Abstract
Code bloat widely exists in production-run software. Left untackled, it not only degrades software performance but also increases its attack surface. In this work, we conduct a case study to understand this issue in statically linked libraries. To be specific, we analyze midilib, a software package enclosing a statically linked library. We show that it is possible to leverage data dependency analysis to trim the unnecessary code state­ment residing in a target library. With this observation, we believe it is possible to build a tool to automatically cut off code pertaining to unnecessary operations.

1 Introduction
Modular design is widely used in traditional software development to control the implementation complexity [17]. After dividing a large program into smaller modules or libraries, developers can focus on their own parts and implement desired functionalities. To improve development productivity, libraries are encouraged to be reusable and to be shared by different programs [15]. Therefore, libraries tend to have generic interfaces and provide different functionalities for various usage scenarios.

When a library is used by a program, it is usually statically linked to that program. Since the program only has limited calling contexts and usage scenarios, more-than­necessary code inside the library is linked, causing code bloat [14]. Bloated code widely exist in production-run software. For example, a recent study shows that only around 20% instructions of Firefox are executed under typical workloads [10].

Bloated code can lead to various problems. First, it potentially introduces more bugs and vulnerabilities. A recent study has showed that most vulnerabilities in protocol implementation reside in modules not widely used [5]. Second, a larger code size increases memory pressure and causes cache misses when loading instructions [1]. Third, bloated code incurs resultless or redundant computation, resulting in computation inefficiency [2, 20, 21]. Last but not least, a larger code size also consumes network bandwidth when being distributed across the Internet [11, 12].

Inspired by this, we propose to address the code bloat­ing problem by performing library customization against statically linked libraries. Different from previous works on detecting runtime bloat [2, 3, 9, 19, 21] or library cus­tomization [5, 8, 11, 13], our technique is a fine-grained code removal scheme built on the basis of the following hypothesis. Many library functions return its computa­tion results as a data object defined by a “struct”. How­ever, it is typical the case that many fields in the struct are not in use by the upper level applications. This means that there must be unnecessary computation residing in the library and we have the potential to trim the code pertaining to the unnecessary computation.

To validate our hypothesis, we conduct a case study against a software package which contains statically linked library. We show that a library returns a data object with 44 primitive fields. However, upper level software only uses 9 of them. By using data dependency analysis along with two code trimming schemes, we can reduce the code space of the library by about 50%. Given that many software statically linked library, we believe this observation and practice could be potentially gener­alized and significantly benefit library customization.

The rest of the paper is organized as follows. In Section 2, we discuss our case study against a software pack­age and describe two proof-of-concept techniques. In Section 3, we discuss the works relevant to library cus­tomization. In Section 4, we conclude our paper and dis­cuss future works.

2 Case Study
As is mentioned above, a shared library statically linked might contain many unnecessary operations that can be potentially trimmed. In this section, we illustrate this practice by taking for example midilib [4] an open­source repository in C.

Midilib contains an implementation of I/O libraries for MIDI files [16] in midifile.c. It also provides other functionalities. After building midilib, we can obtain five executables, which are m2rtttl, to convert MIDI files to RTTTL [19], mididump, to dump MIDI file content, mfc120, to change MIDI file version, mozart, to generate simple musics, and miditest, to conduct tests. The compiled object file of midifile.c is statically linked to every executable. Thus, each executable contains the same library.

Inspired by this, we propose to address the code bloat­ing problem by performing library customization against statically linked libraries. Different from previous works
More specifically, we perform our analysis on the executable m2rtttl as well as the library implemented in midifile.c. Second, we introduce a prototype system to demonstrate the potential of debloating the library. It should be noted that we do not claim our prototype system is an effective solution for dealing with the library debloating. Rather, it is just a preliminary proof-of-concept tool, demonstrating the possibility of reducing the code space for the library.

### 2.1 Code Analysis and Key Observation

The major functionality of m2rtttl is implemented using a while loop shown in Table 1. In each iteration, the while loop reads a midi packet from an input file on line 2 and changes the packet to a string in RTTTL format on line 8. The function midiReadGetNextMessage is implemented in the shared library midifile.c. It parses an input file, indicated by its argument mf, and returns midi packets through argument msg. In midilib repository, msg is in MIDI_MSG struct type. As is shown in Table 2, MIDI_MSG contains an enum field iType, indicating the type of a midi packet. In addition, MIDI_MSG contains a union field MsgData to hold data for different types of midi packets.

Table 1: The code fragment of the while loop implementing the major functionality of m2rtttl.

Table 2: The code fragment of struct MIDI_MSG.
To be specific, we first leverage the struct layout information provided by LLVM to compute the offsets of the fields that neither the library nor the application reads. Then, using this information, we pinpoint those assignment instructions (i.e., LLVM intermediate code) corresponding to these fields. Since the assignment instructions represent the site of assigning a value to a field, and operations pertaining to such sites also contain those instructions that compute the assigned value and the field address, we finally perform a simple data dependence analysis to further identify the instructions relevant to the field assignment. In this preliminary work, we deem such instructions as unnecessary and our tool trim these instructions along with those tied to resultless value assignments.

In the implementation of the library function midiReadGetNextMessage, our tool tracks down 4 fields, the read of which neither presents in the library nor the upper level of the application. These 4 fields associate with 51 instructions in the library, accounting for about 7% (51 out of 722) LLVM intermediate code that we can safely eliminate. It is not difficult to note, as is mentioned in Section 2.1, we have identified only 9 fields that the upper level of the application reads, but our tool tracks down only 4 fields, the read of which are not presented in the application. Here, the reason is as follows.

As is illustrated in Figure 2, many primitive fields are enclosed in the union type field MsgData. From the perspective of LLVM, this means that the machine utilizes the same memory location to store different struct fields, such as NoteOff, NoteOn, NoteKeyPressure, and so on. In our implementation, our tool distinguishes primitive fields based on their offsets. This means it lacks the ability to distinguish the primitive fields referred by the union struct, such as failing to differ mg.MsgData.NoteOn.iNote from mg.MsgData.NoteOff.iNote. Therefore, our current results miss to pinpoint some primitive fields that neither library nor application reads.

Tool for Eliminating Unused Packet Types. Recall that in addition to leveraging resultless field assignment for library code customization, we can use the packet type as an indicator to identify resultless operations in the library. Motivated by this observation, we design and develop another tool that takes advantage of this observation and performs library code customization.

```
1 /* midifile.c */
2 BOOL midiReadGetNextMessage(..., MIDI_MSG *msg)
3     
4 switch(msg.iType) {
5     case msgNoteOn:
6         msg->MsgData.NoteOn.iChannel = ...;
7         msg->MsgData.NoteOn.iNote = ...;
8         msg->iMsgSize = 3;
9         break;
10     case msgNoteOff: ... 
11         break;
12     case msgNoteKeyPressure:
13         msg->MsgData.NoteKeyPressure.iChannel = ...;
14         msg->MsgData.NoteKeyPressure.iNote = ...;
15         msg->MsgData.NoteKeyPressure.iPressure = ...;
16         msg->iMsgSize = 3;
17         break;
18     case msgSetParameter: ...
19         break;
20     case msgSetProgram: ...
21         break;
22     case msgChangePress: ...
23         break;
24     case msgSetPitchWheel: ...
25         break;
26     case msgMetaEvent: ...
27         break;
28     case msgSysEx1: ...
29     case msgSysEx2: ...
30         break;
31     }
32     ptr += msg->iMsgSize
33     ...
34 }
```

Table 3: The code fragment of the library function midiReadGetNextMessage.

1700 write operations, we partition them into two categories. In the first category, we observe that, there are 1015 write operations (about 60%) that neither the lower level of the library nor the upper level of the application read the fields tied to these operations, which implies that we can safely trim these operations and thus reduce the code space of the library. In the second category indicated by the remaining 685 write operations (approximately 40%), we note that, while the fields tied to these operations are not read by the upper level of the application, they are involved in data dependency in lower level of the library. Admittedly, this does not mean we cannot remove these operations. But it implies that we have to trim these operations with the consideration of data dependency.

2.2 Customization Demonstration

Based on the analysis and observation above, we design and develop two simple tools using LLVM to customize the lower level of the library or more precisely speaking the function midiReadGetNextMessage implemented in the library. In the following, we describe them in turn.

Tool for Eliminating Resultless Field Assignments. As is mentioned above, if the value assigned to a field is not accessed by the upper level of the application nor the lower level of the library, then we could safely remove the statements tied to such assignment. Inspired by this observation, we develop a tool to identify and cut off such code statements.

To be specific, we first leverage the struct layout information provided by LLVM to compute the offsets of the fields that neither the library nor the application reads. Then, using this information, we pinpoint those assignment instructions (i.e., LLVM intermediate code) corresponding to these fields. Since the assignment instructions represent the site of assigning a value to a field, and operations pertaining to such sites also contain those instructions that compute the assigned value and the field address, we finally perform a simple data dependence analysis to further identify the instructions relevant to the field assignment. In this preliminary work, we deem such instructions as unnecessary and our tool trim these instructions along with those tied to resultless value assignments.
As is shown in Table 3, the library assigns a value to a field (e.g., msg.MsgData.NoteKeyPress.iChannel in line 13) only after it reads another field (e.g., msg.iType in line 4) and a certain condition holds (e.g., msg.iType == msgNoteKeyPress in line 5). Intuition suggests this can be interpreted as a control dependency. In the higher level of the application, as is shown in Table 1, we do not observe the same control dependency relationship. This means that the corresponding computation in the library (i.e., the statements in line 13-15) has no effect upon the upper level of the application. We could leverage this mismatched pattern to cut off the code fragment accordingly and thus reduce the code size of the library. It should be noted that we do not trim the statements in line 16-17 nor that in line 12 depicted in Table 3 because – as is specified in line 32 – the global variable ptr is dependent upon iMsgSize.

In this work, we implement this pattern matching approach using LLVM. By performing code customization against the aforementioned library through the patterns identified, we track down 33 field assignment instructions in the library that do not have impact to the upper level of the application. Following the procedure used in the first tool mentioned above, we also use data dependency analysis to pinpoint other statements pertaining to those tied to unnecessary filed assignments. Together with the unnecessary 33 instructions, in total, we pinpoint 355 out of 722 LLVM instructions that can be safely eliminated.

Going beyond testing the tools individually, we further combine the unnecessary instructions obtained from both aforementioned tools. We observe the combined techniques can identify 36 unnecessary field assignments in total. Using data dependency analysis, they lead up to the removal of 367 LLVM instructions, accounting for 50.83% instruction removal (i.e., 367 out of 722). We have already noted that these about 50% of instruction removal reflects approximately 39.63% lines of source code removal. This indicates the potential of fine-grained library customization.

3 Related Works

Code bloat [14] refers to unnecessarily large code size, which can increase security attack surface, consume more memory, lower instruction cache performance, and even make the distribution of software more difficult. There are empirical studies that confirm the existence of code bloat and its negative impact. Quach et al. [10] conducts an empirical study to understand how much unused code in different types of programs. The authors propose two methods to measure the size of unused code, one is to identify function calls isolated from call graph statically, and the other is to dynamically profile how many instructions are not executed under typical workloads. The authors report that a large portion of code is not executed in the investigated programs. Hong et al. [5] studies 20 vulnerabilities related to protocol implementation, and finds that most of the vulnerabilities reside in code implementation not commonly used. Customizing protocol implementation can successfully eliminates most of these vulnerabilities.

Researchers and practitioners build many techniques to identify and eliminate unnecessary functionalities [5-8][11,13]. LDoctor [13] identifies inefficient loops conducting resultless computation and suggests developers remove these loops conditionally or unconditionally. Hong et al. [5] proposes a feature access control system to unify protocol implementation customization, which can remove unnecessary features in protocol implementation and reduce attack surface. Application containers usually contain unneeded files. As a dynamic technique, Cimplifier [11,12] can automatically detect unnecessary resources through analyzing system calls. JRed [8] detect unused classes and methods using reachability analysis after building call graphs for programs to be customized. Jiang et al. [6,7] proposes a technique to cut user-specified functions through backward and forward slicing. Although useful, these techniques work on code granularity much larger than our proposed techniques, such as loop or files. These techniques do not target to eliminate fine-grained resultless computation.

4 Conclusion and Future Work

In this paper, we perform a code analysis against a library statically linked to a target executable. We show that using the fields in a data structure we can potentially trim the code statements resultless for the upper level of applications, and thus potentially reduce the attack surface of the library. Using two proof-of-concept tools designed and developed based on our analysis, we demonstrate the possibility of performing fine-grained customization for a library.

As future work, we will extend our techniques from the following aspects. First, we plan to examine more libraries and conduct an empirical study to understand root causes of resultless field assignments and their impact in the real world. Second, we plan to build a robust static technique and explore different design points during technical design. Third, we plan to build an automated testing platform combining static and dynamic analysis to test customized libraries.
References


