5632)}$. Although the latter design point offers a higher amount of reconfiguration space the system parameters are worse. This shows that the design optimization step is mandatory if the system parameters need to be optimized and/or known beforehand.

V. RELATED WORK

Many approaches have been created to solve parts of the goals described in this thesis. Link-Time optimization approaches [14][15] allow binary code to be optimized for speed and memory requirements. This is a valuable technique which already grants huge benefits for software programs. However, it does not solve the general problem if the execution space is still too small for an application to run on an embedded device. It also is not intended as an approach which allows software programs to be adapted at runtime.

Binary Analysis approaches have been used for many reasons for years. Most of the efforts are concerned with analyzing source code which is not available in the contents of this approach. Approaches which analyze binary code are focused on different problems. On the one hand they are used for the link-time optimization described above. On the other hand it is used to cope with security issues of applications [16], [17] quality assurance or compliance testing [18]. Recently Binary Analysis and Binary Rewriting gained popularity inside the research community again. Modern run-time compiler use data flow analysis techniques do to optimizations by using, e.g., trace-scheduling techniques [19][20]. In this approach, we use binary analysis techniques for a different purpose: it is used to gather information on the binary objects, which will enable run-time reconfiguration of binary objects. To the best of our knowledge, there exists no approach which tries to use binary analysis to support software reconfiguration of legacy software systems.

Run-Time Reconfiguration approaches have been proposed for small embedded devices for different goals. It has been shown to be indispensable for some kinds of applications as it allows for re-tasking, fixing bugs, adding functionality or replacing functionality due to memory restrictions. Reconfiguration is supported by some operating systems, particularly often used inside sensor-networks. For example, Agilla [21] or TinyOS [22] support some form of reconfiguration. However the reconfiguration either consist of full binary upgrades (TinyOS) or requires the source code (Agilla), which makes the use of legacy code impossible. Other forms of binary adaptation may be categorized by the following approaches. Whabe et al. [23] propose the creation of adaptable binaries by adding information to the binaries, which may then be used to modify the binary later on. The approach in [24] is based on using new architectures and creating adaptable and reloadable components on source code level. A promising approach has been shown in [25] by creating so called ”delta files”, which contain the byte streams of the adaptations to be made on binary level. However, the delta files are created by compiling the adaptations from source code for the different kinds of configurations. All these approaches have in common that they cannot be used with proprietary libraries that already have been compiled and may not be rebuilt with these kind of information or adaptation support.

VI. CONCLUSION AND FUTURE WORK

In this paper, we demonstrated an approach which allows binary objects as they are contained inside legacy libraries to be used as components inside a reconfigurable system. By using control flow and data flow analysis techniques the approach derives a high level representation of the binaries. Combined with a constraint satisfaction problem solver the system allows the user to easily define components based on

 ISBN: 978-1-61208-229-5

high level constraints. The components are then optimized regarding parameters as number of reconfigurations at runtime, binary overhead due to the added instrumentation code and memory fragmentation. In the end, the approach allows these components to be seamlessly added and removed at runtime. The evaluation showed that it is possible to extract components out of the binary objects. The optimization step derives a Pareto optimal set of design parameters. In the end the legacy objects can be used as components inside a reconfigurable system which can have a lower memory consumption while maintaining the functionality offered by the legacy objects.

Future adaptations may further increase the benefit gained as in the presented work we only used a simple linear basic block scheduling technique inside the components. Using more sophisticated placing algorithms may further decrease the binary and reconfiguration overhead. We may also consider profile based information to better identify components and to identify heavily used program paths. This may allow the design space exploration to find better design points, which will decrease the median number of reconfigurations.

REFERENCES

[1] "RFC 791 - Internet Protocol Version 4." [Online]. Available: http://www.ietf.org/rfc/rfc791.txt

[2] "RFC 2460 - Internet Protocol Version 6." [Online]. Available: http://www.ietf.org/rfc/rfc2460.txt

[3] "RFC 793 - Transmission Control Protocol." [Online]. Available: http://www.ietf.org/rfc/rfc793.txt

[4] "RFC 768 - User Datagram Protocol." [Online]. Available: http://www.ietf.org/rfc/rfc768.txt

[5] "RFC 4346 - Transport Layer Security Version 1.1." [Online]. Available: http://www.ietf.org/rfc/rfc4346.txt

[6] ARM Ltd., "ELF for the ARM Architecture," 2009.

[7] C. Cifuentes and M. V. Emmerik, "Recovery of jump table case statements from binary code," in *Science of Computer Programming*, 1999, pp. 2–3.

[8] B. D. Sutter, B. D. Bus, K. D. Bosschere, P. Keyngnaert, and B. Demoen, "On the static analysis of indirect control transfers in binaries," in *In PDPTA*, 2000, pp. 1013–1019.

[9] C. Cifuentes, "Interprocedural data flow decompilation," *Journal of Programming Languages*, vol. 4, pp. 77–99, 1996.

[10] C. Cifuentes, D. Simon, and A. Fraboulet, "Assembly to high-level language translation," in *In Int. Conf. on Softw. Maint*. IEEE-CS Press, 1998, pp. 228–237.

[11] D. Baldin, S. Groesbrink, and S. Oberthür, "Enabling constraint-based binary reconfiguration by binary analysis," *GSTF Journal on Computing (JoC)*, vol. 1, no. 4, pp. 1–9, January 2012.

[12] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Computer Aided Verification (CAV '07)*. Berlin, Germany: Springer-Verlag, July 2007.

[13] J. Jahn, "Vector optimization, theory, applications and extensions." Springer-Verlag, 2011, p. 345.

[14] D. W. Goodwin, "Interprocedural dataflow analysis in an executable optimizer," 1997.

[15] W. E. Weihl, "Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables," in *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '80. New York, NY, USA: ACM, 1980, pp. 83–94. [Online]. Available: http://doi.acm.org/10.1145/567446.567455

[16] N. Xia, B. Mao, Q. Zeng, and L. Xie, "Efficient and practical control flow monitoring for program security," in *Proceedings of the 11th Asian computing science conference on Advances in computer science: secure software and related issues*, ser. ASIAN'06. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 90–104. [Online]. Available: http://dl.acm.org/citation.cfm?id=1782734.1782742

[17] D. Wagner and D. Dean, "Intrusion detection via static analysis," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, ser. SP '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 156–. [Online]. Available: http://dl.acm.org/citation.cfm?id=882495.884434

[18] R. Venkitaraman and G. Gupta, "Static program analysis of embedded executable assembly code," in *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, ser. CASES '04. New York, NY, USA: ACM, 2004, pp. 157–166. [Online]. Available: http://doi.acm.org/10.1145/1023833.1023857

[19] N. V. Mujadiya, "Instruction scheduling for vliw processors under variation scenario," in *Proceedings of the 9th international conference on Systems, architectures, modeling and simulation*, ser. SAMOS'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 33–40. [Online]. Available: http://dl.acm.org/citation.cfm?id=1812707.1812717

[20] E. Yardimci and M. Franz, "Mostly static program partitioning of binary executables," *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 5, pp. 17:1–17:46, Jul. 2009. [Online]. Available: http://doi.acm.org/10.1145/1538917.1538918

[21] C.-L. Fok, G.-C. Roman, and C. Lu, "Agilla: A mobile agent middleware for self-adaptive wireless sensor networks," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 3, pp. 16:1–16:26, Jul. 2009. [Online]. Available: http://doi.acm.org/10.1145/1552297.1552299

[22] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," *SIGPLAN Not.*, vol. 35, no. 11, pp. 93–104, Nov. 2000. [Online]. Available: http://doi.acm.org/10.1145/356989.356998

[23] R. Wahbe, S. Lucco, and S. L. Graham, "Adaptable binary programs," IN, Tech. Rep., 1994.

[24] S. Kogekar, S. Neema, and X. Koutsoukos, "Dynamic software reconfiguration in sensor networks," in *Proceedings of the 2005 Systems Communications*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 413–420.

[25] R. Keller and U. Hölzle, "Binary component adaptation," in *Proceedings of the 12th European Conference on Object-Oriented Programming*. London, UK: Springer-Verlag, 1998, pp. 307–329.

 ISBN: 978-1-61208-229-5